FPGA-Based Hardware-Accelerated Matrix Multiplication on RISC-V Using Systolic Array Integration

Anirudh Mittal, Arjun Mallya, Shah Tirth
Department of Electrical Engineering
Indian Institute of Technology Gandhinagar
Gandhinagar, Gujarat, India

Mentored by: Prof. Joycee M. Mekie
Department of Electrical Engineering
Indian Institute of Technology Gandhinagar
Gandhinagar, Gujarat, India

***Abstract*—Matrix multiplication is a core computational primitive in signal processing, scientific computing, and machine learning, yet its inherent $\mathcal{O}(n^3)$ complexity makes it a major performance bottleneck on sequential processors. This report presents a hardware-accelerated implementation of matrix multiplication through a custom instruction extension to the RV32IM RISC-V core. A dedicated systolic array accelerator is integrated into the five-stage RISC-V pipeline [1] via a custom `matmul` instruction, with modifications made to both the GNU RISC-V toolchain and the Spike simulator [2] for software simulation. The Verilog-based hardware architecture incorporates a wrapper for operand control, a counter for profiling, and UART for runtime validation. The complete system was prototyped on a Xilinx Nexys4 DDR FPGA [3], and performance profiling via waveform analysis and post-implementation timing reports demonstrated a substantial reduction in execution cycles, reducing the effective computational complexity significantly. This work underscores the potential of custom ISA extensions and domain-specific accelerators [4] in advancing the performance of open-source RISC-V-based embedded computing platforms.**

***Index Terms*—RISC-V, FPGA, Matrix Multiplication, Custom Instruction, Systolic Array, Hardware Acceleration, Spike Simulator, GNU Toolchain**

I. Introduction

Traditional software-based implementations of matrix multiplication on sequential processors struggle to meet real-time performance requirements. To address this problem, this work proposes a hardware-accelerated solution by extending the **open-source RISC-V architecture**. A custom `matmul` instruction is introduced into a **5-stage pipelined RV32IM core**, designed to offload matrix multiplication to a dedicated **systolic array accelerator** [5]. This accelerator is aligned to the processor pipeline via a Verilog-based wrapper, allowing seamless data movement and computation.

To support this hardware-software co-design, the **RISC-V GNU toolchain** was modified to encode the new instruction, enabling developers to invoke `matmul` directly from high-level C code. Parallelly, the **Spike ISA Simulator** [2] was extended to include a functional model of the custom instruction, providing a complete testflow for cycle-accurate simulation and verification. The entire system was synthesized and implemented on a **Xilinx Nexys4 DDR** [3], featuring **UART-based output** for real-time debugging and a **performance counter** for execution profiling. Experimental results confirm a significant reduction in execution time, due to pipelining and parallel dataflow, improving throughput for all matrix sizes.

II. Background and Related Work

A. The RISC-V ISA

RISC-V is a modern, open-standard instruction set architecture (ISA) [6] developed to support research, education, and industrial implementations. Its design follows reduced instruction set computing (RISC) principles, emphasizing simplicity, regularity, and extensibility. The base ISA: **RV32I**, includes essential integer arithmetic and control instructions operating on **32-bit registers**. Additional standard extensions such as the 'M' extension for integer multiplication and division, forming **RV32IM**,allow tailoring the ISA to various application domains. The RISC-V ISA has dedicated opcode space for non-standard instructions, enabling application-specific accelerators to be seamlessly integrated into the processor pipeline.

B. Instruction Formats in RISC-V

RISC-V supports a compact set of instruction formats categorized primarily as: **R-type, I-type, S-type, B-type, U-type, and J-type**. Each format has a pre-decided bit layout for opcode, registers, and immediate fields [6]. The **R-type format** is used for register-register operations and consists of the fields: `opcode` [6:0], `rd` [11:7], `funct3` [14:12], `rs1` [19:15], `rs2` [24:20], and `funct7` [31:25]. The proposed custom instruction follows this format and is encoded using the reserved `CUSTOM-0` opcode space (7'b0001011).

C. Custom Matrix Multiplication Instruction

To enhance the computational speed of matrix multiplication , a custom `matmul` instruction was introduced. The instruction syntax is:

```
matmul rd, rs1, rs2
```

where `rs1` and `rs2` hold base addresses of input matrices A and B, and `rd` holds the destination base address for the result matrix C. The instruction is detected in the decode stage using:

- **Opcode**: `0001011` (`CUSTOM-0`)
- **funct3**: `000`
- **funct7**: `0000000`

Upon detection, the processor pipeline is stalled, and control is transferred to the `Matmul Controller` module. This controller fetches operand matrices from memory, streams them into a **parallel systolic array** [5] for computation, and writes back the results, enhancing performance while maintaining the instruction-level abstraction.

