Journal of Engineering Technology and Applied Physics

Integration of Multiplication and Floating-Point Extensions in Risc-V: Design and Synthesis

Tasigen Visvanathan, Chu Liang Lee*, Kah Yoong Chan and C. Senthilpari
Faculty of Artificial Intelligence and Engineering (FAIE), Multimedia University, Cyberjaya, Malaysia.
**Corresponding author:* cllee@mmu.edu.my, *ORCiD*: 0009-0004-6837-1163
https://doi.org/10.33093/jetap.2026.8.2.8
Manuscript Received: 15 July 2025, Revised: 6 September 2025, Accepted: 13 January 2026, Published: 15 September 2026

***Abstract*—This project involves the design and implementation of the multiplication (M) and floating point (F) extensions and their integration with the base RV32I processor in forming an RV32IMF processor. The multiplication and division unit (MDU) and floating-point unit (FPU) are designed to support these extensions. The MDU is a single module that handles multiplication and division between combinations of signed and unsigned integers. The FPU design is hierarchical with several submodules, such as the AddSubFPU, MulFPU, and DivFPU modules. The single AddSubFPu module handles addition and subtraction. MulFPU and DivFPU handle the multiplication and division, respectively. Additionally, FPU's top-level module handles sign-injection instructions. FPU Arithmetic demonstrates high accuracy, with relative error limited to around 0.25%. Performance analysis shows an increase in area and power usage compared to the baseline design, along with improved critical path timing. The processor demonstrates stable operation at various frequencies, with one hundred megahertz offering a favourable balance of performance and efficiency.**

***Keywords*—RISC-V, FPU, MDU, VCS, DC, ASIC.**

I. INTRODUCTION

The RISC-V Instruction Set Architecture (ISA) has its hallmarks from the University of California in Berkeley, in the year 2010, with its initial purpose for supporting academia and education in the computer architecture subject area. However, it has proliferated beyond its initial objective and serves as a promise of revolutionizing the way we see ISAs. Legacy instruction set architectures (ISAs) are bound by proprietary licensing and limitations in customizability. RISC-V aims at championing free ISAs as opposed to the exorbitant proprietary ISAs. While the base RISC-V adopts a minimalistic integer base instruction set (RV32I), its manual defines "M" and "F" extensions for multiplication and single-precision floating-point operations, respectively.

The architecture was designed to support extensive customization and specialization with the base Integer ISA, which should not be redefined [1]. The base integer ISA is named as "I" and will be prefixed. For example, RV32I means a RISC-V 32-bit instruction length base integer processor. The base RISC-V ISA has a little-endian memory system since it is dominant in 8 commercial ISAs and more natural for hardware designers [1]. Despite its standardized specifications in the RISC-V manual for these extensions, many open-source RISC-V implementations omit them to avoid hardware complexity in the design, difficult verification, and area overhead. This is true especially in academic cores and minimalist processors, which do not provide support for these extensions. To add on, synthesizable RTL implementation for these extensions is not provided by the RIS-V foundation, thus requiring custom development of these extensions.

VLSI designers have proposed numerous designs to implement floating-point units with a trade-off in their performance, consumption of power, area, and IEEE 754 standard compliance. The designs often focus on the application at hand and relate to its energy efficiency, throughput, and flexibility in order to support various floating-point arithmetic and dedicated instructions. Table I shows the different design approaches across different literatures.

The reviewed works show that RISC-V processor designs exhibit significant variation in timing, area, and power depending on process technology, synthesis constraints, and architectural choices, as shown in Table II. The main contribution of this work is the

Published by MMU PRESS. URL: https://journals.mmupress.com/index.php/jetap/index

synthesis of a complete RV32IMF core using the SAED32 nm library. Unlike prior studies, which focused on integer-only cores [8] or fused FPUs [9], this work demonstrates an end-to-end IMF integration, establishing a realistic reference for area, power, and timing trade-offs at the 32 nm node. In contrast to Koay Yenn Nee's synthesis of an integer-only RISC-V processor at 32 nm, this work presents the first complete RV32IMF core synthesized at the same technology node, thereby incorporating both multiplication/division and floating-point functionality.

Table I. Literature review of FPU design approaches.

Authors	**Papers Title**	**Focus/Design Approach**
Monika Maan, Abhay Bindal	A Review of IEEE-754 Standard Floating Point Arithmetic Unit	Review of Various architectural floating-point arithmetic by surveying available papers [2].
Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini	FPnew: An Open-Source Multi-Format Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing	Transprecision Computing [3]
Krste Asanovi´c, Rimas Avi˙zienis, Jonathan Bachrac, Scott Beamer, et al.	The Rocket Chip Generator	Parameterized SoC using Chisel [4]
Mate Kovac, Leon Dragi´c, Branimir Malnar , et al.	FAUST: Design and implementation of a pipelined RISC-V vector floating-point unit	A pipelined FPU for vector processing capabilities [5]
Shivam Aggarwal, Hans Jakob Damsgaard, Alessandro Pappalardo, et al.	Shedding the Bits: Pushing the Boundaries of Quantization with Minifloats on FPGAs	Custom reduced precision floating-point format [6]

In short, this project aims to address the gap by designing, integrating, and synthesizing the M and F extensions for the RISC-V using Verilog HDL with the aid of Synopsys EDA tools. The RISC-V core with base integer instruction set was obtained from T. Halvadia's open-source repository [7].

Table II. Literature review of synthesis parameters.

