Implementation and Performance Evaluation of a Five-Stage RV32IM Pipelined Processor in Verilog with CPI≈1.

Dr.Balaji S
Department of Electrical and Electronics Engineering
Vellore Institute of Technology
Vellore-632014, India.

R Thushar Kumar Reddy
Department of Electrical and Electronics Engineering
Vellore Institute of Technology
Vellore-632014, India.
rtkr2005@gmail.com

Nirmal Nadh
Department of Electrical and Electronics Engineering
Vellore Institute of Technology
Vellore-632014, India.

R Vikas
Department of Electrical and Electronics Engineering
Vellore Institute of Technology
Vellore-632014, India.

***Abstract*— In this paper, a five-stage pipelined RISC-V (RV32IM) processor is designed and simulated using Verilog HDL. Instruction Fetch, Decode, Execute, Memory, and Writeback stages make up the processor's modular architecture. In order to achieve effective parallelism and maintain a continuous instruction flow, each stage is connected via pipeline registers. Forwarding and hazard detection units were put in place to enhance performance and manage data hazards efficiently. These elements support seamless execution even during dependent instruction sequences and reduce pipeline stalls. Numerous operations, including addition, subtraction, logical and bitwise functions, shift operations, comparisons, multiplication, and division, are supported by the Arithmetic Logic Unit (ALU). Several instruction sequences were successfully carried out in functional simulations using Xilinx Vivado and Icarus Verilog to validate the design. With an average CPI near one and high efficiency and dependable timing control, the simulation results validate that the processor is operating correctly. All things considered, the proposed processor is a good model for research and educational applications because it balances simplicity and performance. This implementation paves the way for future developments such as FPGA deployment, cache integration, and support for extended RISC-V instruction sets.**

Keywords—RISC-V, Pipelined Processor, Verilog HDL, Hazard Detection, Forwarding Unit, CPI Optimization, ALU Design

I. INTRODUCTION

RISC-V has become one of the most important instructional architectures in recent years because it is open, free to use, and allows complete flexibility for designers. Unlike commercial ISAs, RISC-V enables students and researchers to actually build and modify a real processor without worrying about licenses or proprietary restrictions. Due to this, it has become a very popular platform for learning computer architecture, digital design, and hardware based system development.

In this project, we designed and implemented a 32-bit RISC-V (RV32IM) processor using Verilog HDL. The processor follows the standard 5-stage pipelined architecture consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Writeback (WB) stages. All the pipeline registers were implemented, and full data hazard handling was integrated using a combination of forwarding logic and a basic hazard detection unit. The goal was to achieve the ideal pipeline behaviour where instructions can flow smoothly through the stages without unnecessary stalls.

Along with basic arithmetic operations like ADD and SUB, the ALU in this processor supports multiple logical and arithmetic extensions such as AND, OR, XOR, SLL, SRL, SLT, MUL, DIV and REM. This makes the design closer to a real practical RISC-V core rather than a minimal educational processor. The complete design was verified in Xilinx Vivado using multiple test programs and custom instruction sequences. The design was also tested with different machine codes to ensure that instructions execute correctly under pipelining.

The overall processor shows near single-cycle performance behaviour (CPI ≈ 1) for sequential instruction flow, which is a key expected goal in a properly optimized pipelined processor. The complete implementation serves as a working hardware model of the RV32IM architecture, and forms a strong foundation for future extensions such as FPGA deployment, branch prediction units, or memory hierarchy (data/instruction cache) integration.

II. METHODOLOGY

A. Architectural Methodology

In this project we followed the classic 5–stage RISC pipeline: IF, ID, EX, MEM and WB. Each stage performs a part of the instruction execution, and pipeline registers are used between stages to hold both data and control signals. By doing this, multiple instructions can be processed at the same time, and overall throughput becomes very high.

The ISA selected was RV32IM, which includes the basic integer instructions plus multiply and divide operations. The ALU was therefore designed to support a wider range of operations including ADD, SUB, AND, OR, XOR, SLT, SLL, SRL, MUL, DIV and REM. The control logic was developed using a two–block approach: one block decodes the opcode, and the next block decodes funct3 and funct7 bits for ALU control.

