SHENKAR —— שנקר

ENGINEERING. DESIGN. ART. הנדסה. עיצוב. אמנות.

RISCV-MatX

RISC-V Acceleration for Matrix Multiplication

Student Names: Adan Masri

Ahmad Waked

Orjuwan Abd Alghany

Academic Supervisor: Oren Gannon Oren Ganon

Acknowledgments

We would like to express our sincere gratitude to our academic advisor, Dr. Gannon Oren, whose invaluable guidance, expertise, and encouragement were instrumental throughout this project. From the earliest stages of conceptualizing a lightweight, vector-extended RISC-V processor to its final FPGA implementation, Dr. Oren Gannon's insights and support continuously inspired us to push boundaries and strive for excellence.

We are also deeply thankful to Dr. Gabriela Dorfman Furman for her unwavering support and leadership, fostering an environment that enabled us to transform challenges into opportunities for growth. Our heartfelt thanks extend to Mr. Liviu Gal, whose practical advice and innovative ideas contributed greatly to shaping the final outcome of this work.

Finally, we are profoundly grateful to our families and friends for their patience, motivation, and constant encouragement throughout this journey. Their belief in us has been the cornerstone of our perseverance and success.

Index

1. Introduction

1.1. Project Overview

Project Title: RISCV-MatX: Fast matrix multiplication

Problem Statement: Normal matrix multiplication method is heavy and slow calculation that takes a long time, especially for large matrices.

Significance & Impact: This project contributes to electrical engineering by demonstrating how vector processing can significantly improve the efficiency of matrix operations in hardware. It addresses industry needs for faster and more energy-efficient solutions in AI and embedded systems.

Innovation and Novelty: This project adds custom vector instructions and hardware (VRF, VALU) to a basic RISC-V processor. Unlike standard vector extensions, the design is lightweight and tailored for matrix multiplication. It demonstrates a novel, simplified approach to vector processing in minimal hardware systems.

Target Audience: This research project targets mainly computer architecture students, researchers.

Project Goals: The main goal is to enhance a simple RISC-V processor with custom vector instructions. We aim to accelerate matrix multiplication using dedicated hardware modules (VRF and VALU).

Methodology: The processor was implemented in Verilog and tested using ModelSim to simulate and verify functionality correctness. Hardware prototyping was performed using Quartus II for FPGA prototyping. Bug-solving was done through waveform analysis, and performance data was collected and analyzed to compare scalar and vector execution.

1.2. Problem Definition

Matrix multiplication is a core operation in many computational fields, especially machine learning (ML) and artificial intelligence (AI). However, general-purpose scalar processors are inefficient at handling large-scale matrix operations due to their sequential nature. This leads to longer execution times and higher energy consumption.

1.2.1 Target Audience

This research project targets computer architecture students, academic researchers, and educators. It offers a practical model for studying custom vector extensions in RISC-V processors. The design supports learning and experimentation with parallel processing in hardware systems.

1.2.2 Existing Solutions

Existing solutions such as Intel's AVX, ARM's NEON, and the RISC-V Vector Extension (RVV) provide powerful vector processing capabilities for high-performance computing. However, these solutions are often complex and optimized for industrial-scale processors, making them less suitable for academic research on minimal architectures.

1.2.3 Our Solution

The proposed solution, RISCV-MatX, focuses on accelerating dense matrix operations by extending a basic RISC-V processor with custom vector capabilities. This includes introducing new ISA extensions—V_LOAD, V_STORE, and V_MUL—that allow multiple data elements to be processed per cycle. To support these instructions, the architecture is enhanced with dedicated hardware components such as a Vector Register File (VRF), a Vector ALU (VALU), and extended control logic. Additionally, the data memory width was expanded to **128 bits** to enable efficient vector data access and alignment. The design remains lightweight and RISC-V compliant, providing an efficient and accessible platform for research into hardware-level acceleration of parallel computations.

1.3. Objectives

Board Objective: To design and implement a lightweight, vector-extended RISC-V processor that accelerates matrix multiplication and supports research in hardware-level parallelism.

Specific Objectives

1. **Problem Addressed:**
   Scalar processors execute matrix operations sequentially, resulting in inefficient performance for tasks like machine learning. This project addresses the lack of parallelism in basic RISC-V cores.

2. **Targeted Improvements:**
   Introduce custom vector instructions ISA; (V_LOAD, V_STORE, V_MUL) and enhance the processor with vector-specific hardware modules (VRF and VALU), including expanding the data memory to 128 bits.

3. **Expected Outcomes:**
   A functioning Verilog-based RISC-V processor with vector support, validated through simulation and FPGA prototyping. Assembly test programs for matrix multiplication will demonstrate correctness and speed-up.

4. **Performance Metrics:**
   Compare scalar vs. vector implementations in terms of:
   - Instruction count reduction
   - Execution time
   - Memory access efficiency

- Hardware resource usage (area/cost, if available)

1.4. Scope and Limitations

Project Scope: The project focuses on designing and implementing a custom RISC-V processor with vector instruction support, specifically targeting efficient matrix multiplication. The main components covered include:

- Implementation of three custom vector instructions: V_LOAD, V_STORE, and V_MUL
- Development of dedicated hardware modules: Vector Register File (VRF) and Vector ALU (VALU)
- Expansion of data memory width to 128 bits for efficient vector access
- Integration and testing using Verilog, ModelSim, and Quartus II on an FPGA platform
- Writing and executing assembly programs to verify functionality and performance

Exclusions

- The project does not implement the full RISC-V Vector Extension (RVV) standard
- Floating-point operations are not supported
- Cache memory and pipeline hazard handling are not part of the current scope
- No integration with external peripherals or operating systems is included

Technical Limitations

- Matrix values are fixed.
- Matrix size is fixed to 4×4.
- The processor is not pipelined.
- Synthesis was performed only for the vector extension, not for the entire processor.

2. Functional Requirements

2.1. List of Requirements

1. Support for 32-bit RISC-V base instructions (RV32I).
2. Custom vector instruction support: V_LOAD, V_STORE, V_MUL.
3. Execution of matrix multiplication using vector operations.

4. Vector Register File (VRF) to store 128-bit wide vector data.
5. Vector ALU (VALU) capable of parallel element-wise multiplication.
6. Data memory width of 128 bits to match vector operations.
7. Ability to execute 4×4 fixed-size matrix operations.
8. Simulation support in ModelSim for verification.
9. Prototyping on FPGA using Cyclone II.
10. Waveform debugging to verify timing and logic correctness.

Motivation for each requirement

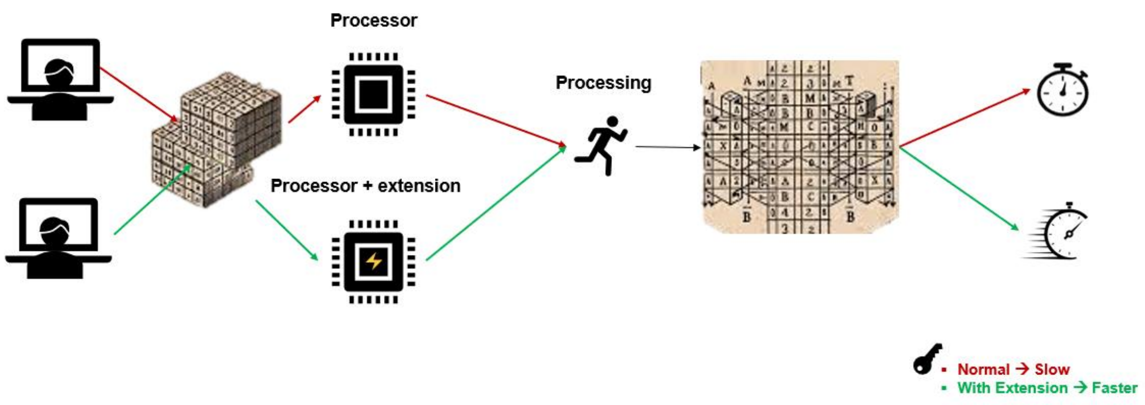
1. Base instruction support (RV32I): Required for compatibility with standard RISC-V software tools and basic control logic.
2. Vector instruction support: Needed to introduce parallelism and accelerate matrix operations.
3. Matrix multiplication functionality: The core use-case demonstrating performance improvement.
4. Vector Register File (VRF): Stores vectors efficiently to reduce memory access and support parallel execution.
5. VALU design: Executes multiple operations per cycle, critical for performance gain.
6. 128-bit memory width: Ensures vector loads/stores are efficient and aligned.
7. Fixed 4×4 matrix size: Simplifies testing, verification, and comparison.
8. Simulation in ModelSim: Enables step-by-step functional validation during development.
9. FPGA prototyping: Validates the design in real hardware and allows performance observation.
10. Waveform debugging: Essential for finding logic bugs and verifying timing without a full pipelined design.

2.2. Use case scenario

Baseline CPU vs. RISC-V + Extension

- Offloads matrix tiles from the CPU to the custom RISC-V extension (VALU + VRF).
- Executes vectorized dot-products (e.g., V_MUL) on 128-bit lanes.
- Returns completed tiles to memory with lower latency and energy.

Scenario Description:
A user submits matrices **A (4×4)** and **B (4×4)**. The baseline CPU computes **C = A x B (4×4)** with scalar loops and takes longer (red path in the diagram). With the extension, the CPU configures the accelerator, tiles A and B, the VALU loads vectors, performs parallel dot-products, and writes back tiles of **C**. The result is produced faster (green path), enabling higher throughput for the same power budget.



3. Design

To tackle this project, we adopted a structured, step-by-step methodology spanning from architectural design to FPGA prototyping.

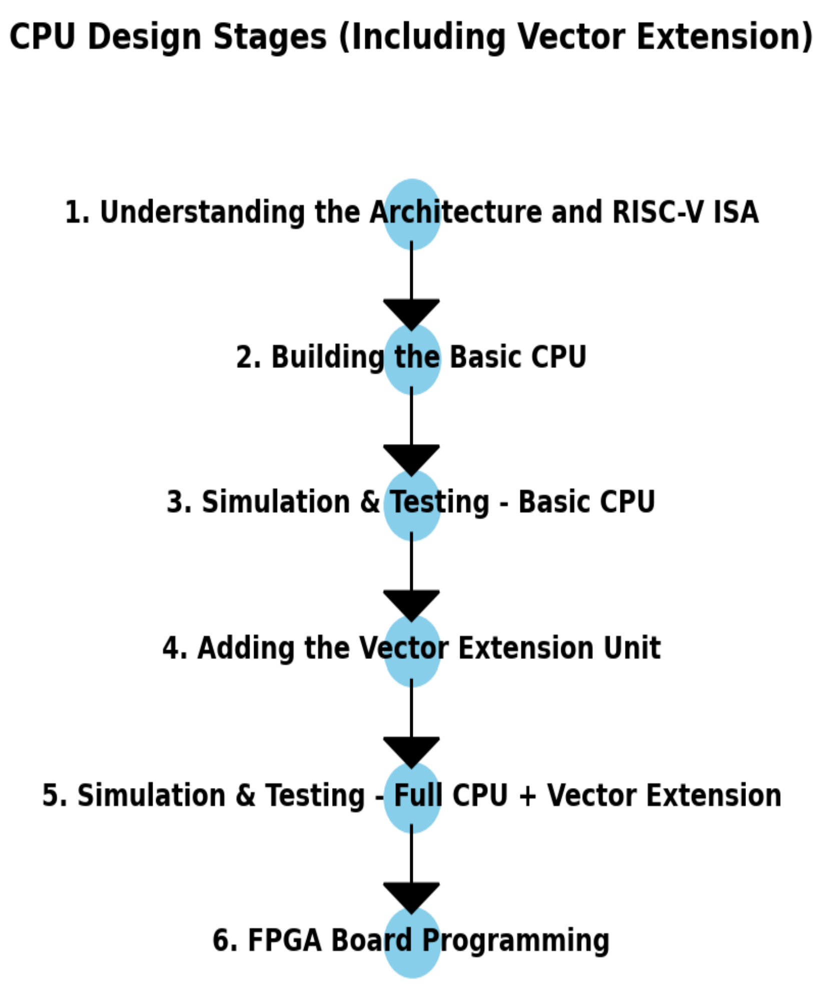

CPU Design Stages (Including Vector Extension)
1. Understanding the Architecture and RISC-V ISA
2. Building the Basic CPU
3. Simulation & Testing - Basic CPU
4. Adding the Vector Extension Unit
5. Simulation & Testing - Full CPU + Vector Extension
6. FPGA Board Programming

Figure 1 presents the development process, from architectural planning to FPGA prototyping.

3.1. System Design and Architecture

3.1.1 Block Diagram

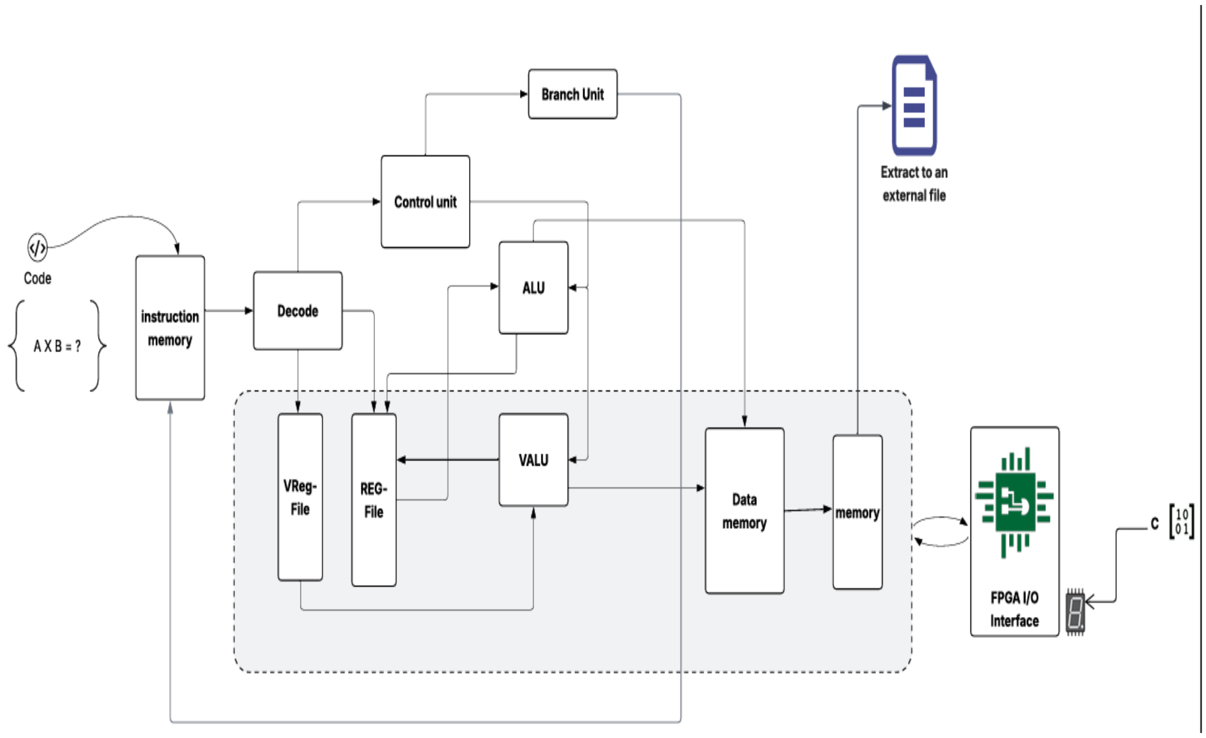


figure 1 – High level architecture

3.1.2 System Overview

RISCV-MatX integrates a scalar RISC-V RV32IM core with a custom vector unit (VALU + VRF). The CPU fetches instructions from Instruction Memory and decodes them. When a vector instruction is fetched, control is routed to the vector unit; otherwise, execution proceeds on the scalar ALU/branch path.

The vector dataflow is tile-oriented: tiles of A and B are loaded into the VRF (via V_LOAD). The VALU performs parallel dot-products on 128-bit lanes and accumulates the partial sums inside the VRF (accumulator registers). Partial sums never go to Data Memory during the inner loop—only when a tile completes do results get written back (V_STORE) to Data Memory, and then assembled into C = A×B.

The RTL is prototyped on Altera/Intel Cyclone II. Selected results and/or counters are shown on the board's 7-segment displays (e.g., C[0,0], checksum, or latency), while simulation can also export the full matrix to an external file.