Authors	Paper Title	Clock period (ns)	Area (μm^2)	Timing (ns)	Power(μW)
Quentin Madariaga	Development of a RISC-V Microprocessor for ASIC Integration [8]	25	2480000.00	5.482	1.25×10^4
Winlog Feng , Huangxu Chen , Guozheng Lin ,	An Area and Power Efficient Design of Fused integer-floating point unit for RISC-V cores [9]	N/R	24739000000.00	N/R	0.065×10^4
Rafael da Silva	Synthesis of Steel-ASIC, a RISC-V Core [10]	51ns	15793.46	0	1.01×10^4
Koay Yenn Nee	Front End Design (Logic Synthesis) of RISC-V Processor Using Design Compiler [11]	5	29594.21	1.77	1.1805× 10^4

II. "M" EXTENSION DESIGN

The M extension introduced provides additional arithmetic operations such as integer multiplication and division. The extension enables multiplication and division to be performed with a single instruction. In contrast, software emulation requires several RISC-V base instructions and consequently takes a significant number of cycles.

The "M" extension uses single instructions through dedicated hardware, thus handling fewer clock cycles and improving the performance of the RISC-V when handling multiplication and division operations.

A. "M" Extension Module

The module introduced for the "M" extension was called MDU, which is short for Multiplication and Division Unit. Figure 1 shows the block diagram of the MDU along with its input and output, which provides dedicated hardware support for RISC-V M-extension. The summary of the multiplication and division operations to their corresponding control bits from "function3" is shown in Table III below. Figure 2 below indicates the dataflow diagram for the MDU module, which provides the design guide in order to write the Verilog HDL for the MDU. Based on "function3", the corresponding operations are performed and stored in the "mdu_result". This control logic was achieved by utilizing the case statement in Verilog, with "function3" being the selection condition. To design the multiplication and division of signed and unsigned numbers, direct castings such as the "$signed ()" and "$unsigned ()" which is synthesizable.

Fig. 1. Block diagram of MDU module.

Table III. The function3 and the corresponding MDU operation.

function3	Operation
000	MUL
001	MULH
010	MULHSU
011	MULHU
100	DIV
101	DIVU
110	REM
111	REMU

B. Results and Discussion

Figure 3 shows the waveform simulation in VCS Verdi for the MDU module with the "mdu_result" in decimal format, while Fig. 4 shows a similar waveform simulation but with "mdu_result" in hexadecimal format. Both waveforms were quintessential, as operations involving straightforward multiplication and division can be easily verified from their decimal values. On the other hand, special multiplication operations such as MULH, MULHSU, and MULHU can be verified by looking at the hexadecimal value if the operation has returned the upper 32 bits of the result.

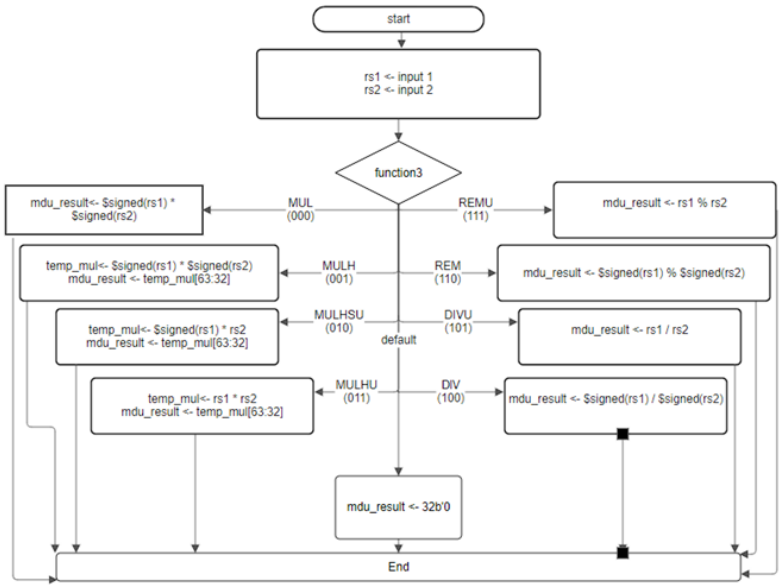

Fig. 2. The dataflow diagram of MDU module.

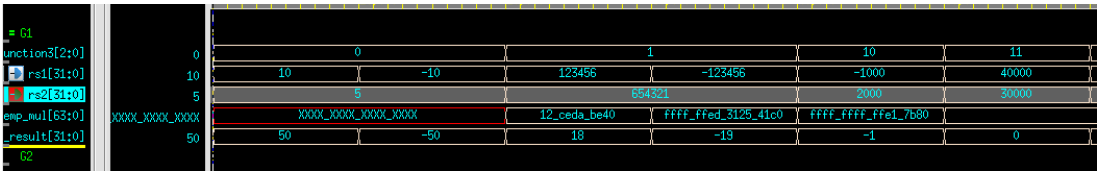

Fig. 3. The waveform simulation on VCS Verdi with "mdu_result" in decimal.

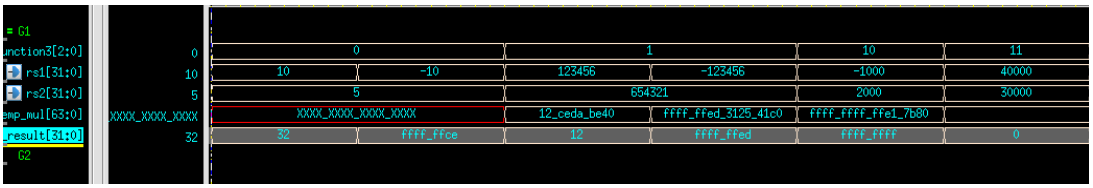

Fig. 4. The waveform simulation on VCS Verdi with "mdu_result" in hexadecimal.

III. "F" EXTENSION DESIGN

The floating-point unit will be designed based on the IEEE-754 standard as specified by the RISC ISA manual. A bottom-up approach was taken in order to design the FPU. With this, the basic floating-point arithmetic can be performed by the FPU module. The modules were written in a combinational style using the Verilog HDL.