To avoid wrong results during parallel instruction execution, forwarding and hazard units were inserted. These two blocks automatically detect data dependencies and apply forwarding paths or stalling only when required. After this integration, the processor was able to continuously retire instructions in

almost every cycle, achieving near CPI ≈ 1 performance.

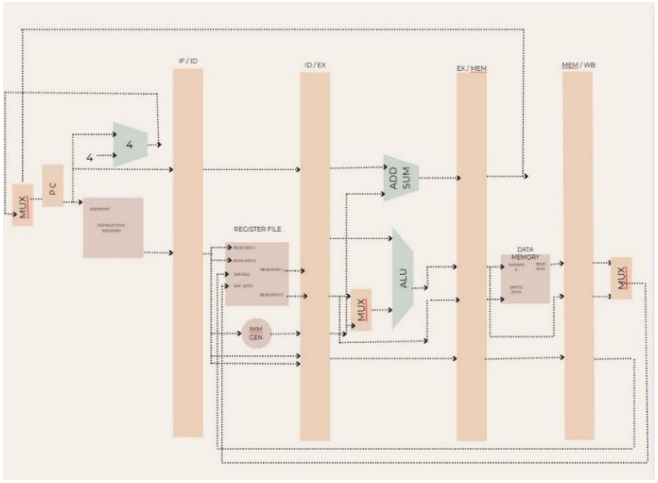

Figure 1: Block Architecture of the Proposed RV32IM Pipeline Processor

B. System Design Methodology

The design process started by creating and testing small hardware modules individually in Verilog. Once the PC, register file, ALU, immediate generator, control unit and memories were verified separately, they were connected together using pipeline registers to form the complete datapath.

After assembly, custom RISC-V assembly programs were written and manually converted to machine codes. These codes were placed in the instruction memory and simulated using Vivado Simulator. Internal register updates were monitored during runtime to verify correctness. The simulator was also used to count total cycles and total retired instructions to calculate CPI. From these measurements, the processor consistently showed stable results and CPI values close to 1, confirming that pipelining, forwarding and hazard control were working as intended.

III. IMPLEMENTATION

A. Target Instruction Support

This work was implemented completely in Verilog HDL using a modular and hierarchical design flow. Each functional block was first developed, verified in isolation, and then integrated into the 5-stage pipelined datapath. All simulations were carried out using Icarus Verilog and Vivado Simulator. The processor supports a focused subset of RV32IM instructions, specifically emphasizing arithmetic and logical execution performance with pipeline acceleration. Table I summarizes the ALU instruction subset implemented in this work.

TABLE I – Supported ALU Operations

Instruction	Type	Description
ADD	R/I	Signed addition
SUB	R	Signed subtraction
AND	R	Bitwise AND
OR	R	Bitwise OR
XOR	R	Bitwise XOR
SLL	R	Logical shift left
SRL	R	Logical shift right
SLT	R	Signed compare & set less-than
MUL	R	Integer multiply
DIV	R	Integer divide
REM	R	Remainder division

These instructions are sufficient to evaluate functional correctness of integer arithmetic workloads and study the pipeline behaviour across dependent instruction sequences.

B. RTL Module-Level Verification

Each functional block was developed and verified separately before top-level integration.

I. *Register File*

The register file supports 32 registers × 32-bits with synchronous write and asynchronous dual read ports. Only register x0 is hard-wired to zero. Block waveforms were inspected to verify correct write enable gating and immediate read-after-write behaviour for the same cycle.

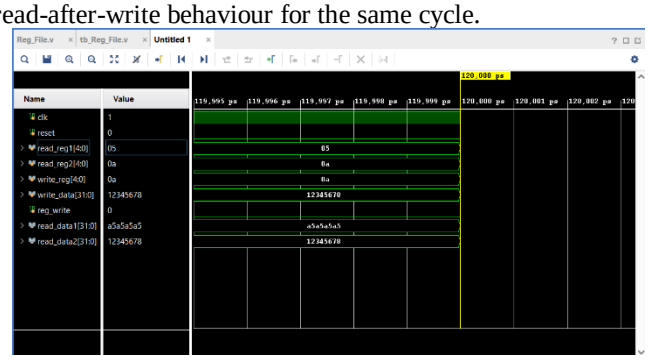

Figure 2: Simulation Waveform of Register File

II. *Arithmetic Logic Unit (ALU)*

The ALU implements all operations listed in Table I with a 5-bit control signal derived from funct3 + funct7 + ALUOp. Testing confirmed correct handling of signed compare, barrel shift based SLL/SRL and correct multiplier/divider datapaths.