Key properties: vector work triggers only on decoded vector opcodes; VRF holds live partial sums to minimize memory traffic; memory is touched mainly at tile boundaries—yielding the measured speedup (e.g., 3200→700 cycles).

3.1.3 Design methodology (Workflow)

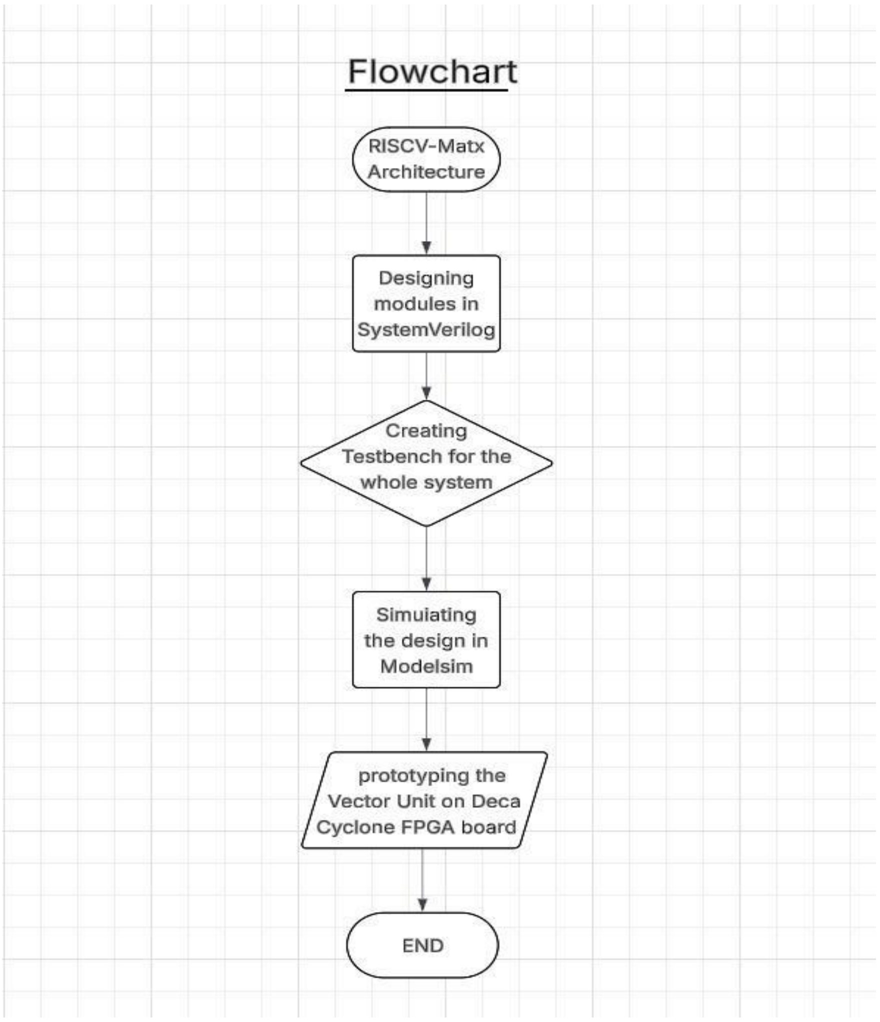
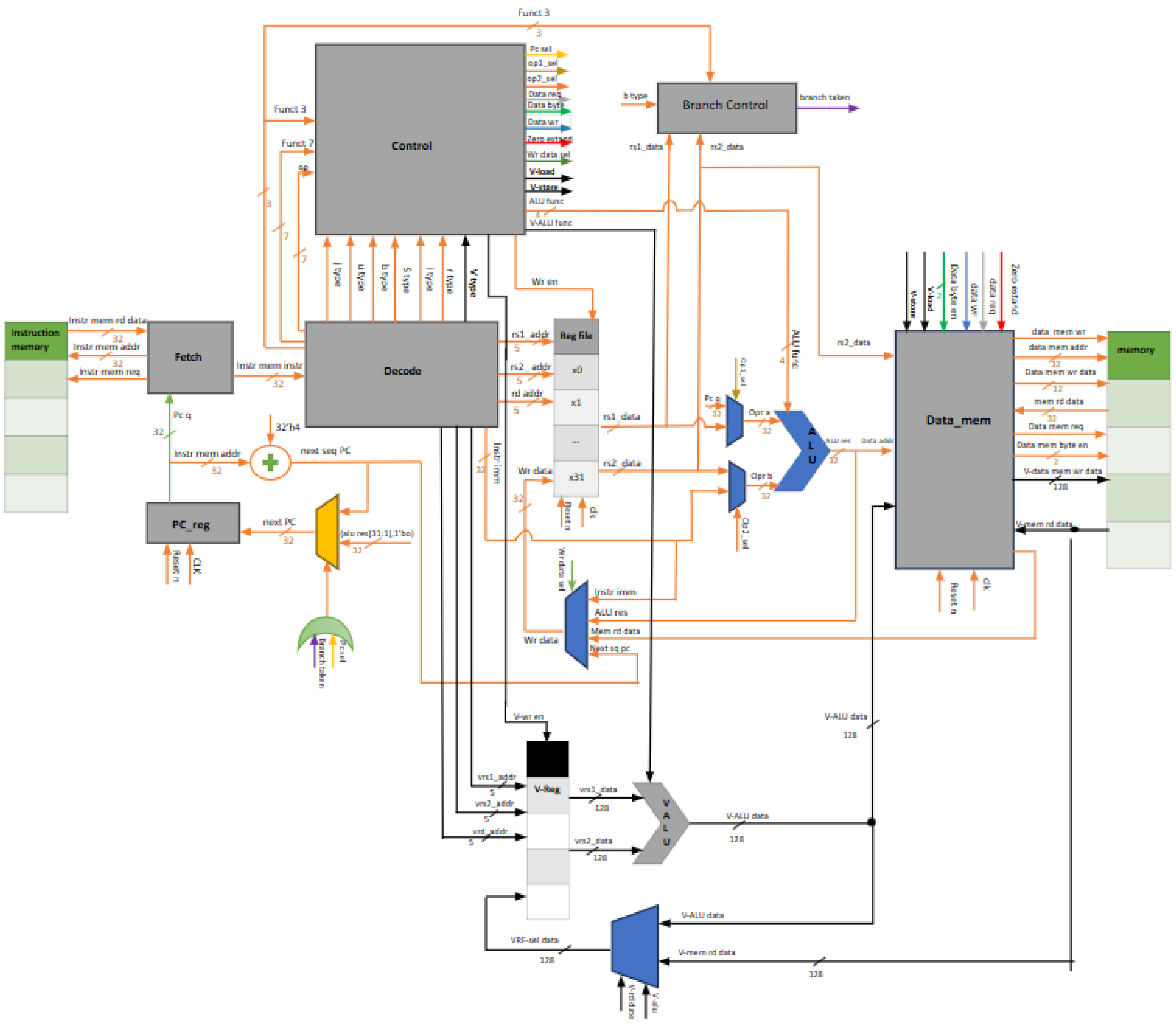

Flowchart
RISCV-Matx Architecture
Designing modules in SystemVerilog
Creating Testbench for the whole system
Simulating the design in Modelsim
prototyping the Vector Unit on Deca Cyclone FPGA board
END

figure 2: Design methodology workflow from architecture definition to FPGA prototyping

3.2 Detailed Hardware design



3.2.1 Component Description

Instruction memory – Stores the RISC-V program instructions, including both scalar and custom vector instructions like V_LOAD, V_MUL, and V_STORE. It feeds the decode stage with the next instruction to execute.

Decode – responsible for analyzing the binary instruction format, extracting key fields such as the opcode, register indices, and immediate values. It differentiates between scalar and vector instructions, enabling the system to select the appropriate data path accordingly**.**

Control unit- coordinates the execution flow across all system modules. It interprets decoded instructions to activate either the ALU or VALU and enables memory read/write, controls branching, and manages pipeline sequencing (even though the processor is not pipelined, basic sequencing still applies).

Branch- Handles control flow operations such as conditional or unconditional branches**.** Decides the next instruction address if a branch condition is met.

ALU (Arithmetic Logic Unit**)-** Executes standard scalar arithmetic and logical operations (e.g., add, sub, and, or). Used when executing normal RISC-V instructions not related to matrix/vector operations. Operates on operands fetched from the scalar register file.

REG-File (Scalar Register File)- Stores scalar values for standard RISC-V instructions. Provides operands for ALU and receives results from it.

VREG-File (Vector Register File)- Designed to store 128-bit wide vector operands. Used during V_LOAD/V_MUL/V_STORE instructions for matrix row/column data. Each register holds a vector of multiple elements for SIMD operations.

VALU (Vector ALU)- Supports parallel vector computations, primarily V_MUL. Operates on two vector operands and generates a vector result. Enhances matrix multiplication performance by reducing clock cycles.

Data Memory- Used for both scalar and vector data storage. Modified to support 128-bit wide accesses to accommodate vector operands. Communicates with VREG-File, REG-File, and VALU. Partial sums remain in VRF; Data Memory is written only at the tile completion via V_STORE.

Memory (Buffer) - Acts as temporary storage or buffer for data to be passed to the FPGA. Holds processed results or inputs before passing them to the FPGA or receiving new data from it.

FPGA I/O Interface- Physical interface between your processor and the FPGA board, displays final matrix multiplication result, allows hardware verification (observing RTL), and performs system testing.

Interface Definitions

- Instruction Memory → Decode Unit: Parallel 32-bit data bus.
- Decode Unit → Control Unit: Control signal set (bitwise control lines).
- Control Unit → ALU / VALU / Branch / Memory Units: Clock-synchronized control signal buses.
- Register Files → ALU / VALU: 32-bit for scalar, 128-bit for vector through direct register access.
- VRF → Data Memory: 128-bit vector loads/ stores. VALU exchanges data with the VRF (not directly with Data Memory).
- Data Memory → Memory Buffer: 128-bit internal bus.
- Memory Buffer → FPGA I/O Interface: Custom FPGA GPIO link for simulation.
- Control Unit → External File Extractor: Write-enable and address logic.

Data Flow

1- Input Stage:
- User inputs matrices A and B into instruction memory or data memory.

- Instructions for matrix multiplication (including custom vector ops) are fetched.

2- Processing Stage:
- Instruction is decoded; control unit routes to either ALU or VALU.
- V_LOAD loads matrix rows/columns from memory into vector registers.
- V_MUL performs element-wise multiplication using VALU.
- V_STORE writes the resulting vector back to memory.

3- Output Stage:
- Memory writes results to the output buffer.
- FPGA I/O Interface displays the result or transfers it to an external file.

3.2.2 Component Selection and Justification

Instruction memory- Used a Verilog-based ROM for full design control and cost efficiency. It supports custom instructions and allows consistent simulation behavior.

Decode and Control Unit- Implemented using custom FSMs in Verilog for seamless compatibility with extended instruction set and fine-grained control of execution. No IP cores from outside vendors were used.

ALU- Basic operations (ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU, ADDI, ANDI, ORI, XORI, SLLI, SRLI, SRAI, LUI, JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU, LW, SW) implemented directly in RTL. No external cores required.

Vector ALU (VALU): Custom 128-bit SIMD ALU designed in Verilog. The vector unit supports three SIMD instructions: V_LOAD to load vectors from memory, V_MUL for element-wise multiplication, and V_STORE to store vectors back to memory. All are implemented in Verilog using a custom 128-bit pipelined ALU optimized for matrix multiplication on FPGA.

REG-File- 32 x 32-bit registers modeled in Verilog per RISC-V convention. Fast access and compatible with scalar datapath.

VREG-File- Designed using Block RAM to store wide 128-bit vectors efficiently. Achieves high throughput during V_LOAD, V_MUL, V_STORE.

Data memory- Supports 128-bit access via inferred Block RAM. Balances simulation speed and FPGA synthesis performance.

Memory- FIFO-like mechanism buffers data before output. Decouples memory access timing from I/O transfer, easing timing closure.

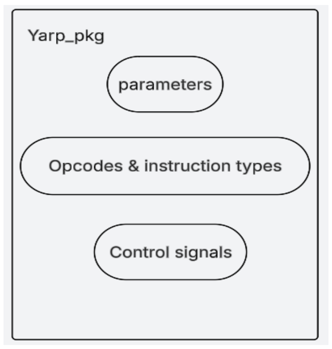
FPGA I/O Interface - Utilizes native GPIOs for LED output or serial emulation. Avoids IP overhead while maintaining simplicity.

3.3 Detailed Software design

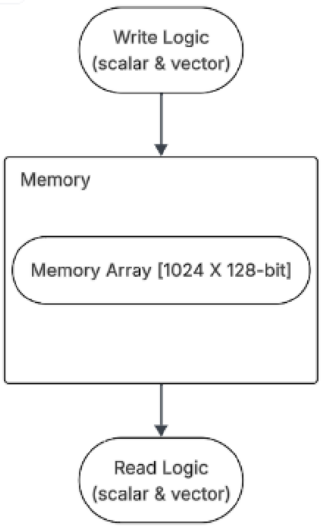
The software stack is structured around modular Verilog RTL blocks that together define the processor behavior and control matrix multiplication operations. Each block corresponds to a functional step in the overall data and control flow.

3.3.1 Component Description:

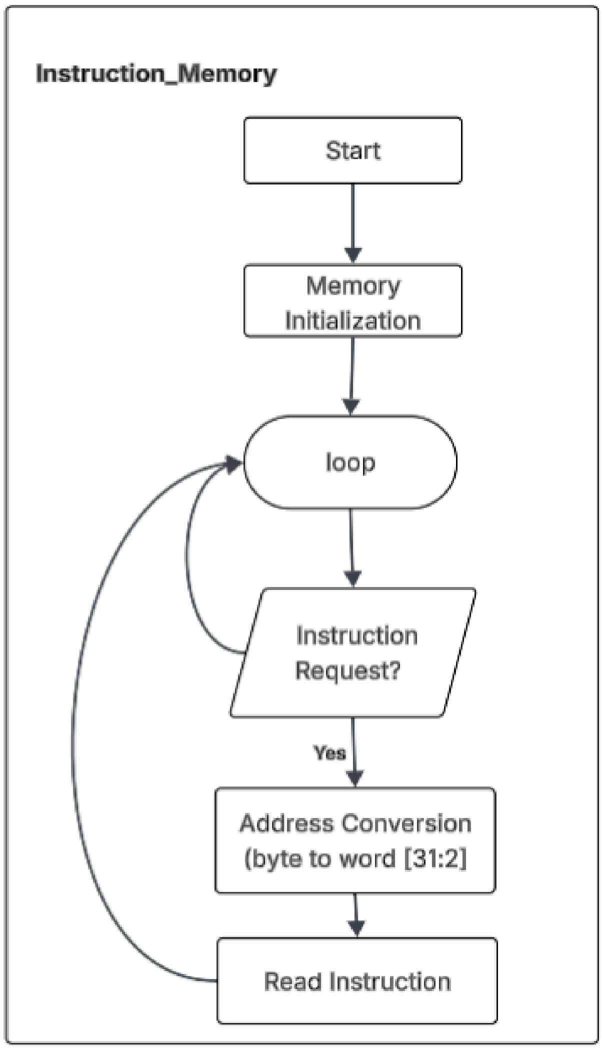
Yarp_pkg - Package definition, groups related parameters, types, and enums together. It shows how the parameters, instruction types, ALU/VALU operations, memory access, and control signals all connect and organize together.