A. FPU Top Level Module

Figure 5 below shows the block diagram of the top-level FPU module, which acts as a centralized unit that manages the FPU operations based on the "fpu_control" signals and the funct3. The FPU receives the input operands rs1 and rs2 along with the control signals via the "fpu_control", "funct3", and "fpu_sel". The output of the FPU module is produced at the "fpu_result" output port. The corresponding operation and the "fpu_control" were shown in Table IV. The "fpu_control" was obtained from bits 2:0 of the function5 of the RISC-V instruction format, and "fpu_sel" was obtained from bit 0 of the function5. The function of the FPU top module was obtained from the corresponding RISC-V instruction format's function3 bits, as well as to differentiate between the floating-point sign injection operation. The data flow diagram of the FPU top-level module is shown in Fig. 6 below. The floating-point unit in this project is comprised of three main submodules, which are the "AddSubFPU" module, "MulFPU" module, and "DivFPU" module.

Table IV. Operation and corresponding fpu controls.

fpu_control	Operation
000	FPU_ADDSUB
010	FPU_MUL
011	FPU_DIV
100	SIGN_INJECTION

B. FPU Addition and Subtraction

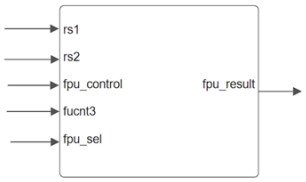

Fig. 5. Block diagram of FPU Top level module.

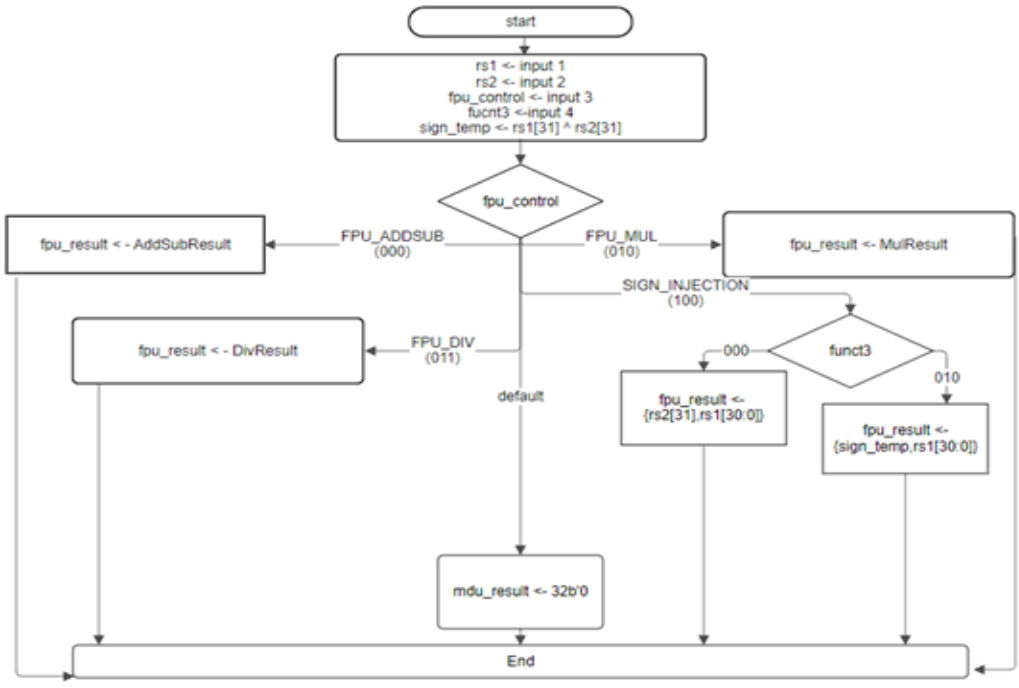

Fig. 6. Data flow diagram of FPU top level module.

Figure 7 below shows the dataflow diagram for the "AddSubFPU" module to perform multiplication between two floating-point inputs. After receiving the inputs, the sign, exponent, and mantissa with the MSB hidden bits were unpacked. Unlike addition and subtraction, multiplication does not require any swapping, as the order of the multiplication does not influence the result. Next, the signs of the floating-point numbers were XOR'd to obtain the sign of the result. The exponents were also added, and the compounding bias value of 127 in decimal was subtracted.