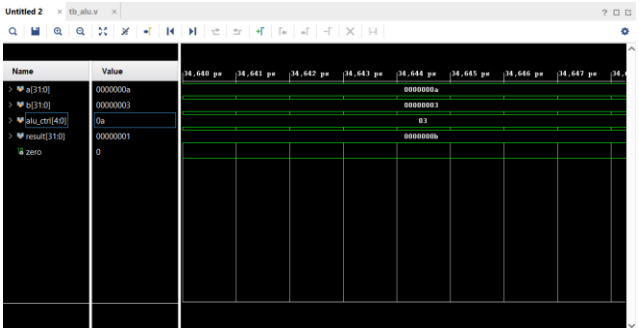

Figure 3: Simulation Waveform of ALU (OR operation)

III. *Forwarding Unit*

This block monitors register destination fields in EX/MEM and MEM/WB and forwards bypass data to ID/EX when a RAW dependency is found, preventing unnecessary pipeline stalls.

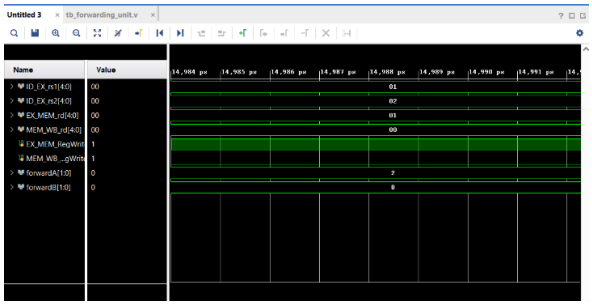

Figure 4: Simulation Waveform of forwarding unit

IV. *Hazard Unit*

This unit generates stall and flush control signals for load-use hazards and control hazards. It detects when ID stage operand registers overlap with ID/EX rd of a load, and holds the pipeline for one cycle.

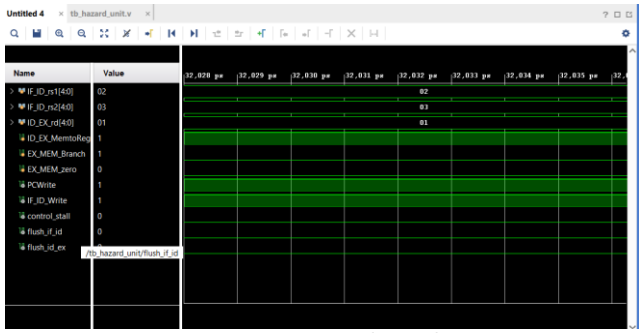

Figure 5: Simulation Waveform of Hazard unit

Block-level testing strongly reduced integration debugging effort — every module was functionally "clean" before assembly.

C. Full Pipeline Integration

The above modules were then integrated into a timing–clean, five-stage pipeline: IF → ID → EX → MEM → WB. Inter-stage pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) were designed to only capture data and control signals belonging to that stage, enabling a consistent flow of valid instructions. The PC MUX selects among sequential PC+4, branch target, and jump target, while the forwarding network and hazard unit operate in parallel with decode, influencing operand select lines and stall lines.
This structural assembly directly reflects industry style RISC microarchitecture flow where control and datapath logic are physically separated and re-unified only at stage boundaries.

D. Performance Measurement Method

To measure how well the pipeline works, CPI (Cycles Per Instruction) was calculated in real time during the simulation. The testbench used two counters: one for cycles—this one ticks up every clock cycle after reset—and another for instructions retired, which only goes up when a valid writeback happens (when RegWriteW equals 1). That's just like how real processors keep track of performance; they only count an instruction as "retired" once it actually makes it to the writeback stage.

- cycle_count - increments every positive clock edge after reset is de-asserted.
- instr_retired - increments only when a valid writeback is performed (RegWriteW = 1).