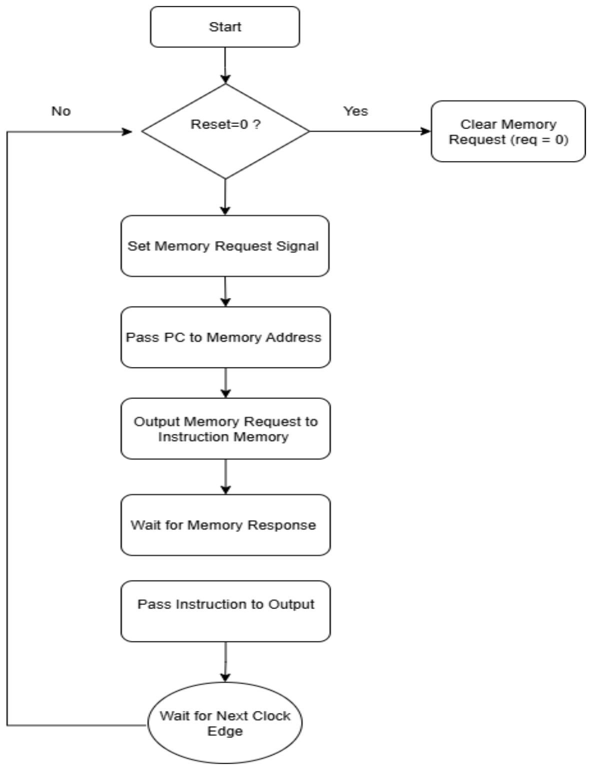
Memory- A dual-mode memory module supporting both scalar and vector operations.
Memory Structure:
Internal memory (memory_array) with 1024 entries, each 128 bits wide.
Allows accessing data either as 32-bit words (scalar) or as 128-bit × 4 vectors.


Write Logic (scalar & vector)
Memory
Memory Array [1024 X 128-bit]
Read Logic (scalar & vector)

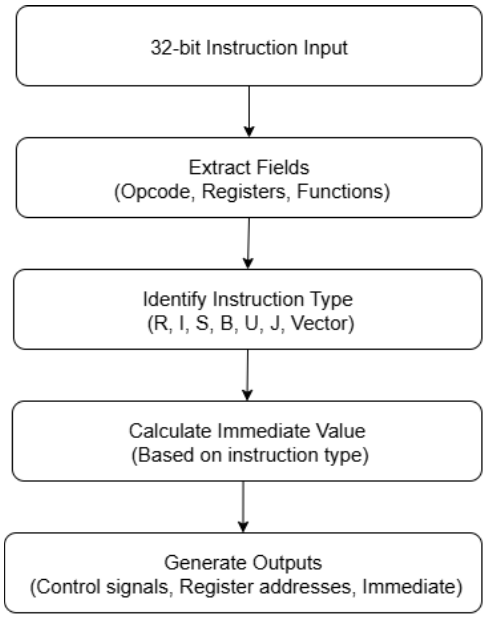
Instruction_memory- This module implements a simple instruction memory for a RISC-V processor. It stores 496 instructions, each 32 bits wide, in a memory array and initializes the memory at simulation start by reading the contents of a hex file (MA.txt). When an instruction memory request (instr_mem_req_i) is received, the module outputs the instruction corresponding to the given address. Since each instruction is 4 bytes, the module converts the input byte address to a word address by using the upper 30 bits of the address ([31:2]). This design provides a straightforward read-only memory (ROM) functionality for instruction fetching in the processor.

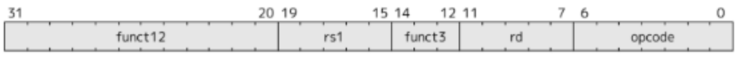


Fetch- responsible for retrieving instructions from the instruction memory. It continuously generates a memory request and uses the Program Counter (PC) as the address to fetch the next instruction. The fetched instruction is then forwarded directly to the decode stage. On reset, the memory request is cleared, and during normal operation it remains active, ensuring that one instruction is fetched every cycle. This simple design provides a constant instruction flow without stalling or additional control logic.

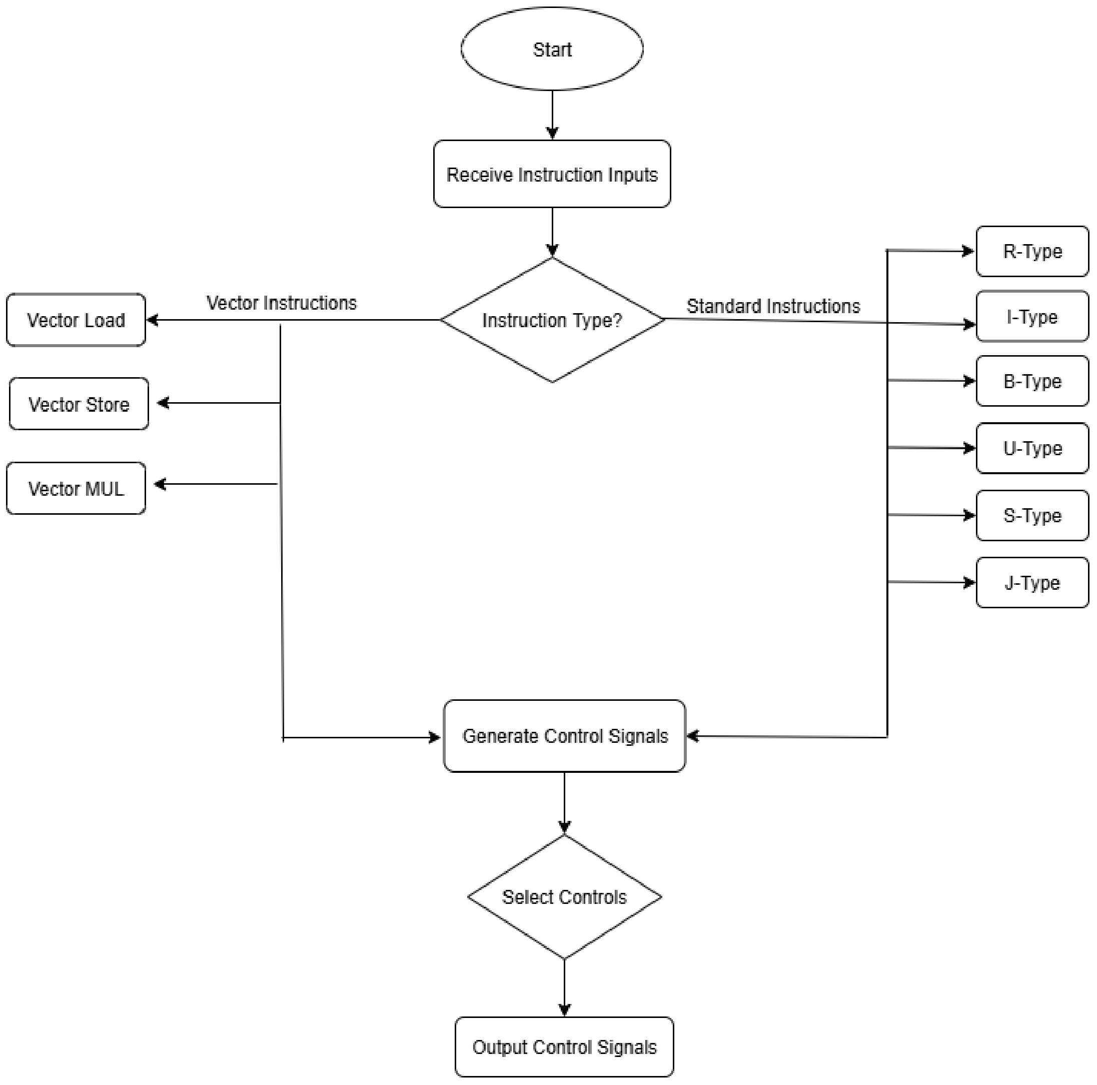

Start
Reset=0 ?
No
Yes
Clear Memory Request (req = 0)
Set Memory Request Signal
Pass PC to Memory Address
Output Memory Request to Instruction Memory
Wait for Memory Response
Pass Instruction to Output
Wait for Next Clock Edge

Decode- Parses the fetched 32-bit instruction (opcode, funct3/funct7, rd, rs1, rs2), generates the **control signals**, and routes the operation to the right execution path.

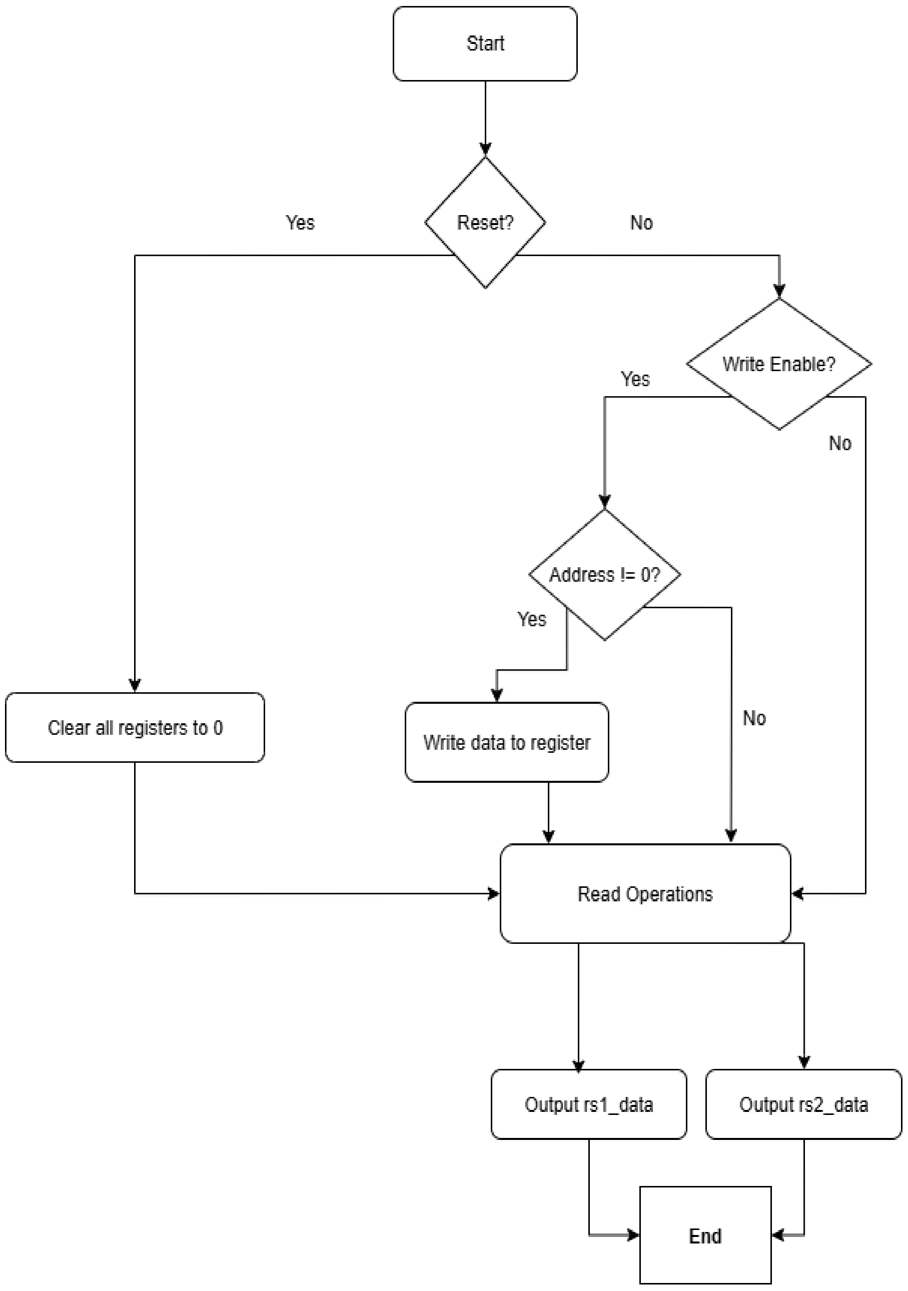

32-bit Instruction Input
Extract Fields (Opcode, Registers, Functions)
Identify Instruction Type (R, I, S, B, U, J, Vector)
Calculate Immediate Value (Based on instruction type)
Generate Outputs (Control signals, Register addresses, Immediate)



Control_Unit - The control unit module generates the control signals required to drive the processor's datapath. Based on the decoded instruction type (R, I, S, B, U, J, and vector instructions), it sets the correct ALU operation, memory access controls, register file write enables, and program counter selection. It ensures proper handling of arithmetic, logic, branching, jumps, loads, stores, and vector operations by selecting the right control structure. This allows seamless coordination between instruction decoding and datapath execution.



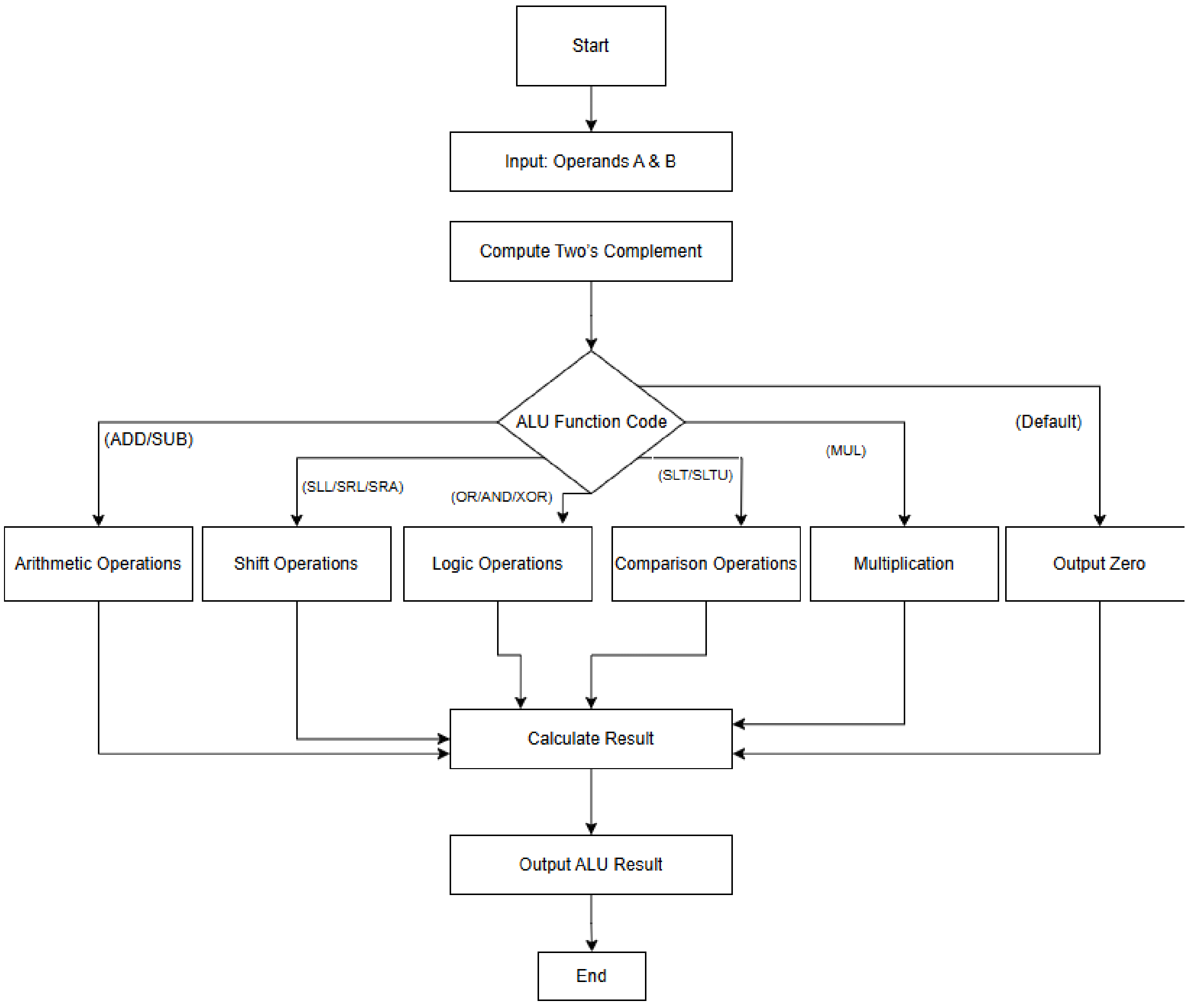
Reg_file – The register file module implements 32 registers, each 32 bits wide, for storing processor data. It supports synchronous write with reset initialization and asynchronous read from two source registers simultaneously. Register x0 is fixed at zero, ensuring compliance with the RISC-V specification.