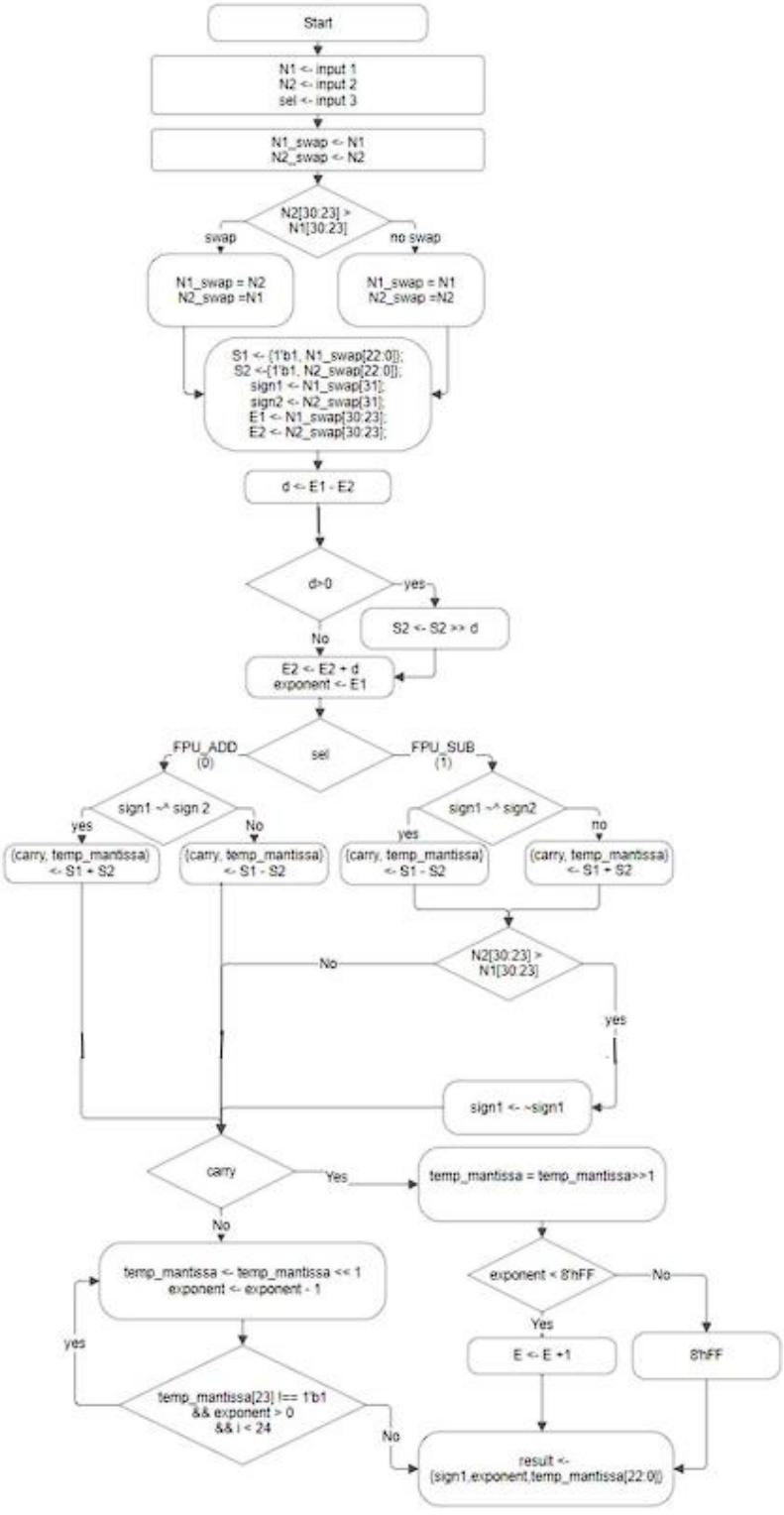

Fig. 7. The dataflow diagram of the AddSubFPU module.

C. FPU Multiplication

Figure 8 below shows the dataflow diagram for the "MulFPU" module to perform multiplication between two floating-point inputs. After receiving the inputs, the sign, exponent, and mantissa with the MSB hidden bits were unpacked. Unlike addition and subtraction, multiplication does not require any swapping as the order of the multiplication does not influence the result. Next, the signs of the floating-point numbers were XOR'd to obtain the sign of the result. The exponents were also added, and the compounding bias value of 127 in decimal was subtracted".

D. FPU Division

The Fig. 9 shows the "DivFPU" dataflow diagram, which shows how much it mirrors the multiplication operation. However, the "DivFPU" implements the floating-point division where, after unpacking the floating-point values and extending the mantissa's the division operation is performed. The quotient was shifted into "raw_mantissa" before division to ensure we can get a 24-bit long significand. This is because after the division operation, the value quotient becomes half its size and results in a loss of significand values which are then normalized similarly to the MulFPU. The normalization process repeats itself until the hidden bit MSB is 1. In the case of Division, the exponents were subtracted instead of added, and the bias value of 127 was added back into the exponent since subtraction cancels the bias from the exponents. 0FFH represented the overflow value in the exponent to represent infinity.

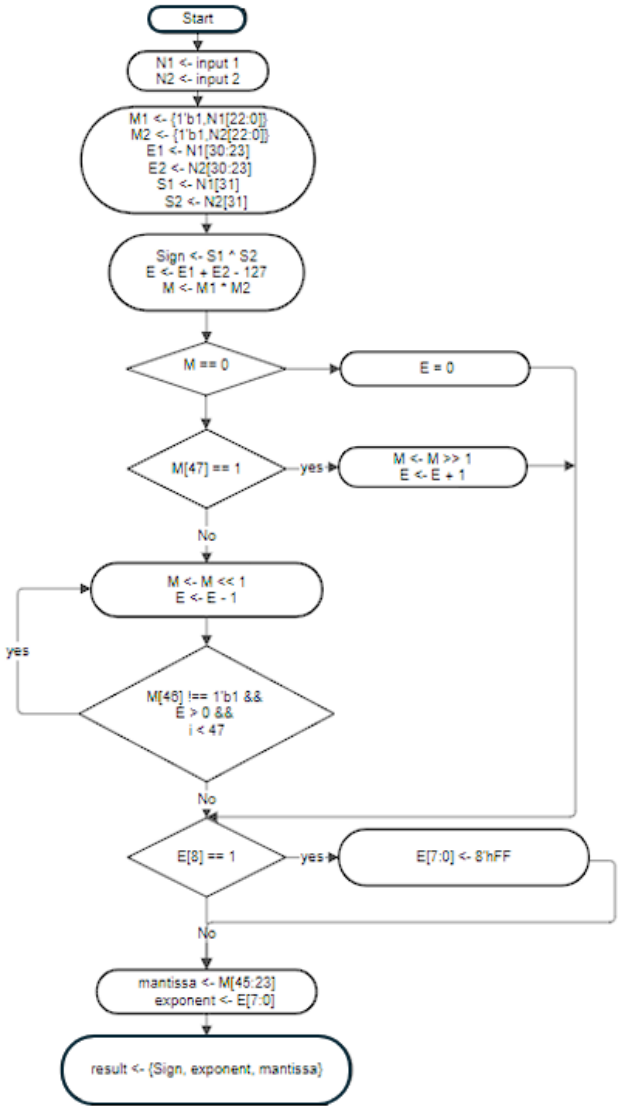

Fig. 8. The dataflow diagram of the MulFPU module.