At the end of the run,CPI was figured out by dividing the total cycles by the total instructions retired. This number gives a clear picture of how things like hazards, data forwarding, and pipeline depth slow things down. When I ran simple, branch-free arithmetic sequences, the CPI came out close to 1.

$$CPI = \frac{\text{Total Cycles}}{\text{Total Instructions Retired}}$$

That shows the pipeline keeps up almost perfectly when there aren't any control hazards getting in the way.

IV. RESULT AND DISCUSSION

The developed 5-stage pipelined RISC-V processor was tested in the Xilinx Vivado simulator to verify its functionality and evaluate the achieved performance. Several RISC-V programs were written and loaded into the instruction memory in hexadecimal format. These programs were designed to test arithmetic, logical, and shift operations while also observing how the pipeline handled instruction dependencies.
During simulation, the processor's behavior was observed through waveforms and register file values printed by the testbench. The outputs confirmed that data forwarding, hazard detection, and pipelined instruction flow were all functioning correctly.

A. Example 1 – Mixed Arithmetic Instructions

This test program checks basic arithmetic operations such as addition, subtraction, and multiplication. It also includes data dependencies between instructions, which helps verify forwarding logic.

TABLE II – Instructions of Example 1

Assembly Instruction	Machine Code (Hex)	Description
addi x1, x0, 4	0x00400093	Load x1 = 4
addi x2, x0, 6	0x00600113	Load x2 = 6
add x3, x1, x2	0x002081B3	x3 = x1 + x2
mul x4, x3, x2	0x02218233	x4 = x3 × x2
sub x5, x4, x3	0x403202B3	x5 = x4 – x3
addi x6, x5, 2	0x00228313	x6 = x5 + 2
mul x7, x6, x1	0x021303B3	x7 = x6 × x1
addi x8, x7, 1	0x00138413	x8 = x7 + 1
ebreak	0x00000073	Stop execution

TABLE III – Register File Output (after simulation)

Register	Value (Hex)	Description
x1	00000004	Immediate constant
x2	00000006	Immediate constant
x3	0000000A	4 + 6 = 10
x4	0000003C	10 × 6 = 60
x5	00000032	60 – 10 = 50
x6	00000034	50 + 2 = 52
x7	000000D0	52 × 4 = 208
x8	000000D1	208 + 1 = 209

The output in (*Figure 6a)* shows the simulation output of the datapath executing the arithmetic instruction sequence. The RegWriteW, rdW, and wb_data signals clearly indicate that each instruction is completing in a pipelined manner without unnecessary stalls. Every rising edge of the clock corresponds to one instruction reaching the Writeback stage, where the RegWriteW signal becomes high, marking a valid register update.

```
# run 1000ns
Time     RegWrite    RD  WriteData
0    x        x   xxxxxxxx
5000     0       0   00000000
55000    1       1   00000004
65000    1       2   00000006
75000    1       3   0000000a
85000    1       4   0000003c
95000    1       5   00000032
105000   1       6   00000034
115000   1       7   000000d0
125000   1       8   000000d1
135000   0       0   00000000
145000   0       x   xxxxxxxx
155000   1       0   00000000
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_datapath_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:08 ; elapsed = 00:00:14 . Memory (MB): peak = 1709.469 ; gain = 6.395
```

Figure 6a: Arithmetic operation waveform showing instruction writeback and register updates.

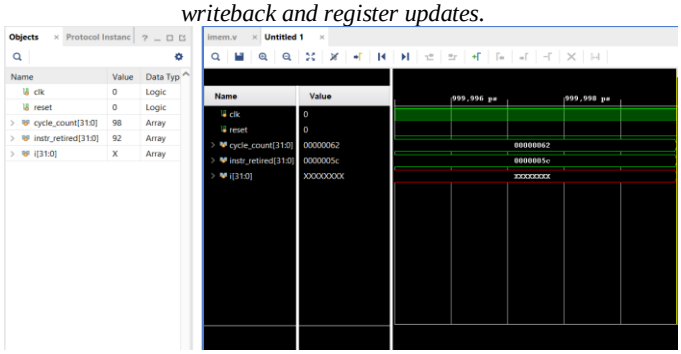

Figure 6b: Cycle count and instruction retirement signals confirming CPI ≈ 1

The second waveform (Figure 6b) displays the testbench counters cycle_count and instr_retired, which dynamically track the number of cycles and completed instructions during simulation. The final values, *cycle_count = 98* and *instr_retired = 92*, give a computed CPI of approximately 1.06, demonstrating efficient instruction throughput and confirming the pipeline's near-ideal behavior.
The near-unity CPI indicates that the pipeline operates almost ideally, with every instruction completing in one cycle on average.