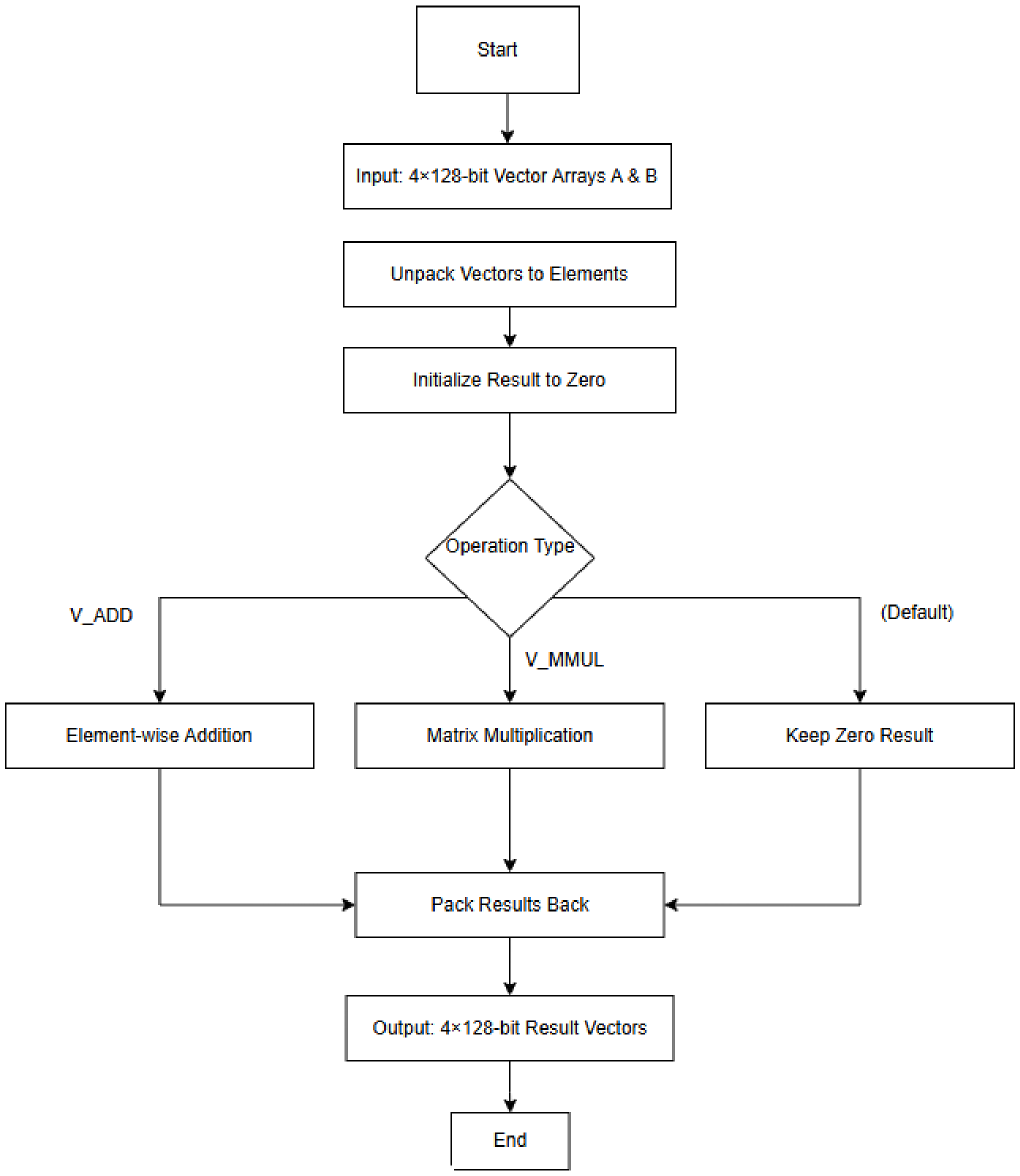
Vreg_file- The vector register file module stores and manages vector operands for matrix operations. It supports parallel read and write of four consecutive vector registers, enabling efficient access to rows of matrix A and columns of matrix B. Each register is 128 bits wide, divided into 32-bit elements. On reset, all registers are cleared, and on write enable, four registers are updated simultaneously. For debugging, the contents are dumped to a results file at the end of simulation.

Branch- The branch control module determines whether a branch instruction should be taken based on the operands and the instruction's funct3 field. It supports equality, inequality, signed and unsigned comparisons. The output signal indicates the final branch decision, which is enabled only for branch-type instructions.

Execute- The execute module is the ALU of the processor. It takes two 32-bit operands and a control code to perform arithmetic, logic, shift, comparison, or multiplication operations. The result is produced in one cycle and can also serve as a memory address for load/store instructions.



VALU- performs vector arithmetic for the processor. It unpacks input vectors into individual elements, executes element-wise operations, and packs the results back into vector form. Supported operations include matrix multiplication (V_MUL), V_LOAD and V_STORE, enabling acceleration of parallel computations such as dot products.


Start
Input: 4×128-bit Vector Arrays A & B
Unpack Vectors to Elements
Initialize Result to Zero
Operation Type
V_ADD
V_MMUL
(Default)
Element-wise Addition
Matrix Multiplication
Keep Zero Result
Pack Results Back
Output: 4×128-bit Result Vectors
End

Data_memory- implements a 32-word memory, each 128 bits wide, supporting both scalar and vector accesses. It handles synchronous writes on the clock, with byte, half-word, and word granularity for scalar operations, including optional zero or sign extension for loads. Vector reads and writes operate on four consecutive 128-bit words. At the end of simulation, the memory contents are automatically dumped to a file for verification.

3.4 List of CAD Tools

ModelSim was used extensively for RTL simulation and functional verification of the Verilog code.

- **Usage:**
  - Compiling and simulating both scalar and vector processor designs.
  - Running testbenches to verify correct execution of standard RISC-V instructions and custom vector instructions (V_LOAD, V_MUL, V_STORE).
  - Waveform analysis to debug timing mismatches and logic errors in the processor datapath.

Altera/Intel Quartus II
Quartus II served as the main synthesis and hardware implementation tool.

- **Usage:**
  - Synthesizing Verilog designs for FPGA implementation.
  - Performing pin assignments for FPGA I/O connections.
  - Running RTL Viewer to inspect the synthesized architecture.
  - Programming the FPGA board to test real-time behavior of scalar and vector operations.

4 Testing and Validation Procedures

Test Objectives: The purpose of the testing and validation in the project was to ensure that all components of the processor operate correctly, including the new instructions we added (such as V_MUL , V_LOAD , V_STORE), which were designed to improve performance in computational tasks. The tests included verifying the correctness of each module in the system, validating the correct execution of the new instructions, evaluating performance improvements in various scenarios, testing system stability with the RISC-V architecture — all to guarantee that the system is reliable, efficient, and standards-compliant.

Test Plan: Our test plan includes three main parts: unit tests, integration tests and system tests.

Unit Tests: Each component was tested individually (testbench for each module) – including the ALU, register file module, etc..... The tests ensured that the component is operating correctly.

The following diagram (figure 3) outlines the unit testing process, highlighting the verification flow applied to individual modules of the RISC-V design.

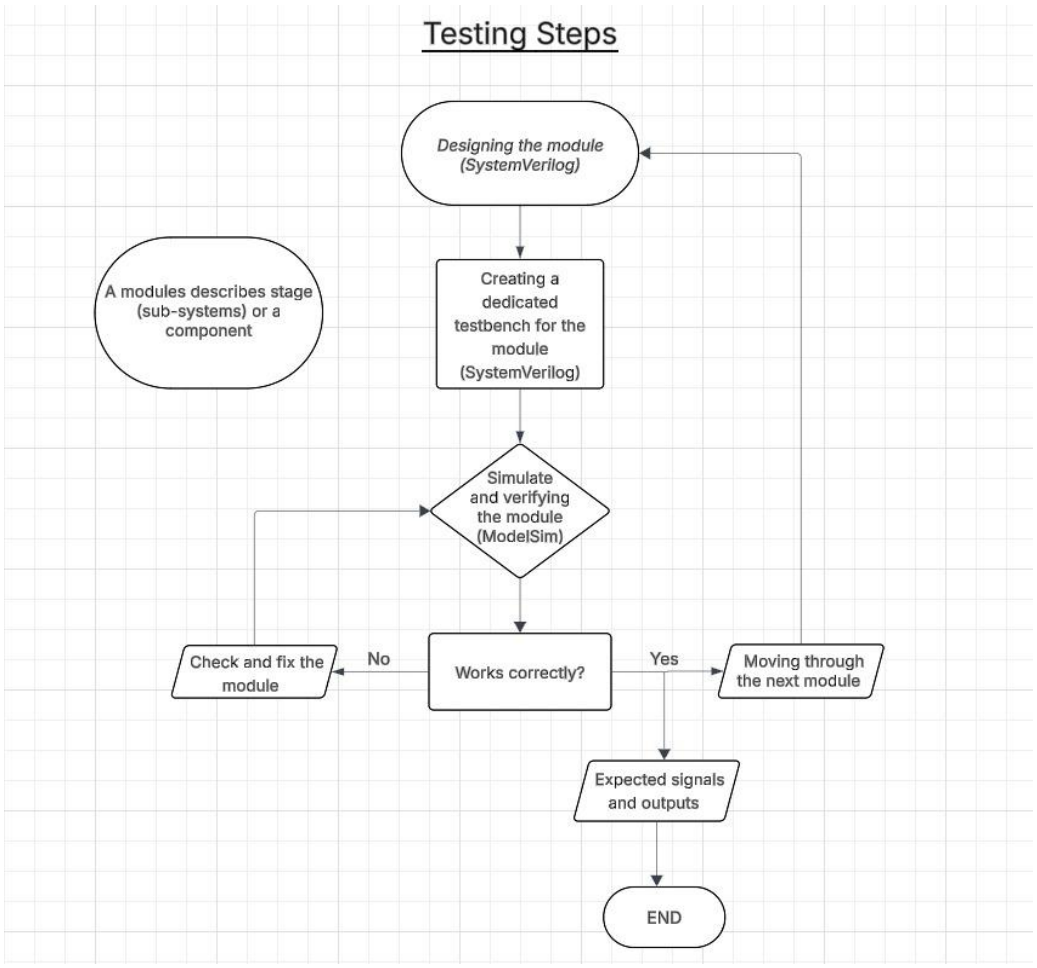

Testing Steps
Designing the module (SystemVerilog)
A modules describes stage (sub-systems) or a component
Creating a dedicated testbench for the module (SystemVerilog)
Simulate and verifying the module (ModelSim)
Check and fix the module
No
Works correctly?
Yes
Moving through the next module
Expected signals and outputs
END

figure 3: Modules testing steps

Integration Tests: After verifying the individual components works correctly, we conducted tests to examine the integration and coordination between them. That means each component receives the correct signals and inputs and performs the correct operations. This test is available once the modules are integrated by a **Top module**.

The diagram below, figure 4, illustrates the integration testing process, verifying the interaction and data flow between interconnected modules of the RISC-V design.

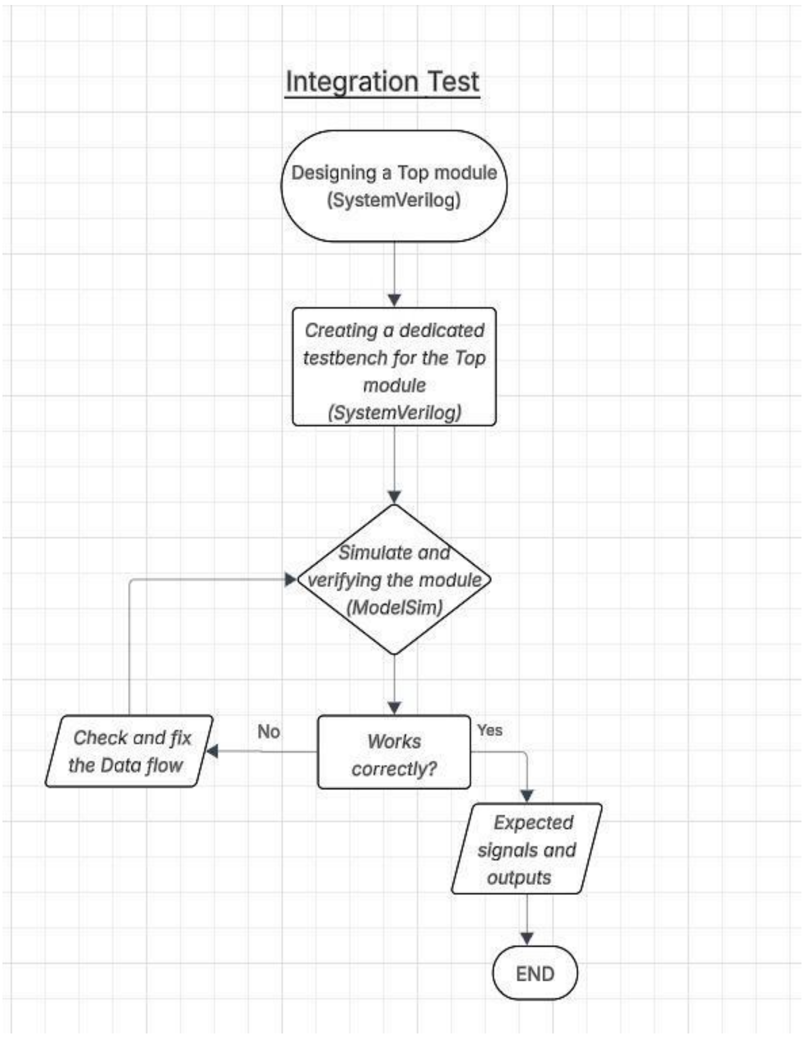

Integration Test
Designing a Top module (SystemVerilog)
Creating a dedicated testbench for the Top module (SystemVerilog)
Simulate and verifying the module (ModelSim)
Works correctly?
No
Yes
Check and fix the Data flow
Expected signals and outputs
END

figure 4: Integration testing process

System test: After constructing the complete architecture, we performed system testing using a full simulation to verify that the entire system functions correctly.
For example ,we executed a 4×4 matrix multiplication assembly program that uses the V_LOAD, V_MUL and V_STORE instructions.
We tracked the flow of data through all processor stages — fetch, decode, read, execute, and write-back — ensuring proper timing of control signals and correct data transfer between modules.
The computation results were compared to the expected output and confirmed to be accurate.

Figure 5 presents the approach adopted for system-level testing, validating the overall functionality of the integrated hardware design.

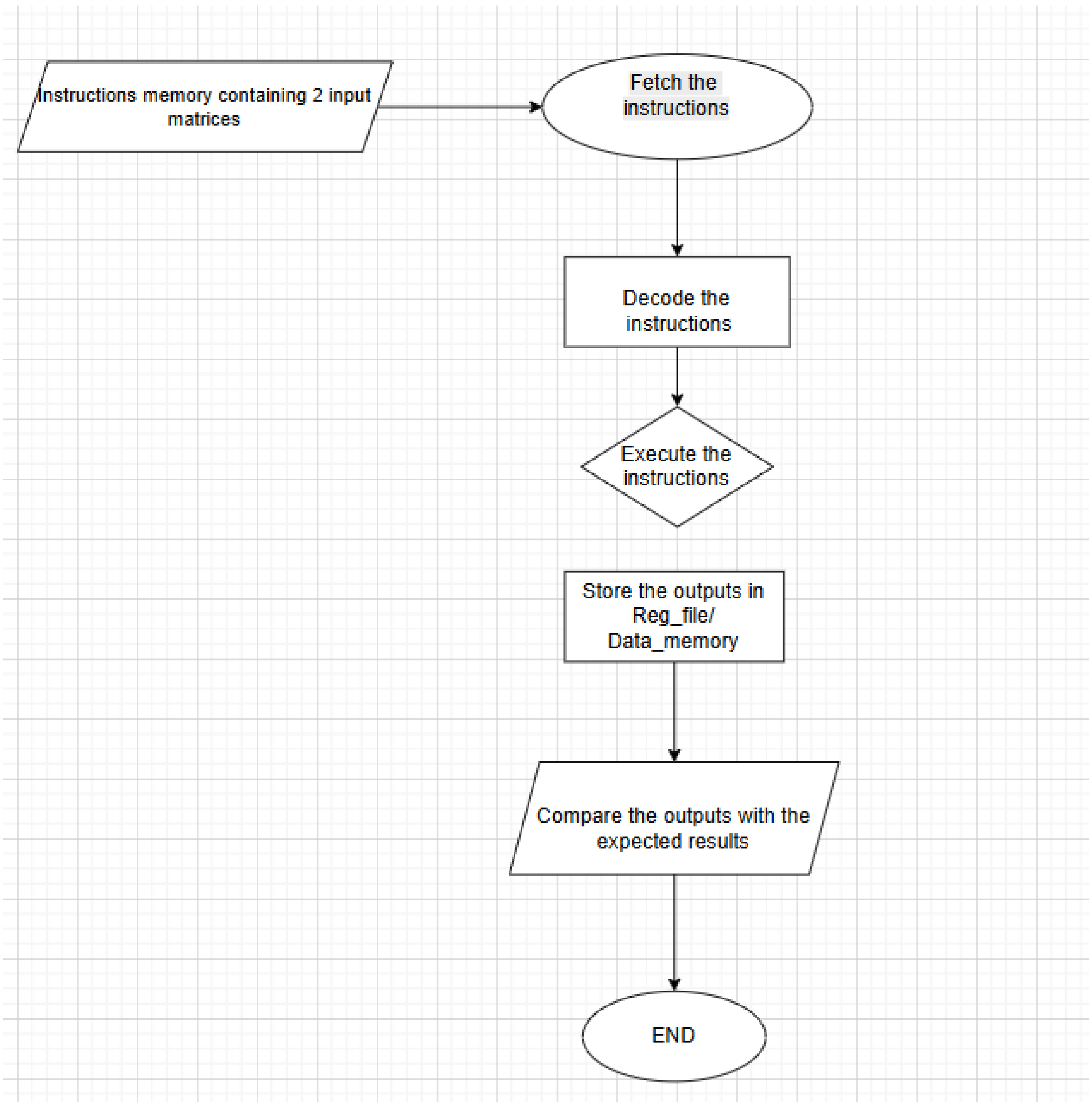

Instructions memory containing 2 input matrices
Fetch the instructions
Decode the instructions
Execute the instructions
Store the outputs in Reg_file/ Data_memory
Compare the outputs with the expected results
END

figure 5: System testing process

Test Execution: We conducted the test cases using the ModelSim simulation environment, where we wrote dedicated testbench files that matched the assembly code performing operations such as 4×4 matrix multiplication.
These files were used to run the full set of instructions on the custom-designed architecture and observe data flow, signal timing, and final outputs.
We documented the results using ModelSim's waveform viewer, comparing the actual output with the expected results.