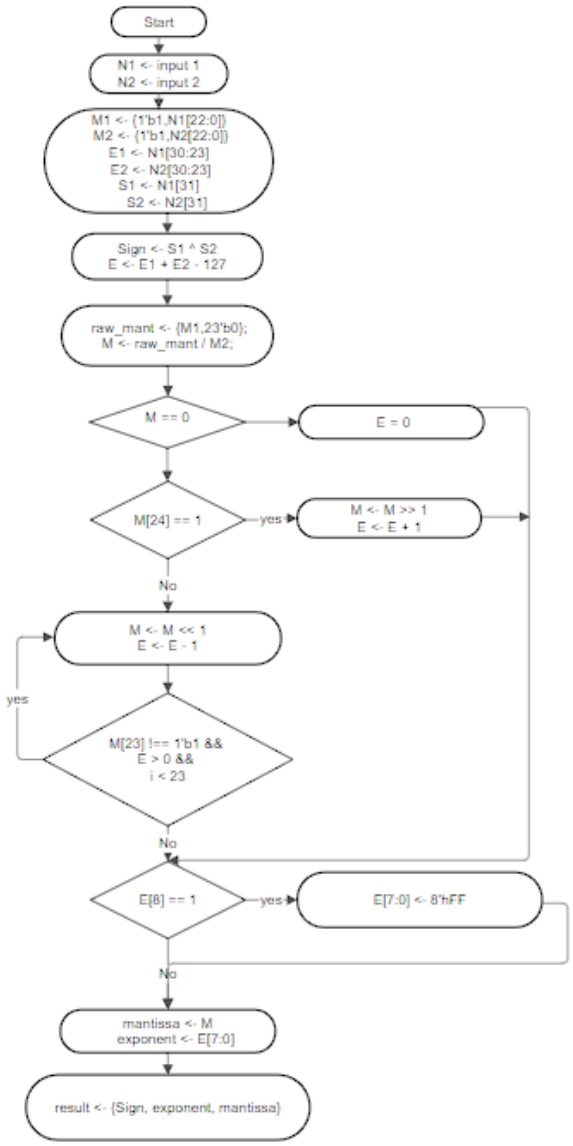

Fig. 9. The dataflow diagram of the DivFPU module.

E. Results for Arithmetic Operations

The arithmetic operations for the floating-point unit, such as addition, subtraction, multiplication, and division, were tested using similar test case inputs. Table V shows the test cases and their corresponding inputs for validation of the FPU designed. The expected values and actual values were compared, and the relative errors were calculated and tabulated. Fig. 10 shows the maximum relative error for each test case. The maximum relative error for 5th test case was the highest among all the test cases at about 0.25%.

Table V. Test cases and corresponding inputs.

Test case	**rsl (decimal)**	**rs2 (decimal)**
1	3.2	4.2
2	0.66	0.51
3	-0.5	-6.4
4	-0.5	6.4
5	2.8235295	-0.9411765

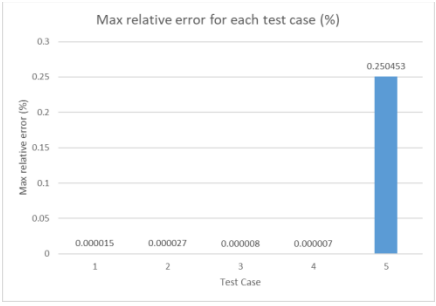

Fig. 10. Maximum relative error for each test case of the arithmetic operations.

F. FPU Sign Injection

The sign injection operations implemented were the FSGNJ.S and FSGJX operations in the FPU top-level module. The FSGNJ.S instruction involves injecting the sign of the second source operand and the magnitude of the first source operand into the destination register. The FSGNJX.S involves the injection of the XOR result between the signs of the source registers and the magnitude of the first source into the destination register. Figure 11 shows the waveform simulation for the FPU sign extension, while Table VI shows the tabulation of the results for the FPU Sign injection.

Table VI. The tabulation of results for the Sign Injection operation of the FPU module.

rs1 (base10)	rs2 (base10)	Expected result (base10)	Expected result (base16)	Actual result (base10)	Actual result (base16)
-0.5	6.4	0.500000	3f000000	0.500000	3f000000
-0.5	6.4	-0.500000	bf000000	-0.500000	bf000000

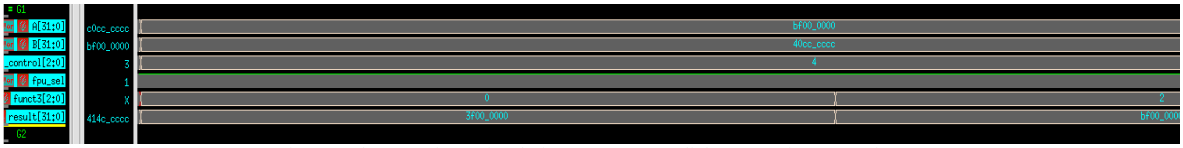

Fig. 11. The waveform simulation on VCS Verdi for FPU Sign extension.

IV. "MF" INTEGRATED RV32I

A. Base RV32I

Figure 12 shows the pipeline view of the base RV32I design acquired. The top module of the design is referred to as the "pipeline_rv32i". In the module, the processor can be divided into 5 pipeline stages, which allow for the processor to handle multiple instructions at a particular clock cycle. Thus, increasing the throughput as the next instruction can be started instead of just waiting for the previous instruction to finish its lifecycle. The five stages of the pipeline are modularized based on the stages, and their modules are called "fetch", "decode", "execute", "memory", and "writeback".