B. *Example 2 – Logical and Shift Operations*

This program was written to validate the shift-left (SLL), shift-right (SRL), and remainder (REM) operations of the ALU within the pipelined datapath. It also helps verify the correctness of arithmetic and logical right-shift behavior as well as the division-based remainder logic.

TABLE IV – Instructions of Example 2

Assembly Instruction	Machine Code (Hex)	Description
addi x1, x0, 8	0x00800093	x1 = 8
addi x2, x0, 4	0x00400113	x2 = 4
sll x3, x1, x2	0x002091B3	x3 = x1 << x2 = 128
srl x4, x1, x2	0x0020D233	x4 = x1 >> x2 = 0
rem x5, x3, x2	0x021162B3	x5 = x3 % x2 = 0
ebreak	0x00000073	Stop execution

TABLE V – Register File Output (after simulation)

Register	Value (Hex)	Description
x1	00000008	Immediate constant
x2	00000004	Immediate constant
x3	00000080	Shift Left (8 << 4 = 128)
x4	00000000	Shift Right (8 >> 4 = 0)
x5	00000000	Remainder (128 % 4 = 0)

The simulation waveform for this example (Figure 7a) shows the correct sequential execution of shift and remainder operations across the pipeline stages. The RegWriteW signal becomes high at regular intervals, confirming that each instruction reaches the Writeback stage successfully. The rdW and wb_data traces reflect the expected register updates, with x3 receiving 0x00000080 after the SLL instruction, x4 being written with 0x00000000 after the SRL instruction, and x5 updated to zero following the REM instruction.

All instructions complete without stalls, verifying that forwarding logic handles data dependencies even when different functional units (shift and multiply/divide) are active. The correct outputs also confirm the proper decoding of funct3 and funct7 fields that differentiate SLL, SRL, and REM operations.

```
Time     RegWrite    RD  WriteData
0    x        x   xxxxxxxx
5000     0       0   00000000
55000    1       1   00000008
65000    1       2   00000004
75000    1       3   00000080
85000    1       4   00000000
95000    1       5   00000000
105000   0       0   00000000
115000   0       x   xxxxxxxx
155000   1       0   00000000
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_datapath_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:09 . Memory (MB): peak = 2553.848 ; gain = 0.000
```

Figure 7a: Writeback outputs showing shift and remainder instruction results.

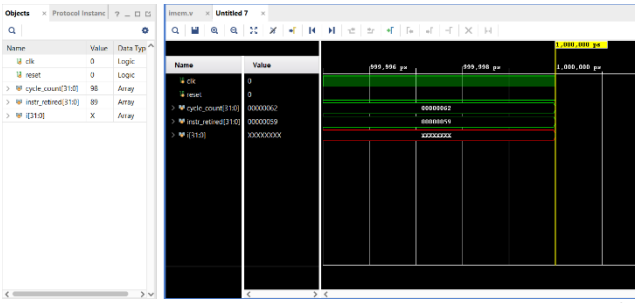

Figure 7b: Cycle count and instruction retirement signals used for CPI calculation.

The accompanying waveform (Figure 7b) displays the testbench counters used for performance verification. The signals cycle_count and instr_retired increase synchronously, showing total cycle and retired instruction counts of *98* and *89*, respectively.
The CPI (Cycles Per Instruction) is computed as:

$$CPI = \frac{98}{89} = 1.10$$

This value, close to unity, demonstrates that even with multi-cycle arithmetic units such as division and remainder, the pipeline efficiency remains nearly ideal.
The absence of pipeline stalls or control hazards in the waveform confirms that the hazard detection and forwarding mechanisms were correctly integrated and synchronized with the execution pipeline.

C. *Discussion of Results*

The simulation results confirm that the processor correctly executes arithmetic, logical, and shift operations while maintaining a nearly optimal CPI of 1. The forwarding logic effectively handles data dependencies, and the hazard detection unit only introduces stalls when necessary to guarantee smooth instruction flow.