The development environment included both ModelSim and Quartus for design, simulation and prototyping.

Results Analysis: After testing, we compared actual results to expected ones to identify deviations.
We evaluated performance improvements from our extensions (e.g. 4×4 matrix multiplication) vs. standard RISC-V code.

Error Handling and Troubleshooting: When we encountered errors — such as incorrect multiplication results or control signals not being triggered at the correct time — we used **waveform analysis** in ModelSim to locate the root cause of the issue. We examined the timing of all critical signals across clock cycles and compared the actual values to the expected ones.

Once a mismatch was identified, we went back to the relevant **Verilog code**, updated the control logic, data flow, and reran the simulation to verify that the issue was resolved.This process was repeated several times until the system functioned correctly and reliably.

Test coverage: We conducted tests for key scenarios to ensure the system operates correctly:

These included fixed 4x4 size matrix multiplication, where we verified the accuracy of the results. Different values were inserted to check results. We ensured that the computed results were correctly stored back into memory.

Test Documents: Throughout the project, we carefully documented the entire testing process, including test plans, test cases, and actual results. This documentation ensures transparency, reproducibility, and traceability of the validation process. For clarity, we also included screenshots of the Excel-based test table, where each subsystem was tested independently and validated for correctness.

All testbenches are added to the folder in the link at section 9

Specific Component Testing Details

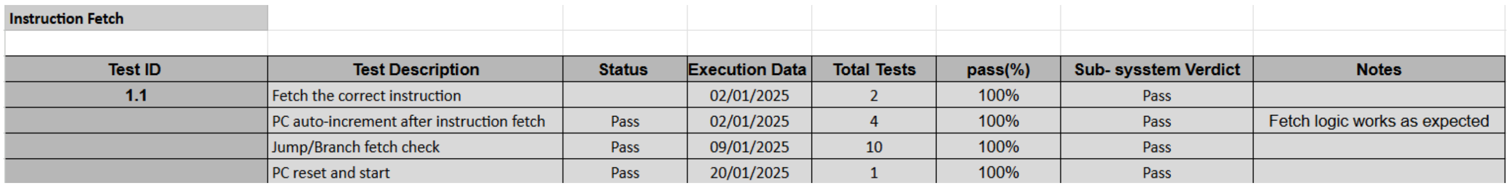
1. Instruction Fetch stage Testing:

Objective: To ensure the correct instructions are fetched, the Program Counter (PC) updates automatically, the system resets correctly, and instructions are fetched from memory properly after executing a JUMP or BRANCH instruction.

Plan: Four scenarios were tested — sequential fetch, fetch after reset, fetch after a JUMP, and fetch after a BRANCH.

Execution: Each scenario was tested using simulation in ModelSim (via a dedicated testbench), monitoring the PC values and the instructions fetched from instruction memory.

Results: In all cases, instruction fetch was performed correctly. The PC incremented correctly when no jump or branch occurred, and in cases of JUMP or BRANCH, it correctly pointed to the target address. The results matched expectations — **100% success**.



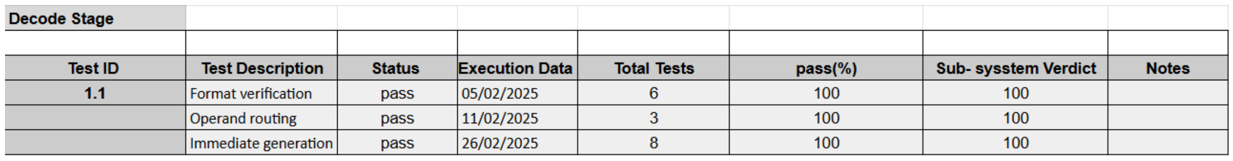
Instruction Fetch

Test ID	Test Description	Status	Execution Data	Total Tests	pass(%)	Sub- sysstem Verdict	Notes
1.1	Fetch the correct instruction		02/01/2025	2	100%	Pass	
	PC auto-increment after instruction fetch	Pass	02/01/2025	4	100%	Pass	Fetch logic works as expected
	Jump/Branch fetch check	Pass	09/01/2025	10	100%	Pass	
	PC reset and start	Pass	20/01/2025	1	100%	Pass	

2. Decode stage Testing:

Objective: To verify correct decoding of instruction formats, proper routing of operands, correct generation of the immediate value, and appropriate control signal generation based on the instruction type.
Plan: Test a variety of instruction types (R-type, I-type, S-type, and V-type) and ensure that fields such as rs1, rs2, rd, opcode, etc., are correctly identified.
Execution: This decode stage was tested by ModelSim (via dedicated testbench). The output signals from the Decode and Control modules were observed.

Results: All instructions were decoded correctly. Immediate values were generated properly for each format, and the appropriate control signals were routed — **100% success**.



Decode Stage

Test ID	Test Description	Status	Execution Data	Total Tests	pass(%)	Sub- sysstem Verdict	Notes
1.1	Format verification	pass	05/02/2025	6	100	100	
	Operand routing	pass	11/02/2025	3	100	100	
	Immediate generation	pass	26/02/2025	8	100	100	

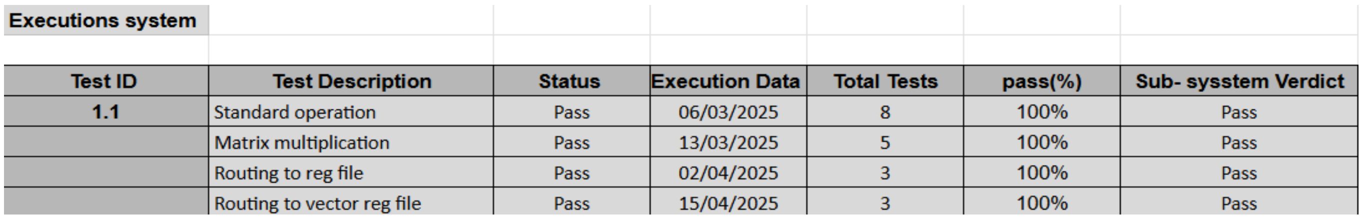
3.Execution stage Testing:

Objective: To test the operation of the Arithmetic Logic Unit (ALU), VALU (vector arithmetic logic), branch logic, and execution operation selection based on the instruction.

Plan: Five test cases were executed, including standard instructions (such as ADD, SUB), vector instructions (such as V_MUL, V_LOAD, V_STORE), and conditional branch logic scenarios.

Execution: All cases were simulated in ModelSim, analyzing the input and output signals of both the ALU and the Vector ALU. Signal timing and the selected execution path were closely observed.

Results: All instructions were executed as expected. The results were correct, and the execution path was selected according to the instruction type. All test cases passed



Executions system

Test ID	Test Description	Status	Execution Data	Total Tests	pass(%)	Sub- sysstem Verdict
1.1	Standard operation	Pass	06/03/2025	8	100%	Pass
	Matrix multiplication	Pass	13/03/2025	5	100%	Pass
	Routing to reg file	Pass	02/04/2025	3	100%	Pass
	Routing to vector reg file	Pass	15/04/2025	3	100%	Pass

successfully — **100% success**.

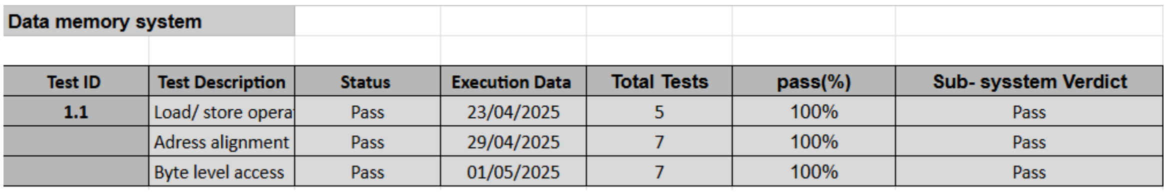
4.Data Memory Access stage testing :

Objective: To ensure that memory read and write operations are performed correctly, that addresses are properly aligned, and that byte-level access is supported.

Plan: Four cases were tested — standard read (LOAD), standard write (STORE), block loading using V_LOAD, and result storing using V_STORE.

Execution: The operations were simulated in ModelSim while monitoring address values, the data being read or written, and the activation of control signals such as MemRead and MemWrite.

Results: All operations were executed correctly — the data read matched the previously stored values, addresses were properly aligned, and write operations occurred only when the write signal was asserted. **100% success**.



Data memory system

Test ID	Test Description	Status	Execution Data	Total Tests	pass(%)	Sub- sysstem Verdict
1.1	Load/ store opera	Pass	23/04/2025	5	100%	Pass
	Adress alignment	Pass	29/04/2025	7	100%	Pass
	Byte level access	Pass	01/05/2025	7	100%	Pass

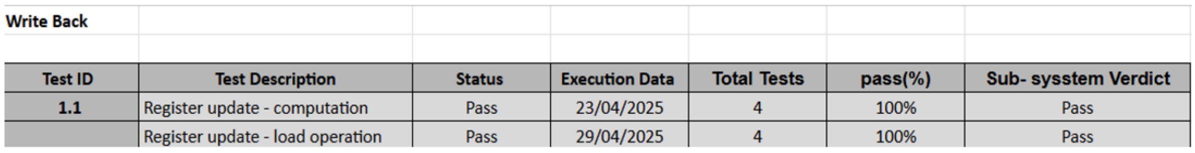
5.Write back stage testing :

Objective: To verify that results from the ALU or data memory are correctly written to the registers or to the data memory, ensure that matrix results are stored as expected, and confirm that writing occurs only when the RegWrite signal is asserted.

Plan: Four different operations were tested — writing a standard computation result, writing a value loaded from memory, writing a result from the Vector ALU, and attempting to write when the RegWrite signal is not active (to ensure no unintended write occurs).

Execution: The operations were simulated in ModelSim, monitoring the register contents, the RegWrite control signal, and the destination register address (rd).

Results: In all cases, values were correctly written to the intended register when required, and no unintended writes occurred when the control signal was inactive. All test scenarios passed successfully — **100% success.**



Write Back

Test ID	Test Description	Status	Execution Data	Total Tests	pass(%)	Sub- sysstem Verdict
1.1	Register update - computation	Pass	23/04/2025	4	100%	Pass
	Register update - load operation	Pass	29/04/2025	4	100%	Pass

5 Implementation

5.1 Hardware Design and Development

Introduction:
The hardware implementation of this project involved designing, simulating, and prototyping a custom RISC-V single-cycle processor with an integrated vector processing unit for accelerating matrix multiplication. The processor was described in Verilog HDL, verified through simulations, and deployed onto an FPGA (Altera/Intel Cyclone II board) to achieve real-time hardware execution of both scalar and vector instructions.

Design and Construction Stages

1. Component Selection:

- **CPU Core:** Single-cycle RISC-V processor for base integer instructions.
- **Vector ALU (VALU):** Designed for 128-bit parallel operations, enabling efficient 4×4 matrix multiplication.
- **Vector Register File (VRF):** Dedicated storage for vector operands to optimize throughput.
- **Instruction and Data Memory:** Separate modules for storing instructions, input matrices, and results.
- **FPGA Board (Altera/Intel Cyclone II):** Chosen for its sufficient logic resources, on-chip RAM, and integrated seven-segment displays.
- **Integrated Seven-Segment Display:** Used to directly show computation results without an external interface.

2. Assembly and Integration:

- Each module (CPU, VALU, VRF, memories) was coded and verified independently using ModelSim.
- Modules were interconnected to form a functional processor system supporting both scalar and custom vector instructions.
- The design was synthesized in Quartus and programmed onto the FPGA board.
- Results of matrix multiplication were mapped to the built-in seven-segment display for real-time monitoring.

3. Challenges and Considerations:

- **Timing Constraints:** The single-cycle architecture caused long combinational paths, making timing closure on the FPGA challenging. This was mitigated by careful optimization of the VALU logic and memory access paths.
- **Resource Utilization:** The 128-bit vector operations consumed significant FPGA logic and memory resources. Module design was optimized to fit within available hardware limits.

- **Debugging on FPGA:** Initial tests showed incorrect results due to signal propagation delays. SignalTap and simulation traces were used to locate and resolve these issues before successful final deployment.

- **Result Display Mapping:** Properly sequencing and formatting 16 matrix elements for seven-segment display required an additional control mechanism to cycle through results automatically.

Summary:
The hardware implementation successfully demonstrated a RISC-V processor with custom vector instructions for matrix multiplication. Despite challenges in timing, debugging, and resource usage, the final FPGA implementation executed parallel computations correctly, with results displayed on the integrated seven-segment interface.

5.2 Software Development

The software development phase focused on creating simulation and verification environments for the custom RISC-V processor with a vector extension. Although the project was hardware-centric, software components were essential for functional testing and FPGA deployment of the design.

Programming Languages, Tools, and Frameworks:

- **SystemVerilog / Verilog HDL:** Used to develop testbenches that initialized instruction memory, data memory (with two fixed 4×4 matrices), and monitored the processor's execution flow.
- **ModelSim:** Provided the simulation environment for debugging the processor and verifying correctness before synthesis.
- **Quartus II:** Used for synthesis, timing analysis, and FPGA bitstream generation for hardware prototyping.

Coding and Testing Process:

- A fixed test program was written and loaded into the instruction memory to execute a 4×4 matrix multiplication using the custom vector instructions.
- Testbenches initialized data memory with predefined constant matrices and observed the results written back after computation.
- Simulation logs and waveforms were analyzed to ensure that all scalar and vector instructions were executed correctly.
- Once functional correctness was validated, the design was synthesized in Quartus and programmed onto the Altera/Intel **Cyclone II** FPGA for real-time verification on the seven-segment display.

Challenges and Considerations:

- Ensuring proper synchronization between instruction fetch, memory access, and vector execution in the test environment.
- Debugging discrepancies between simulation results and FPGA behavior due to propagation delays and timing issues.
- Creating a simple yet effective testbench that fully exercised the vector extension without requiring complex external software tools.

The software implementation relied entirely on HDL-based testbenches and simulation workflows. This allowed thorough verification of the processor and its vector unit using fixed 4×4 matrix inputs before successful deployment to the FPGA platform.

5.3 Integration of Hardware and Software

Introduction:
The integration phase ensured seamless communication between the custom RISC-V processor hardware and the HDL-based test environment. Since this project was primarily hardware-focused, the "software" component consisted of HDL testbenches and predefined instruction/data memory contents that interfaced directly with the hardware modules.