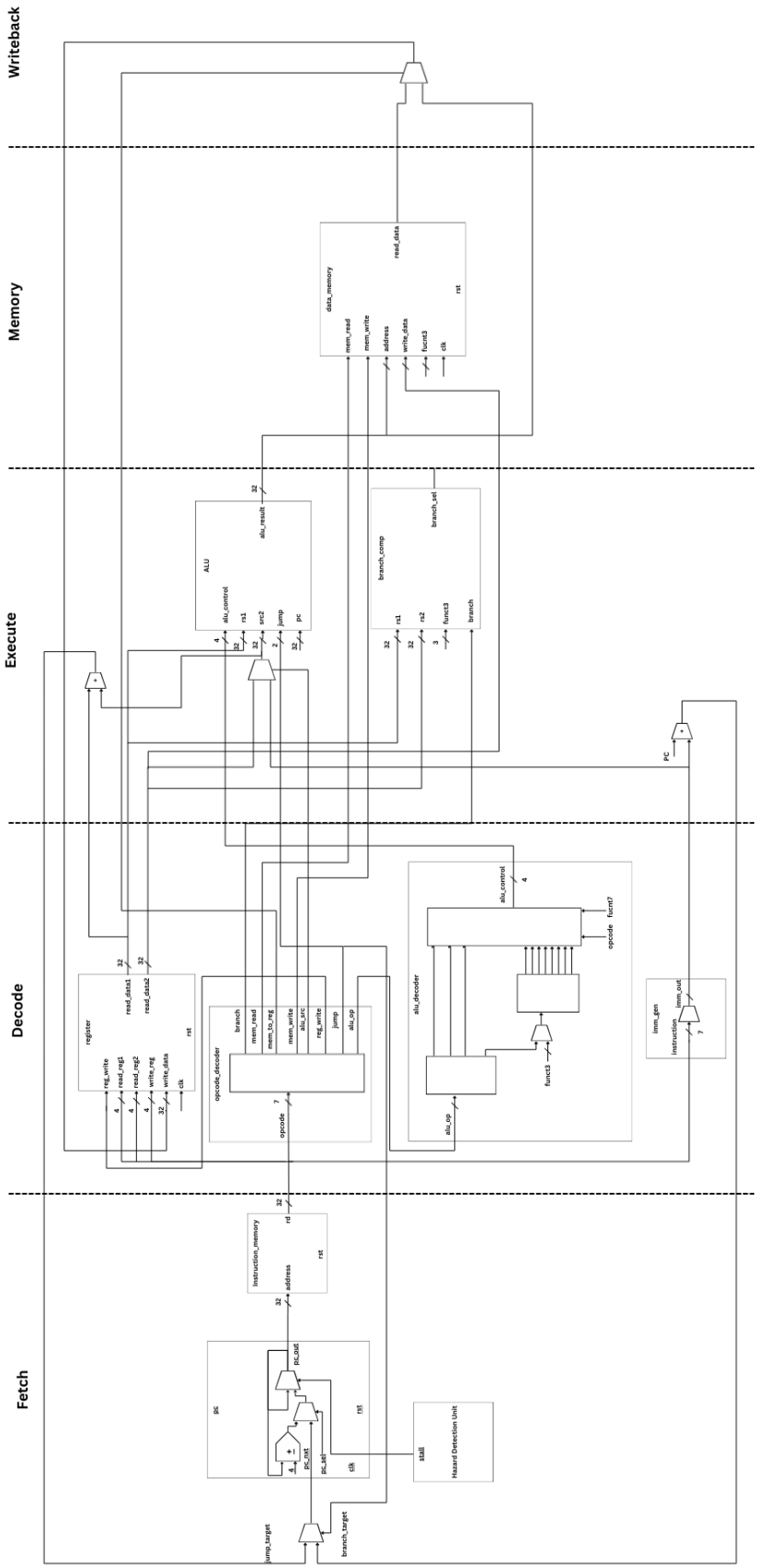

Fig. 12. Overview of pipeline_rv32i module.

B. "MF" Extension into RV32I

This section will discuss the integration of the "M" and "F" extensions into the base RV32I processor. Figure 13 shows the pipeline view of the RV32I integrated with the "MF" extensions, thus illustrating a complete RV32IMF processor. It is designed on top of the base RV32I processor of the "pipeline_rv32i" module with enhanced integer multiplication and single precision floating point capabilities. The changes made to the pipeline are highlighted in red, where the new modules "MDU" and "FPU" have been integrated into the "execute" module. At the "decode" stage, a set of 32-bit FPU registers was introduced to handle floating-point instructions. New inputs were also introduced at the "decode" stage, such as "FPR_GPR_sel," which selects between the general-purpose integer register and dedicated floating-point registers for writing data into them. The additional output control signals, such as "fpu_op" and "mul_en," aid in selecting between the different units available in the "Execute" stage. Additionally, "fpu_op" and "reg_sel" enable the selection between integer registers and floating-point registers for the operands of an operation, depending on the control logic.