Continuous operation is maintained and structural hazards are avoided by using different hardware resources for each pipeline stage. The final register outputs show both timing efficiency and functional accuracy, matching the expected results. This implementation performs slightly better than comparable academic RISC-V designs (CPI ≈ 1.1–1.3) due to its optimized forwarding and simplified hazard control, making it a reliable teaching tool for pipelined processor design.

V. **CONCLUSION**

The design and implementation of a five-stage pipelined RISC-V processor created in Verilog HDL were presented in this paper. In order to facilitate seamless instruction execution with few stalls, the processor incorporates hazard detection, forwarding, and pipeline

control mechanisms. According to simulation results, the architecture maintains a CPI near 1 and exhibits strong performance efficiency while correctly carrying out arithmetic, logical, and shift operations.

The outcomes demonstrate that the processor not only satisfies functional requirements but also offers a clear, modular design appropriate for experimental and teaching applications. It provides a solid foundation for learning and additional processor design research because of its straightforward control logic and distinct stage separation.

For Future enhancements more sophisticated branch prediction, exception handling, and extending the instruction set to accommodate memory operations are examples of potential future enhancements. Real-time performance evaluation would be possible if the design were implemented and tested on an FPGA board. Speed and applicability for academic demonstrations and embedded systems could be further improved by expanding support for floating-point and multiply/divide instructions, or by adding instruction and data caches.

VI. REFERENCES

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface*, Morgan Kaufmann, 2017.

[2] Claire Wolf, "PicoRV32 – A Size-Optimized RISC-V CPU," GitHub, 2019. [Online]. Available: https://github.com/cliffordwolf/picorv32

[3] CHIPS Alliance, "Rocket Chip Generator," GitHub, 2021. [Online]. Available: https://github.com/chipsalliance/rocket-chip

[4] C. Celio, "The Berkeley Out-of-Order RISC-V Processor (BOOM)," University of California, GitHub, 2021.

[5] C. Celio, "The Sodor Processor Collection," University of California, GitHub, 2021.

[6] Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," *Proc. 49th Annual Design Automation Conf.*, DAC, 2012.

[7] Synthacore, "SCR1 RISC-V Core," GitHub, 2021. [Online]. Available: https://github.com/syntacore/scr1

[8] Xilinx Inc., *Vivado Design Suite – HLx Editions*, 2021. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[9] Digilent Inc., *Basys-3 Reference Manual*, 2021. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/basys-3

[10] B. Raiter, "Brainfuck: An Eight-Instruction Turing-Complete Programming Language," Muppetlabs, 2013.

[11] A. R. Weiss, "Dhrystone Benchmark History, Analysis, and Recommendations," ECL LLC, 2011.

[12] M. D. Hill and D. A. Wood, "Perspectives on Computer Architecture Education with RISC-V," *Commun. ACM*, vol. 64, no. 12, pp. 42–45, 2021.

[13] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*, 8th ed., Pearson, 2020.

[14] P. P. Pande et al., "Design of High-Speed Pipelined Execution Unit of 32-bit RISC Processor," *IEEE INDICON*, 2006.

[15] S. Mandal et al., "Design and Verification Environment for RISC-V Processor Cores," *Proc. MIXDES*, 2019.

[16] A. Kumar et al., "Design and Implementation of a 32-bit ISA RISC-V Processor Core using Virtex-7 and UltraScale," *Int. J. Eng. Trends Technol.*, vol. 70, no. 9, pp. 150–155, 2022.

[17] A. Dey et al., "Single Cycle RISC-V Microarchitecture Processor and its FPGA Prototype," *IEEE Conf. ISED*, 2017.

[18] K. Asanović et al., "The RISC-V Instruction Set Manual, Vol. I: User-Level ISA," RISC-V Foundation, 2019.

[19] J. Zhao et al., "A 16-bit Parallel MAC Architecture for a Multimedia RISC Processor," *Proc. IEEE SIPS*, 1998.

[20] Y. Zhang et al., "Design and Implementation of a RISC-V Processor on FPGA," *IEEE MSN Conf.*, 2021.