Interfaces and Communication:

- Instruction and Data Memory Interface: The testbench initialized program instructions and two fixed 4×4 matrices in memory, providing the processor with the required inputs.
- Internal Buses: Scalar and vector execution units shared a unified data path, with dedicated signals for 128-bit vector operations.
- Result Output: Computed results were automatically written to data memory and then displayed on the FPGA's built-in seven-segment displays through a control mechanism cycling through 16 matrix elements.

Steps for Seamless Integration:

1. Verified individual modules (CPU, VALU, VRF, memories, display controller) independently in simulation.
2. Loaded instruction memory with a predefined test program for matrix multiplication.
3. Connected hardware modules with testbench signals, ensuring proper synchronization between instruction fetch, memory read/write, and result output.
4. Validated integration in simulation before deploying to FPGA.

Summary:
The system integration relied on HDL-based software stimuli feeding the custom hardware design. Memory-based data exchange and synchronized control signals ensured that the processor executed the intended operations and displayed results in real-time on the FPGA.

5.4 Deployment and Installation

Introduction:
The deployment phase consisted of synthesizing the designed RISC-V processor with vector extension, generating the programming file, and loading it onto the Altera/Intel **Cyclone II** FPGA to demonstrate real-time execution of matrix multiplication.

Deployment Process:

1. **Compilation:** The full design, including CPU core, VALU, VRF, memories, and display controller, was compiled in Quartus II to verify functionality and timing requirements.

2. **FPGA Programming:** The generated. sof bitstream was uploaded to the Altera/Intel **Cyclone II** board using the USB-Blaster connection.

3. **Execution and Output:** Fixed 4×4 input matrices were preloaded into data memory, and the FPGA system executed the multiplication, displaying results automatically on the integrated seven-segment display.

4. **Validation:** Several test runs were carried out to confirm correct functionality, ensuring the deployed design behaved as expected on hardware.

Readiness and Limitations:
The project successfully demonstrated the intended hardware functionality and real-time processing on FPGA. The main limitation is the fixed matrix size and lack of dynamic input loading, restricting the design to predefined test cases only.

5.5 Configuration and Setup

The configuration and setup process involved preparing the FPGA environment and initializing the system for execution. The steps included:

1. **Project Setup in Quartus:** Importing all Verilog modules (CPU, VALU, VRF, memory, display controller) into a Quartus project and setting the Altera/Intel **Cyclone II** board as the target device.
2. **Pin Assignment:** Mapping FPGA I/O pins for clock, reset, and seven-segment display outputs according to the board's datasheet.
3. **Memory Initialization:** Loading predefined 4×4 input matrices and instruction sequences into the instruction and data memory modules before synthesis.
4. **Compilation and Bitstream Generation:** Running synthesis, fitting, and timing analysis to ensure successful bitstream generation.
5. **FPGA Programming:** Uploading the generated .sof file to the FPGA and resetting the system to start computation.

No calibration or runtime adjustments were required, as all input data and parameters were fixed and hardcoded during design and synthesis.

5.6 Bill Of Materials (BOM)

Component	Description	Quantity
FPGA Development Board	Altera/Intel Cyclone II with onboard RAM and seven-segment displays	1

Computer with Quartus II & ModelSim	Used for hardware description, simulation, and bitstream generation	1
USB-Blaster Programmer	Interface for uploading the compiled design to the FPGA board	1
Verilog HDL Source Files	Custom modules for CPU core, VALU, VRF, memory units, and display controller	N/A (digital files)
Instruction Memory Initialization File	Preloaded with program instructions for matrix multiplication	1
Data Memory Initialization File	Preloaded with two fixed 4×4 matrices as input data	1
Power Supply (USB)	Provides power to the FPGA board	1

6 Results and Evaluation

In this chapter, we will comprehensively evaluate the performance and functionality of the RISCV-MatX system, examining how effectively it meets the project's objectives. Key aspects addressed include:

Performance Analysis:

- Assessing the efficiency and accuracy of matrix multiplication operations by comparing the baseline RISC-V processor with the enhanced version that includes custom ISA extensions.
- Evaluating the execution speed and correctness of the newly implemented vector instructions (V_MUL, V_LOAD, V_STORE), as well as the system's overall stability under varying computational loads.

Functional Assessment:

- Verifying that each hardware component — including the vector ALU, extended register file, and memory modules — functions correctly after integration with the base processor.
- Ensuring seamless integration between the new vector unit and the existing control and data paths, including compatibility with the FPGA testing environment.

Achievement of Project Goals:

- Determining whether the system meets the defined project goals: accelerating matrix operations, supporting vector instructions, and demonstrating correct behavior on real hardware.
- Analyzing how the system improves performance and usability in matrix-based applications compared to a non-extended RISC-V implementation.

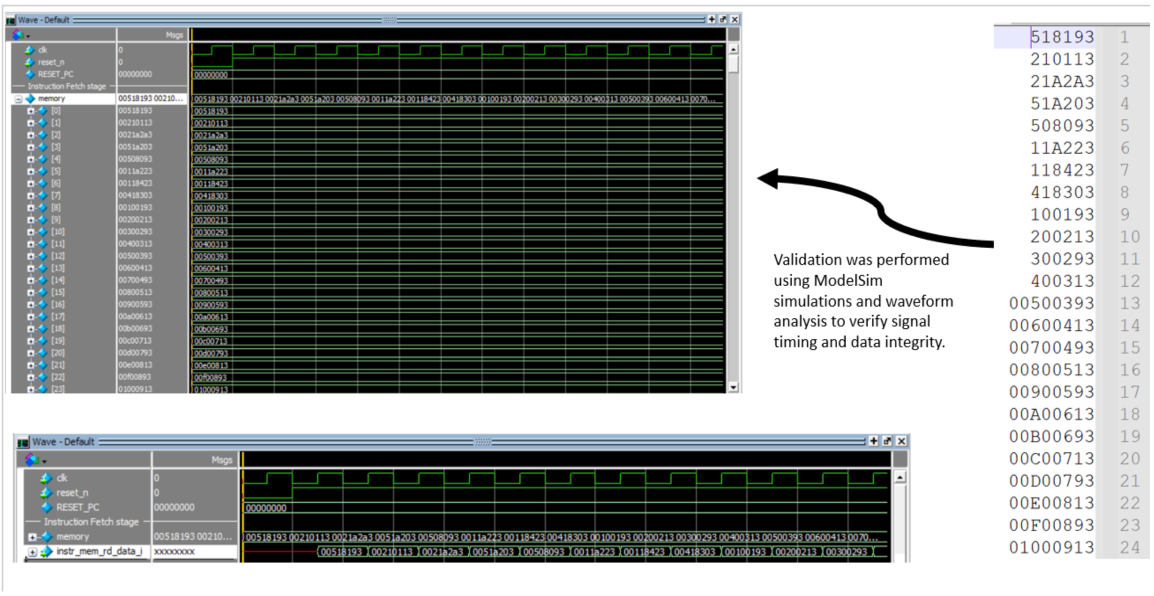
Test Results

This section presents the test results and validation findings performed during the project process, including quantitative data, performance metrics, and conclusions gathered in the testing stages.

Individual Component Tests

1. Instruction Fetch Stage

The instruction fetch was verified by simulating sequential PC increments and ensuring the correct instruction was loaded from the instruction memory.



Validation was performed using ModelSim simulations and waveform analysis to verify signal timing and data integrity.

```
  518193
  210113
  21A2A3
  51A203
  508093
  11A223
  118423
  418303
  100193
  200213
  300293
  400313
00500393
00600413
00700493
00800513
00900593
00A00613
00B00693
00C00713
00D00793
00E00813
00F00893
01000913
```

figure 6: Instruction Fetch Validation: Fetching instructions from Instruction Memory and verifying the fetch process in ModelSim simulation.

At this stage, the Instruction Memory was validated, preloaded with instruction content corresponding to the matrix multiplication sequence. Using ModelSim simulation, the signal **instr_mem_rd_data_i** displayed the value of the instruction fetched from memory in every clock cycle, showing full consistency between the fetched instructions and the values defined in memory. The waveforms indicated that with every change in the PC value, the correct instruction was fetched from the corresponding memory address. The instruction fetch stage successfully passed all validation tests, as demonstrated in the attached waveform diagrams.

2. Decode stage

In the Decode stage, the instruction fetched from the Instruction Memory is decoded into its respective fields (opcode, funct3, funct7, rs1, rs2, rd, immediate), which are then forwarded to the other processor units. The Control Unit receives these instruction fields and generates the necessary control signals that activate the ALU, Register File, vector units, and memory access paths according to the instruction type.

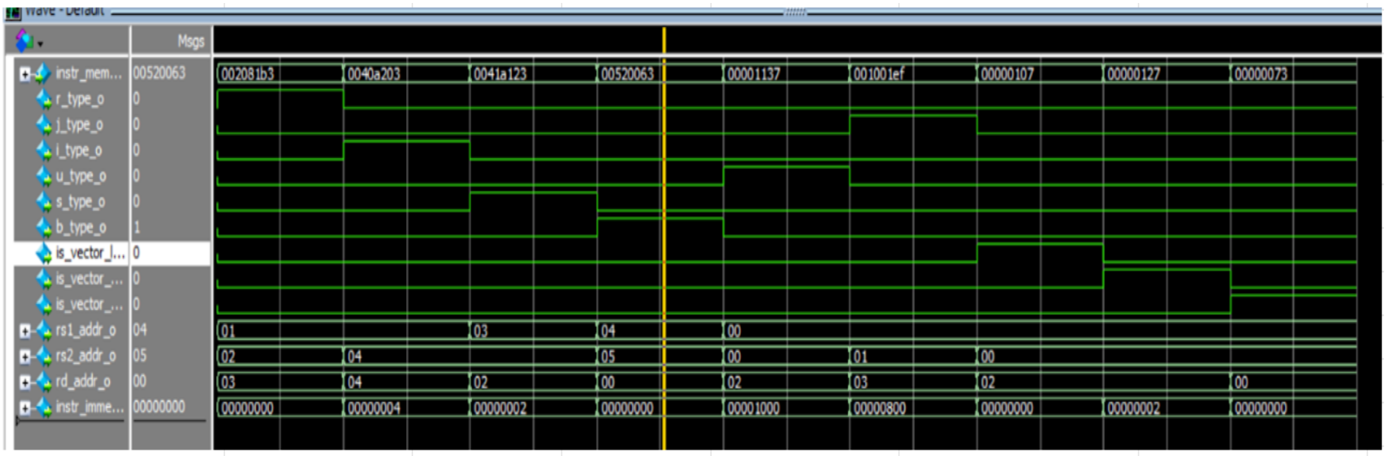


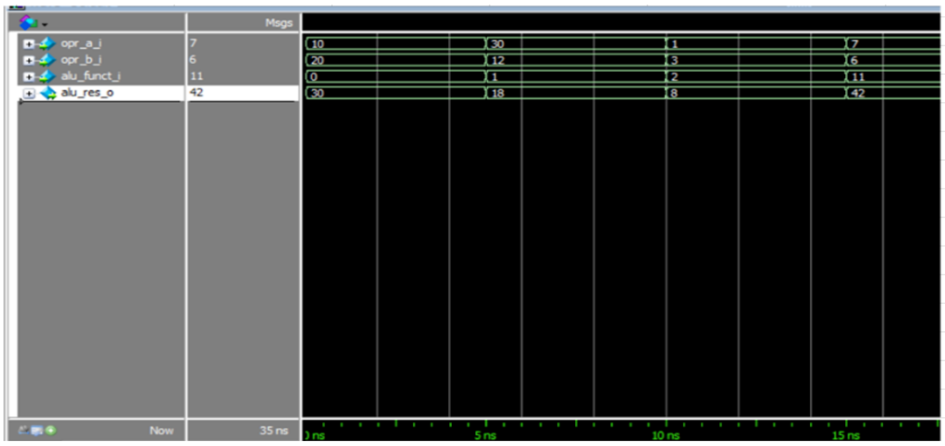
figure 7: Decode Stage Validation: Decoding instruction fields and generating control signals in ModelSim simulation.

At this stage, the Decode unit was validated by verifying the correct extraction of instruction fields (opcode, funct3, funct7, rs1, rs2, rd, immediate) from the fetched instruction. Using **ModelSim** simulation, it was confirmed that each instruction was accurately decoded into its respective fields, matching the predefined instruction format of the RISC-V ISA. The decoded values were then passed correctly to the control unit and register file. The simulation waveforms indicated that control signals (such as ALU func, V-type, V-load, V-store, and register addresses) were generated accurately according to the instruction type. The decode stage successfully passed all validation tests, as demonstrated in the attached waveform diagrams.

3.Execution stage

The Execution Stage is responsible for performing arithmetic, logical, and vector operations using the ALU and VALU, based on the decoded instruction and control signals received from the control unit.

Scalar Execution:



Vector Execution:

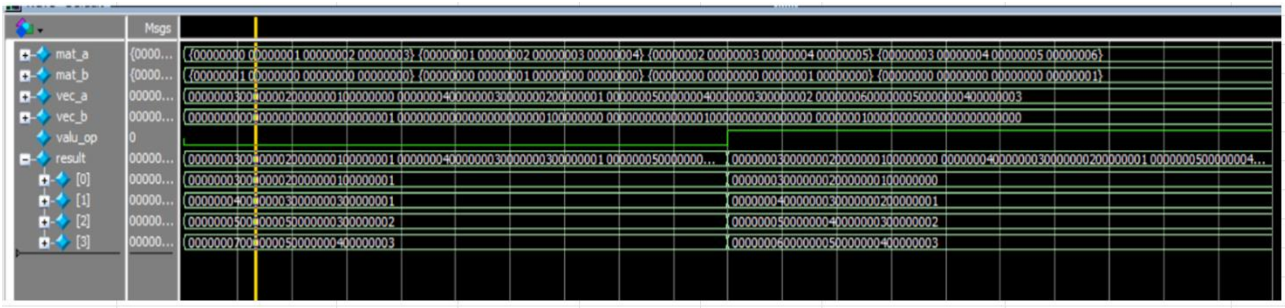


Figure 8: Executing arithmetic and vector operations and verifying output correctness in ModelSim simulation.

At this stage, the execution units, including the standard ALU and the extended VALU, were validated. Using **ModelSim** simulation, various arithmetic and vector instructions (e.g., ADD, SUB, V_MUL, V_LOAD, V_STORE) were applied, and the resulting outputs were compared against expected values. The simulation confirmed that the **ALU func** and **VALU func** control signals were correctly interpreted, and the operands provided from the register file were processed accurately. Waveform analysis showed that the execution outputs matched the intended results for all tested instructions, without timing errors or invalid states. The execution stage passed all validation tests, as demonstrated in the attached waveforms.

4.Data Memory Access stage

The Data Memory Access Stage is responsible for performing read and write operations to the Data Memory based on the instruction type, ensuring correct data flow between memory and registers.