C. Simple Projectile Motion Program

Figure 14 shows the assembly commands as they are contained in the "projectile.mem" file. Figure 15 is the memory data from the "projectile_data.mem" file. The operand (floating point) values are stored in IEEE-754 single-precision floating-point hexadecimal format. The waveform simulation for the simple projectile motion program was verified as shown in Fig. 16 and tabulated as shown in Table VII.

```
projectile.mem
1   00002087  //flw f1, 0(x0)
2   00402107  //flw f2, 4(x0)
3   00802187 //flw f3, 8(x0)
4   00c02207 //flw f4, 12(x0)
5
6   1020f2d3 //fmul.s f5, f1, f2
7   10217353 //fmul.s f6, f2, f2
8   1061f3d3 //fmul.s f7, f3, f6
9   10727453 //fmul.s f8, f4, f7
10
11  0082f4d3//fadd.s f9, f5, f8
12
```

Fig. 14. Assembly code snippet for calculating position of Projectile Motion in "projectile.mem".

```
projectile_data.mem
1   00
2   00
3   40
4   40 // initial velocity (v0)
5   00
6   00
7   00
8   40 // time (t)
9   cd
10  cc
11  1c
12  41 // acceleration cosntant (a)
13  00
14  00
15  00
16  3f // 0.5 coefficient
```

Fig. 15. The input values and constant in "projectile_data.mem".

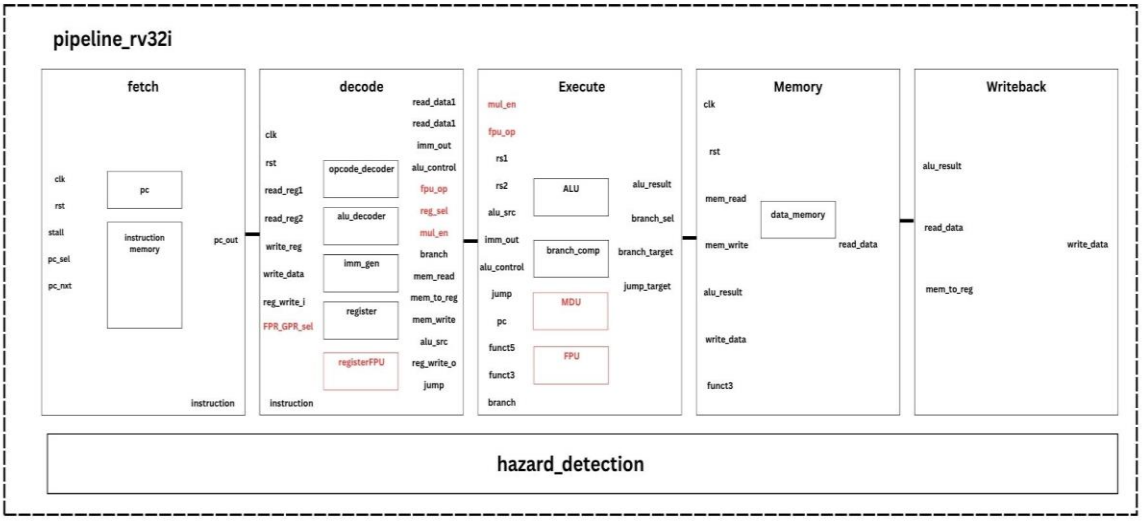

Fig. 13. The pipeline view of the RV32IMF.

Table VII. The tabulation of assembly instructions with the expected result and simulation results for the projectile motion program.

Instruction (assembly)	Instruction (Hexadecimal)	Expected result	Actual result
flw f1, 0(x0)	00002087	f1 = 40400000(3.0)	f1 = 40400000
flw f2, 4(x0)	00402107	f2 = 40000000(2.0)	f2 = 40000000
flw f3, 8(x0)	00802187	f3 = 411ccccd(9.8)	f3 = 411ccccd
flw f4, 12(x0)	00c02207	f4 = 3f000000(0.5)	f4 = 3f000000
fmul.s f5, f1, f2	1020f2d3	f5 = 40c00000(6.0)	f5 = 40c00000
fmul.s f6, f2, f2	10217353	f6 = 40800000(4.0)	f6 = 40800000
fmul.s f7, f3, f6	1061f3d3	f7 = 421ccccd(39.2)	f7 = 421ccccd
fmul.s f8, f4, f7	10727453	f8 = 419ccccd(19.6)	f8 = 419ccccd
fadd.s f9, f5, f8	0082f4d3	f9 = 41cccccd(25.6)	f9 = 41cccccd

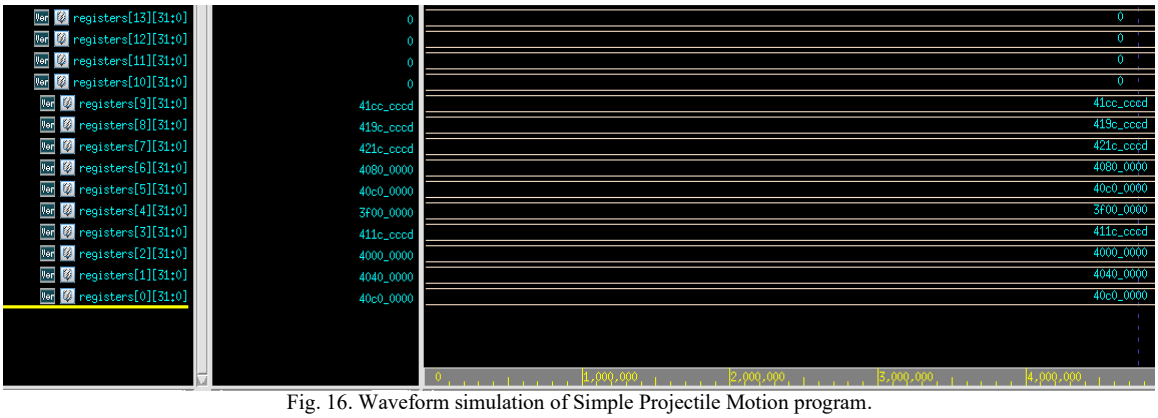

Fig. 16. Waveform simulation of Simple Projectile Motion program.

D. Synthesis Comparison between RV32I and RV32IMF

Firstly, after successfully integrating the extensions into the RV32I, we will attempt to compare the RV32I to RV32IMF at an operating frequency of 100 MHz. We will base this comparison on the three main ASIC design metrics, which are area, timing, and power. Table VIII below shows the performance comparison of the RV32I and RV32IMF based on area, timing, and power. Integrated "MF" showed good performance in contrast to the base RV32I. There was a small increase in area (9.4%), a decrease in timing closure (0.92%), and a minimal increase in power (0.4%)

Table VIII. Performance comparison between RV32I and RV32IMF.

Processor	Area (μm^2)	Timing(ns)	Power (μW)
RV32I	67774.176515	8.65	3.4805e+04
RV32IMF	74126.795963	8.57	3.4949e+04

E. Synthesis RV32IMF at 100MHz, 200MHz and 2GHz

Table IX shows the synthesis results at the target frequencies of 100MHz, 200MHz, and 2GHz for the RV32IMF core. At 100MHz, the RV32IMF has a modest performance. Despite a more constrained timing, RV32IMF was able to meet the slack at 200MHz. However, at a 2GHz operating frequency there were timing violations emerged from the long delays due to the long modules introduced.

Table IX. The tabulation of the synthesis results of frequency with respect to area, timing, and power.

Frequency (Hz)	Period (ns)	Area (μm^2)	Timing (ns)	Power (μW)
2G	0.5	74202.857062	-0.60	6.9177e+05
200M	5	74127.097069	3.57	6.9518e+04
100M	10	74126.795963	8.57	3.4949e+04

V. CONCLUSION

In conclusion, this project accomplished the design, integration, verification, and synthesis of the RISC-V "M" and "F" extensions into the base RV32I processor, producing the RV32IMF core. In terms of functional verification, the MDU was verified functionally to support signed and unsigned integer multiplication and division operations. The relative error in floating-point arithmetic consistently demonstrated an acceptable error margin, with the highest error margin well within the IEEE-754 single-precision tolerance, as the highest error margin was approximately 0.25%. Synthesis across several frequency targets also showed the RV32IMF processor was functional to 200 MHz, but above this, such as at 2GHz, timing violations emerged from the long arithmetic paths through the MDU and FPU. Yet, at the modest frequency of 100 MHz, the processor operated at a substantially lower power consumption with a large timing margin, making for a suitable component of a power-sensitive system.

ACKNOWLEDGEMENT

Appreciation is given to the Faculty of Artificial Intelligence and Engineering, Multimedia University, and the Center of Excellence for Microelectronics Design Collaboration, Universiti Sains Malaysia. The authors would also like to extend our deepest gratitude for the invaluable comments and insights from the reviewers.

FUNDING STATEMENT

The authors received no funding from any party for the research and publication of this article.

AUTHOR CONTRIBUTIONS

Tasigen Visvanathan: Conceptualization, Formal Analysis, Investigation, Methodology, Validation, Visualization, Testing, Implementation, Writing–Original Draft Preparation;
Chu Liang Lee: Conceptualization, Formal Analysis, Investigation, Methodology, Validation, Visualization, Supervision, Resources, Technical Guidance, Writing–Review & Editing;
Kah Yoong Chan: Project Administration, Resources, Supervision,
C Senthilpari: Project Administration, Supervision, Technical Guidance, Writing–Review & Editing.

CONFLICT OF INTERESTS

The authors declare no conflicts of interest regarding this manuscript.

ETHICS STATEMENTS

Our publication ethics follow The Committee of Publication Ethics (COPE) guideline. https://publicationethics.org/

REFERENCES

[1] A. Waterman and K. Asanović, "*The RISC-V Instruction Set Manual, Volume I: User-Level ISA,*" Version 2.1, Electrical Engineering and Computer Sciences Department, University of California, 2016. [Available Online] https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf.

[2] M. Maan and A. Bindal, "A Review on IEEE-754 Standard Floating Point Arithmetic Unit," *Int. J. Adv. Res., Ideas and Innov. in Technol.*, vol. 2, no. 3, pp. 1–4, 2016.

[3] S. Mach, F. Schuiki, F. Zaruba and L. Benini, "FPnew: An Open-source Multi-format Floating-point Unit Architecture for Energy-proportional Transprecision Computing," *arXiv*, 2007.01530, 2020.

[4] K. Asanović *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-17. [Available Online on 15 April 2016] http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html \.

[5] M. Kovač *et al.*, "FAUST: Design and Implementation of A Pipelined RISC-V Vector Floating-point Unit," *91 Microproc. and Microsyst.*, vol. 97, pp. 104762, 2023.

[6] S. Aggarwal *et al.*, "Shedding the Bits: Pushing the Boundaries of Quantization with Minifloats on FPGAs," *arXiv*, 2311.12359v3, 2024.

[7] T. Halvadia, "RISC-V-RV32I," *GitHub Repository*. [Available Online on 8 July 2025] https://github.com/Tirthhalvadia/RISC-V-RV32I/tree/main/code.

[8] Q. Madariaga, "Development of a RISC-V Microprocessor for ASIC Integration," *M.S. Thesis*, OMEGA Microelectronics, Ecole Polytechnique, France, 2022.

[9] W. Feng, H. Chen, G. Lin and X. He, "An Area and Power Efficient Design of Fused Integer–floating Point Unit for RISC-V Cores," in *Int.l Conf. on Electron. Informat. Eng. and Comput. Commun.*, Wuhan, China, pp. 219–223, 2023.

[10] R. da Silva, V. dos Santos, F. Petkowicz, R. Calcada and R. Reis, "Synthesis of Steel-ASIC, A RISC-V Core," *J. Integr. Circ. and Syst.*, vol. 17, no. 2, pp. 1–8, 2022.

[11] Y. N. Koay, "Front End Design (Logic Synthesis) of RISC-V Processor Using Design Compiler," *Master's Thesis*, Universiti Tunku Abdul Rahman, 2021.