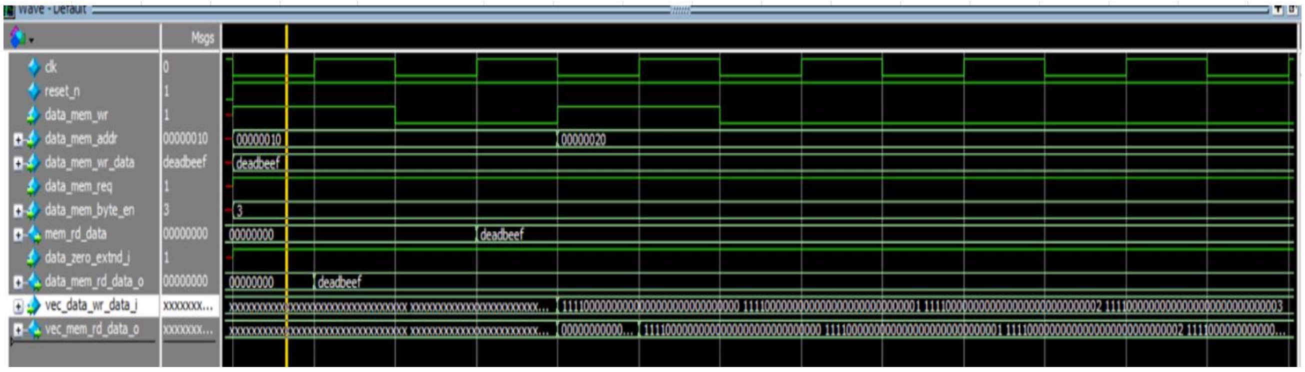


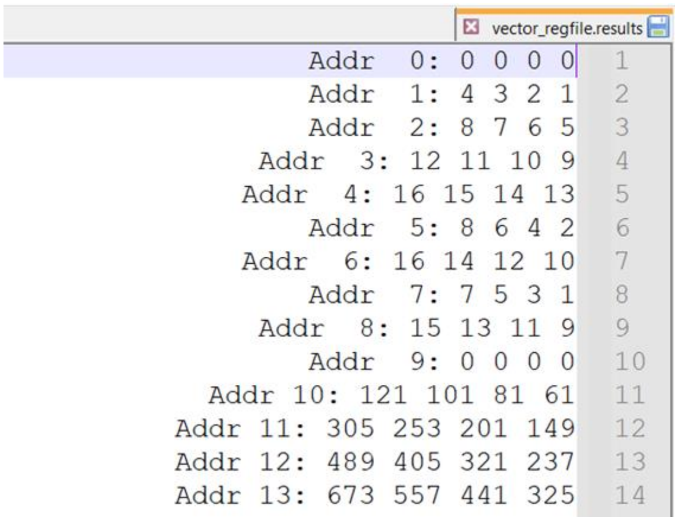
figure 9: Data Memory Access Stage Validation: Performing load/store operations and verifying memory address access in ModelSim simulation.

At this stage, the data memory operations were validated by executing load (LW, V_LOAD) and store (SW, V_STORE) instructions and verifying proper access to the designated memory addresses. Using ModelSim simulation, the effective memory addresses were computed correctly by the execution stage and passed to the memory access unit. The simulation confirmed that for each load instruction, the data was accurately read from the corresponding address and forwarded to the register file. Similarly, for store instructions, the correct data values were written back to the intended memory locations. Waveform analysis indicated full consistency between the address calculations, data bus activity, and control signals (Data wr, V-load, V-store). The data memory access stage successfully passed all validation tests, as demonstrated in the attached waveform diagrams.

5.Write back stage

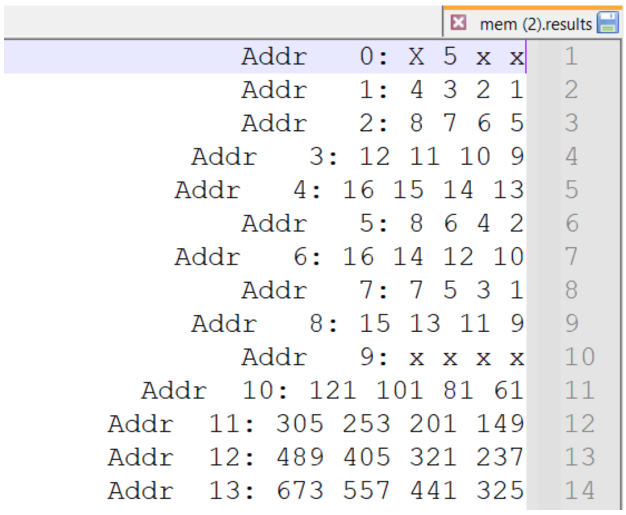
The Write-Back stage is responsible for writing the final result of an executed instruction back into the register file, vectore register file or the Data Memory, ensuring that computation outputs are correctly stored for subsequent operations

The Write-Back Stage is responsible for writing the final result of an executed instruction back into the register file , vector register file or data memory, ensuring that computation outputs are correctly stored for subsequent operations.


vector_regfile.results

```
Addr  0: 0 0 0 0
Addr  1: 4 3 2 1
Addr  2: 8 7 6 5
Addr  3: 12 11 10 9
Addr  4: 16 15 14 13
Addr  5: 8 6 4 2
Addr  6: 16 14 12 10
Addr  7: 7 5 3 1
Addr  8: 15 13 11 9
Addr  9: 0 0 0 0
Addr 10: 121 101 81 61
Addr 11: 305 253 201 149
Addr 12: 489 405 321 237
Addr 13: 673 557 441 325
```

Figure 10: Vector register file contents after execution, showing initial matrix values (Addr 0–9) and computed results (Addr 10–13).


mem (2).results

```
Addr  0: X 5 x x
Addr  1: 4 3 2 1
Addr  2: 8 7 6 5
Addr  3: 12 11 10 9
Addr  4: 16 15 14 13
Addr  5: 8 6 4 2
Addr  6: 16 14 12 10
Addr  7: 7 5 3 1
Addr  8: 15 13 11 9
Addr  9: x x x x
Addr 10: 121 101 81 61
Addr 11: 305 253 201 149
Addr 12: 489 405 321 237
Addr 13: 673 557 441 325
```

Figure 11: Data memory contents after execution, showing input matrices (Addr 0–9) and computed results (Addr 10–13).

At this stage, the correctness of the write-back process was validated by simulating instructions that write computation results into destination registers (rd). Using ModelSim simulation, the signal Wr_en (Write Enable) was monitored to ensure it was asserted only when a valid write-back was required. The data written to the register file was cross-checked against expected execution outputs (from ALU, VALU, or Data Memory). Waveform analysis confirmed that each instruction's result was correctly routed back to its destination register without overwrite errors or timing mismatches. The write-back stage passed all validation tests, as demonstrated in the attached waveform diagrams.

Algorithm Tests

Vector Multiplication Instruction (V_MUL) Test:

The V_MUL instruction was tested in a setup replicating the final configuration of the RISC-V processor, using preloaded vector data in the Data Memory. The instruction was executed to multiply two vectors element-wise, and the output was compared to expected results. The correctness percentage was **100%**, and the instruction timing matched the expected hardware cycle count.

Performance Evaluation

The performance evaluation of the system was conducted by comparing the execution time of a 4x4 matrix multiplication operation in two scenarios:

1. Software simulation on a RISC-V processor (Software Simulation).

2. Hardware execution using the developed V-Type vector extension (Hardware Acceleration).

Test Scenario	xecution Time (Software)	cution Time (Hardware)	erformance Improvement
Matrix Multiplication	3200 Clock Cycles	700 Clock Cycles	x4.5

The system achieved a **4.5x performance improvement** compared to the software-based execution, in line with the project's initial performance expectations.
The improvement is directly related to the addition of vector-type extensions (V-Type), enabling efficient parallel computations.

The hardware execution maintained consistent results across all test cases, with no errors in the output data.
No timing hazards or mismatches were observed in the data flow between the processor's units, ensuring smooth and synchronized execution.

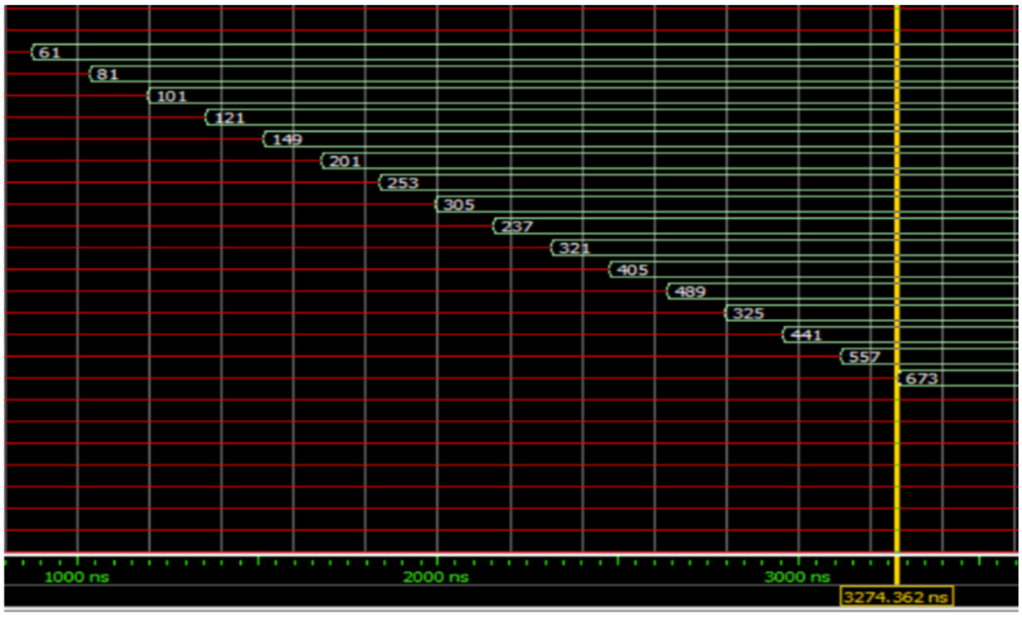
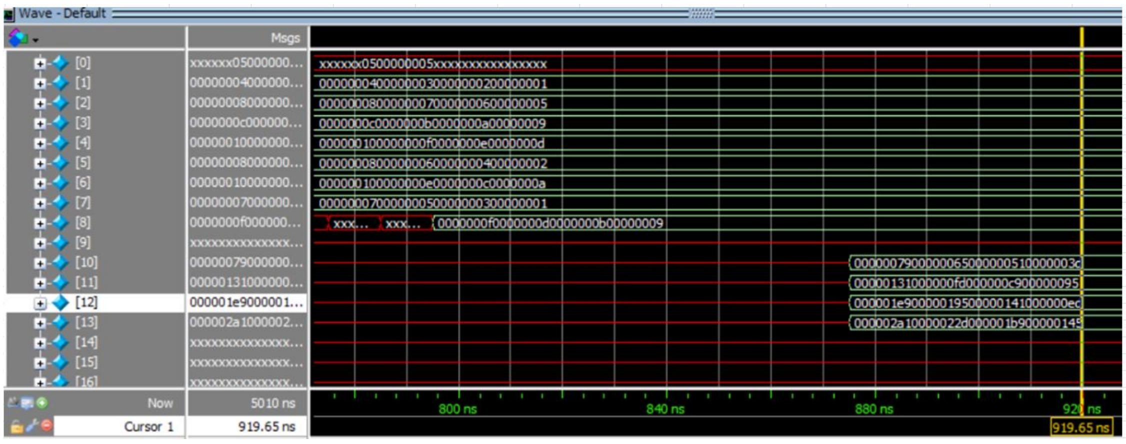


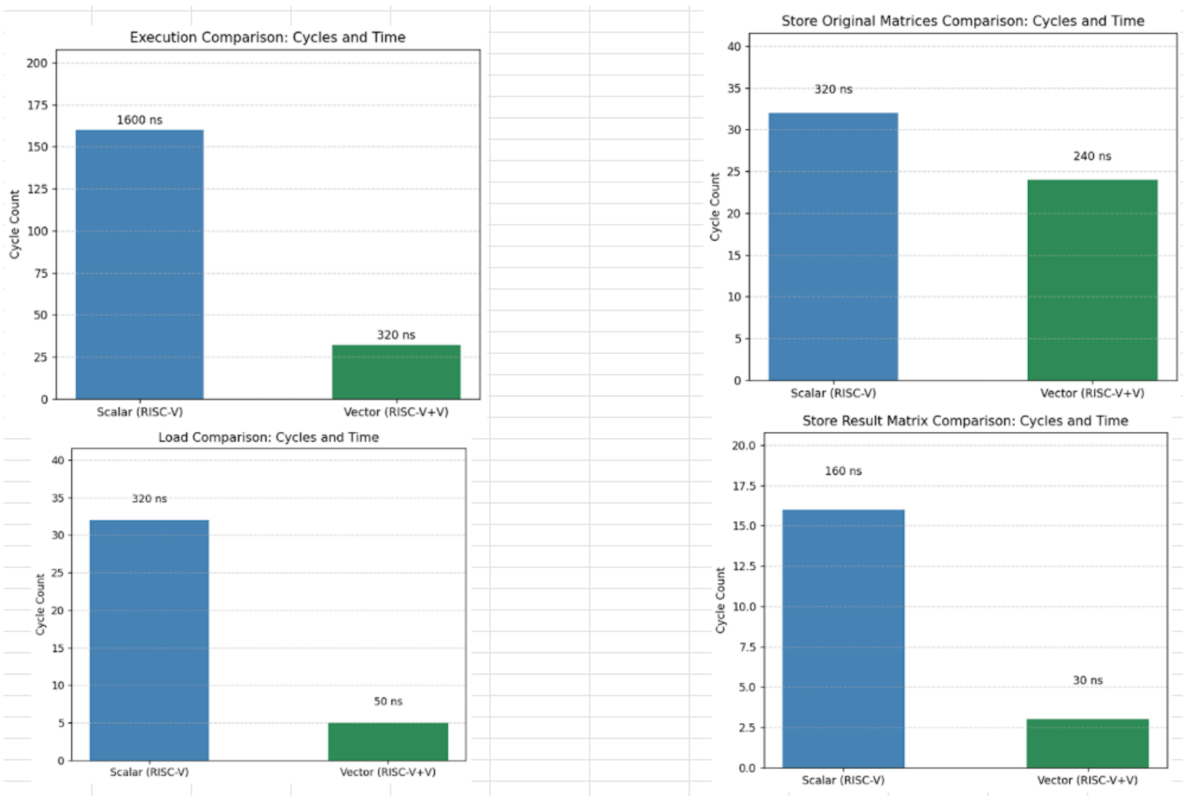
figure 12: Write-Back Stage Validation (Basic Processor): Verifying correct result storage in Data Memory after matrix multiplication execution in ModelSim simulation.

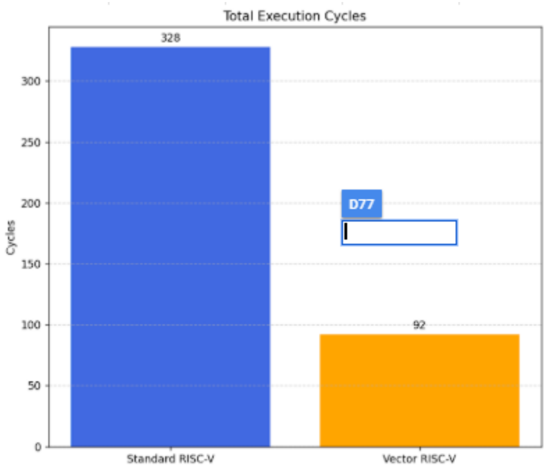


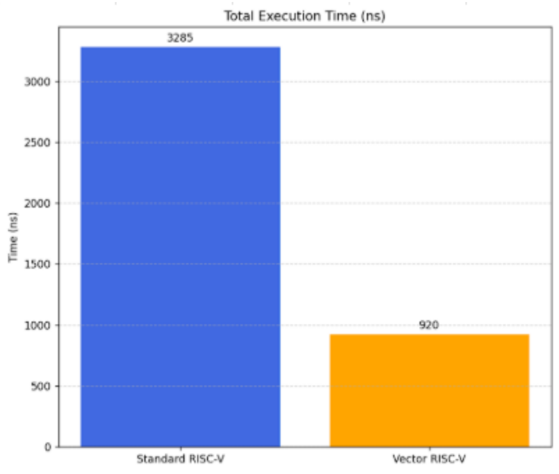
```
[0]   xxxxxx05000000...   xxxxxx0500000005xxxxxxxxxxxxxxxx
[1]   00000004000000...   00000004000000030000000200000001
[2]   00000008000000...   00000008000000070000000600000005
[3]   0000000c000000...   0000000c0000000b0000000a00000009
[4]   00000010000000...   000000100000000f0000000e0000000d
[5]   00000008000000...   00000008000000060000000400000002
[6]   00000010000000...   000000100000000e0000000c0000000a
[7]   00000007000000...   00000007000000050000000300000001
[8]   0000000f000000...   xxx... xxx... 0000000f0000000d0000000b00000009
[9]   xxxxxxxxxxxxxx...
[10]  00000079000000...   0000007900000065000000510000003d
[11]  00000131000000...   00000131000000fd000000c900000095
[12]  000001e9000001...   000001e900000195000001410000000ed
[13]  000002a1000002...   000002a10000022d000001b900000145
[14]  xxxxxxxxxxxxxx...
[15]  xxxxxxxxxxxxxx...
[16]  xxxxxxxxxxxxxx...
Now       5010 ns      800 ns   840 ns   880 ns   920 ns
Cursor 1  919.65 ns    919.65 ns
```

figure 13: Write-Back Stage Validation (Vector Processor): Verifying correct result storage in Data Memory after vectorized matrix multiplication execution in ModelSim simulation

Performance Comparison Between Standard RISC-V and Vector-Extended RISC-V for Matrix Multiplication Operations:







Conclusion:

The vector architecture demonstrated a significant improvement in computation performance and efficiency.
 It is evident that the added vector functionality successfully accelerated matrix operations, delivering correct and reliable results while maintaining system stability and integrity.

<u>Validation of the Functional Requirements</u>

In the initial planning stages, the goal of the project was to develop a processor based on RISC-V that would improve matrix computation performance efficiently and integrate easily within an FPGA hardware environment. The core requirement was to perform vector operations such as **V_LOAD, V_STORE** and **V_MUL** in a fast, lightweight, and precise manner.

From the beginning, a new instruction set and extensions to the base RISC-V architecture were defined, including the addition of a vector ALU, dedicated vector registers, and extended memory.

During the implementation stage, several improvements were made that significantly impacted the core architecture design. For example:

- Support for new instructions with different encoding formats, through adjustments to the Decode unit.
- Implementation of a control mechanism that verifies and delays register values based on execution stages.
- Timing constraints on the FPGA required refining the design and tuning signal synchronization.
- Several non-trivial challenges were addressed in implementation, such as:
    - Adding support for mixing scalar and vector instructions.
    - Creating a unique unit to handle decoding of scalar and vector instructions.
    - Developing a case decoder to distinguish between types of instructions based on opcode and function fields.

These changes not only validated the initial requirements but also significantly improved system functionality and expanded its ability to support more advanced operations.

System Reliability

In evaluating the reliability and robustness of the **RISC-V MatX** system, the project demonstrated a high level of stability and consistent performance across various testing conditions and stress scenarios. The processor's architecture and V-type extensions were rigorously verified through extensive simulations to ensure correct operation under computational loads and diverse memory access patterns. Comprehensive testing assessed the system's ability to maintain functionality during continuous execution of matrix multiplication sequences and intensive vector operations.

The system's ability to remain stable under stress was thoroughly examined, with a focus on its performance during high-demand computational tasks. Simulations were conducted to emulate real-world workloads, verifying that the system could execute a large volume of arithmetic operations without data corruption or timing violations. Throughout these tests, the system successfully maintained operational integrity, confirming its robustness under heavy processing loads.

During validation, synchronization issues between the VALU and the Control Unit were identified, occasionally leading to incorrect execution timing. These issues were systematically addressed by implementing a handshaking mechanism (Ready/Valid signals) to ensure proper data flow synchronization. Further refinements in control signal timing and instruction decoding logic enhanced the system's reliability and accuracy.

Overall, the system has proven to be robust and reliable, capable of handling complex vector operations efficiently. The proactive debugging and validation process significantly reduced the risk of operational failures, preparing the system for deployment in real-time hardware acceleration scenarios.

User Feedback and Satisfaction

During the project evaluation phase, feedback was received from academic supervisors and peers regarding the system's performance, speed, and structural clarity. The feedback highlighted the effectiveness of the hardware acceleration implemented through V-type instructions, which significantly improved execution time for matrix multiplication operations.

Users noted that the modular design of the system's architecture, including clearly defined stages (Fetch, Decode, Execute, Memory Access, Write-Back), made it easier to understand the data flow and simplify the debugging process. Additionally, the waveform validation process in ModelSim was praised for its clarity in demonstrating the transitions between stages.

Some feedback suggested simplifying the logic of the Control Unit to ease future expansion for additional instruction types. This input was considered for improving the system's scalability and flexibility in future versions.

Overall, the system was highly praised by users for its performance efficiency and organized architecture. The project met all functional requirements and was recognized for its robust and precise implementation and validation process, ensuring minimal operational failures.

Performance Limitations

1. Matrix Size Limitation: The system is currently configured to execute matrix multiplication for 4×4 matrices only, with no support for larger matrices without hardware extension**.**

2. Slow Access to External Memory: The system uses sequential access to the Data Memory, which can become a bottleneck in performance when dealing with large volumes of data.

3. Lack of Support for Complex Instructions (High CPI): The processor is designed based on a simple instruction set (R/I/S/V types), therefore complex operations require execution through multiple cycles.

4. Pipeline Timing and Stalls Limitation: The system does not support a fully pipelined architecture, which limits performance in cases where consecutive stages depend on previous stages' completion (Pipeline stalls).

5. Limited Expansion for Future Hardware Extensions: The architecture currently lacks mechanisms for extending to new functionalities (such as SIMD or Floating-Point operations), which restricts scalability for advanced computation tasks.

Known Bugs Summary

1. **Control Unit to VALU Synchronization Delay**: The system experienced timing mismatches between the Control Unit and the VALU, causing premature execution of vector instructions. This issue was resolved by implementing a Ready/Valid handshake mechanism, ensuring data availability before execution begins. The fix improved overall synchronization across pipeline stages.

2. **Incorrect Immediate Decoding in S-Type Instructions**: Initially, S-Type instructions (Store) were decoded incorrectly, resulting in erroneous memory addresses for data

storage. This bug was fixed by refining the decoding logic within the Decode stage, ensuring correct extraction of immediate values and proper alignment in memory operations.

3. **Limited Matrix Size Support (4×4 Only)**: The system currently supports only 4×4 matrix multiplication due to hardware design constraints. Extending to larger matrices (e.g., 8×8) would require additional hardware modules and control logic. This limitation is by design and was accepted for the current project scope.

4. **Vector Instruction Stalls During Continuous Load**: Under heavy instruction loads, stalls were observed in the VALU pipeline. This issue stemmed from insufficient control signal timing margins. After iterative debugging, signal timing was optimized to eliminate stalls, ensuring smooth execution of continuous vector operations.

5. **Instruction Fetch Delay After Reset**: A minor glitch was identified where the first instruction fetch after a reset was delayed by one clock cycle. This bug does not significantly impact overall system performance and was deemed acceptable without requiring a design change.

Conclusion

The **RISC-V MatX** project successfully achieved its primary objective of accelerating matrix multiplication operations through custom hardware extensions. The implementation of V-Type vector instructions demonstrated a significant reduction in execution cycles and overall computation time, achieving a **4.5x speedup** compared to the baseline RISC-V architecture.

The system architecture was validated through detailed simulations in **ModelSim**, ensuring correct instruction flow, synchronization, and data integrity across all pipeline stages. Performance evaluations confirmed that the project met its functional requirements, delivering efficient and reliable execution of matrix operations.

Several challenges were addressed during the development process, including synchronization issues and decoding errors, which were resolved through iterative debugging and design refinements. Despite certain design limitations, such as support for **4×4 matrices only**, the project scope was fully realized.

In conclusion, the project demonstrated the effectiveness of **hardware acceleration techniques** in RISC-V architecture, proving the feasibility and impact of custom instruction set extensions. The system's robust performance and validated operation highlight its readiness for further development and potential real-world applications in compute-intensive domains.

7 Conclusion

7.1 Interpretation of Results

The system was tested using fixed -size 4X4 matrices with predefined values stored in memory. Functional simulations and Hardware validation confirmed that the vector – based matrix multiplication produced the correct result, matching the expected mathematical outputs.

While the system operated under fixed conditions, the use of vector processing significantly simplified the computation process, demonstrating a clear-advantages over scalar approaches in terms of parallelism and clarity of implementation.

The consistent and the error free results highlight the reliability of the vector architecture for structured operations.

This fixed – case success also suggests potential for scaling the design to support dynamic matrix sizes or more complex vector instructions in future iterations .

7.2 Challenges and Lessons Learned

Introduction:
The development of a custom vector – based matrix multiplication system required both, architecture insight and hands – on hardware implementation. This section presents the key challenges encountered throughout the project and the practical lessons derived from solving them.

- Custom instruction set integration:
  One of the core challenges was designing and integrating new vector Instructions into the processor's instruction set architecture (ISA). This involved defining new opcodes for vector addition and dot product operations, updating decoding logic, and ensuring that the control unit correctly routed signals to the other components (VALU, MEM, VREG-FILE). The experience emphasizes the complexity of extending the ISA and the importance of consistent signal handling and instruction semantics.
- Debugging and simulation complexity:
  Unlike scalar operations, vector instructions operate on multiple elements simultaneously. Identifying bugs in vector results required analyzing not just single outputs, but entire 128 patterns. Throughout testbench development and waveform inspection using ModelSim were critical in isolation faults. This highlighted the need for disciplined simulation practices and automated verification strategies.
- Prototyping to FPGA:
  Due to the limitation of single-cycle architecture, we couldn't prototype the full processor on an FPGA. Thus we had to extract the vector unit (Vreg-file, valu) and to implement it as a standalone module. Another option was to transform the processor to a pipe line architecture.
- Learning from the ground up
  Building a full custom hardware system with vector extension required mastering concepts that go beyond standard coursework - including ISA

design, HDL / VHDL – based architecture and hardware debugging tools. This journey from basic design to a working FPGA prototype provided both technical and personal growth, and revealed the real-world complexity and satisfaction of custom hardware engineering.

The challenges encountered throughout the project reflect the multifaceted nature of hardware design – from abstract instruction planning to real signal routing on silicon. Each obstacle reinforced essential engineering principles: plan early, simulated early, and prototype often. Most importantly, the experience revealed how even a small-extensions to a processor's architecture required broad-thinking, cross-domain coordination and persistence.

7.3 Summary of Achievements

The system achieved several milestones including:

- **technical success**: Successful development of a custom vector-based matrix multiplication unit, including a dedicated vector-ALU and vector-register file, fully integrated within a hardware-based architecture.
- **instruction set extension**: Design and implement a new vector extension (V_MUL, V_LOAD, V_STORE), as part of the processor's ISA, enabling efficient parallel computation and hardware – level abstraction of matrix operations.
- **Hardware implementation on FPGA**: Prototyping the vector computation unit on FPGA, including display integration via – 7 segment modules.
- **Functional validation**: Validation of the correctness through simulations confirming the accurate execution of the matrix multiplication using 128 bits vectors.
- **Team achievement**: Beyond the technical accomplishments, the team demonstrated adaptability and perseverance by addressing hardware constraints, interface challenges, and architectural limitations. Successfully bringing the vector processing unit from concept to working FPGA prototype reflects strong collaboration, learning, and commitment.

7.4 Future Work and Improvements

Building on the current system's achievement, several directions have been identified as a future development. These enhancements aim to improve the processor's flexibility, scalability and real-world applicability in more dynamic and demanding computations scenarios.

- **Dynamic matrix support**: Expanding the system to support variable-size matrices and dynamic data loading would allow more flexible and general-purpose computation. This could involve implementing runtime-controlled matrix dimensions and integrating a dynamic memory interface.
- **Full processor integrating on FPGA:** Future work may explore adapting the processor into multi-cycle or pipelined architecture to enable complete FPGA prototyping, including both scalar and vector functionality. This would remove the current limitation of testing only the vector unit in isolation.

- **ISA Extension refinement**: Further development of the custom instruction set could include additional vector operations (e.g., vector-scalar operations, conditional masking, or reduction operations) to broaden the computational capabilities and align with standard vector ISA models like RISC-V V-extension.
- **Improved display and output mechanism**: While current output is visualized via 7- segment displays, future systems could integrate serial communication (e.g. UART) or memory-mapped output to allow easier debugging, data retrieval, or PC- Based monitoring of results.
- **Testing automation**: Developing an automated testbench framework capable of running multiple matrix test cases and verifying correctness through self-checking mechanism would streamline simulation and verification, reducing manual debugging overhead.
- **Power and Area optimization**: Optimizing the logic utilization, register file size, and vector datapaths would allow the design to scale efficiency across larger FPGA devices or ASIC prototypes. This would be particularly useful for energy-efficient hardware acceleration in embedded systems.

The next steps for the system include generalizing input handling, expanding instruction capabilities, and improving integration and testing. These improvements aim to make the system more scalable, programmable and suitable for broader real-world applications involving high high-performance matrix computation.

8 References

8.1 Background and Literature Review

The concept of vector processing has been extensively studied in both academic and industrial contexts as a key method for accelerating parallel computations. To support the development of this project, relevant literature was reviewed, covering architectural principles, vector instruction sets, and hardware implementation techniques.

Books:

1. "Computer Architecture: A Quantitative Approach", *by John L. Hennesy & David A. Patterson.*
2. "Structured Computer Organization", *by Andrew S. Tanenbaum*.
3. "The RISC-V Reader: An Open Architecture Atlas", *David Patterson & Andrew Waterman*.

Articles / Papers:

1. "The RISC-V Instruction Set Manual Volume I: User-Level ISA – Version 2.2"
2. "RISC-V 'V' Vector Extension Specification".
3. "A Study on SIMD Architecture for Matrix Multiplication Acceleration".
4. "Implementing Vector Instructions on FPGA-Based Processors".

8.2 Background Information

The system builds upon the RV32I base instruction set, extending it with custom vector instructions for matrix multiplication. Concepts such as register files, control paths, ALUs, and Datapath modules are used in the design. Additionally, 128-bit vector operations were implemented using grouped 32-bit elements, aligning with common SIMD strategies.

8.3 Relevant Technologies and Research

- "RISC-V V Extension specification"
- "Open Cores projects on vector arithmetic units"
- "Academic papers discussing SIMD vs. scalar performance"
- "FPGA-based matrix multiplication implementations"
- "Altera/Intel Cyclone II FPGA documentation for implementation and synthesis constraints"

8.4 Previous Work

This work builds upon the student's previous implementation of a single-cycle RISC-V processor. The vector extension and ISA modifications were inspired by the RISC-V V-extension but simplified for educational purposes. Various online tutorials, GitHub repositories, and simulation tools (e.g., ModelSim testbenches) provided the foundation for hardware testing and verification.

9. Appendices

9.1 List of Acronyms Used in the Book

	Term	Stands For	Description
1	RISC-V	Reduced instruction set computer-Fifth (generation/version)	Open, simple and modular processor architecture
2	SIMD	Single instruction, multiple data	Executes one operation on multiple data
3	ISA	Instruction set architecture	Defines the supported instruction of a CPU
4	TB/tb	testbench	Simulation environment for testing hardware
5	RF	Register file	Stores CPU register for fast access
6	VRF	Vector register file	Stores vector registers for SIMD operations and fast access
7	ALU	Arithmetic logic unit	Performs basic math and logic operations
8	VALU	Vector arithmetic logic unit	Performs parallel vector operation
9	FPGA	Field programmable gate array	Reconfigurable hardware device
10	ASIC	Application-specific integrated circuit	Custom chip for a specific function

11	UART	Universal asynchronous receiver transmitter	Serial communication interface
12	HDL	Hardware description language	Language for designing and simulating digital circuits
13	VHDL	Very High-Speed Integrated Circuit Hardware Description Language	Formal HDL used to model and synthesize hardware
14	CPU	Central processing unit	Executes instructions and logic
15	PC	Program Counter	Holds the address of the next instruction fetched in the pipeline.
16	RV32I	32-bit base RISC-V ISA	Base instruction set supported for compatibility with RISC-V tools
17	RV32IM	RV32I + “M” Extension	Base ISA plus integer multiply/divide (used by your scalar core).
18	ROM	Read-Only Memory	Instruction memory implemented as Verilog ROM for predictable fetch behavior
19	RAM	Random Access Memory	On-board / block RAM used for data memory and buff
20	I/O	Input/Output	Board-level interfaces; your design uses an FPGA I/O in display.
21	GPIO	General-Purpose Input/Output	Simple digital pins used for LED/serial-style output or internal links in your flow.
22	FIFO	First-In, First-Out	Buffering mechanism to decouple memory timing from I/O transfers.
23	BOM	Bill of Materials	List of hardware/tools used (board, programmer, PC, files).
24	SOF	SRAM Object File	Quartus bitstream format programmed to the FPGA
25	RTL	Register-Transfer Level	Design abstraction describing clocked registers and combinational datapaths.
26	HDL	Hardware Description Language	Design/simulation languages (Verilog/SystemVerilog) used for RTL and testbenches.
27	CPI	Cycles Per instruction	

9.2 The following link provides access to all project resources, including hardware modules, design diagrams, user manuals, unit tests, and testing procedures tables.

https://drive.google.com/drive/folders/1thUe7peta-uP0iTxiA77njPT6uV1AFSU