D. *Systolic Arrays in Hardware Acceleration*

Systolic arrays are a class of spatial dataflow architectures composed of regularly interconnected processing elements (PEs). Each PE performs a local **multiply-accumulate (MAC)** operation and passes partial results to neighbors in a pipelined fashion. Such architectures are well-known for their high throughput and low memory bandwidth requirements, particularly in matrix-heavy domains like image processing and neural networks.

In this implementation, the systolic array is accessed via a Verilog-based wrapper, invoked by the custom `matmul` instruction. Input matrices are passed into the systolic array from orthogonal directions, processed across multiple clock cycles, and the output matrix is written back to memory. Unlike many accelerator approaches that rely on **memory-mapped I/O** or external interfaces, this system embeds the accelerator directly into the **RISC-V execution pipeline**, ensuring low latency.

E. *Toolchain and Software Integration*

To validate the custom instruction, the **RISC-V GNU Toolchain** was modified, adding opcode and instruction definitions. Additionally, the **Spike ISA Simulator** [2] was extended with a functional C model of the new instruction, allowing verification of the correctness of generated binaries before deploying them to hardware. Test programs were written in C, using inline assembly to invoke the `matmul` instruction, and compiled using the modified toolchain. The final instruction stream was exported as a hexadecimal file for Vivado testing, forming a complete software-hardware co-design workflow.

III. System Architecture

The overall system architecture is illustrated in 1.

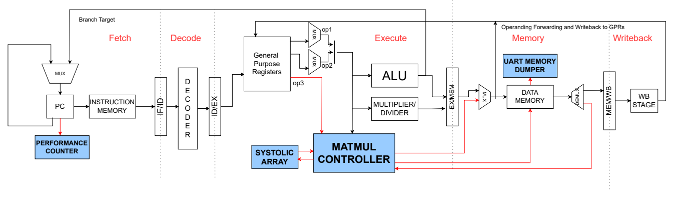


Fig. 1. Implemented System Architecture and Systolic Array

A. *Baseline Core and Pipeline Overview*

The base processor comprises five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). It includes a register file, ALU, control logic, and data/instruction memory interfaces. This architecture serves as the foundation upon which custom hardware modules are integrated.

B. *Hardware Accelerator Integration*

To enable efficient matrix multiplication, the processor is augmented with three key hardware modules:

1) **Matmul Wrapper:** A control module containing a **Finite State Machine (FSM)** that activates upon detection of the `matmul` instruction in the **Decode stage**. The wrapper stalls the pipeline, initiates matrix operand fetches from Data Memory, coordinates computation with the systolic array, and writes the results back to memory.
2) **Systolic Array:** A two-dimensional grid of **Multiply-Accumulate (MAC)** units that operates in a pipelined and data-synchronous fashion. It accepts streamed matrix operands and produces matrix products, by performing one MAC operation per clock cycle, thereby exploiting parallelism and increasing throughput.
3) **Memory Access Arbiter:** This module arbitrates access to the shared Data Memory between the processor's **Load/Store Unit** and the **Matmul Wrapper**. Under normal execution, memory access is granted to the core. When the `matmul` instruction is active, the MUX redirects access control to the wrapper, ensuring exclusive access for matrix operand transfer and result storage. A DEMUX is used to provide the data to either the Load/Store Unit or the Matmul wrapper, depending on if the matmul instruction is active.

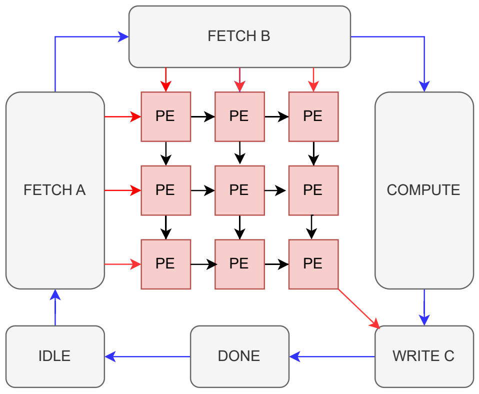


Fig. 2. Systolic Array Integration with Matmul Controller

C. *Instruction-Driven Switching Logic*

The integration is instruction-driven and requires minimal software intervention. During standard instruction execution, the datapath and control flow remain unchanged. Upon decoding the custom `matmul` opcode (7'b0001011), the Decode stage asserts a control signal that:

- Stalls the program counter and all downstream pipeline registers.
- Signals the Memory Arbiter to redirect Data Memory access to the Matmul Wrapper.
- Triggers the FSM within the wrapper to begin matrix fetch and computation operations.

Once the systolic array finishes computation and the wrapper writes back the result matrix, a `done` signal is raised. This signal re-enables the processor pipeline and resumes normal program execution.

D. Modularity and Extensibility

The modular design ensures that the accelerator is non-intrusive to the existing core architecture. It does not require changes to the ALU or core pipeline logic, other than a decode hook for recognizing the `matmul` instruction. This approach preserves the integrity of the RV32IM design while enabling scalable domain-specific acceleration.

IV. Methodology and Implementation

This project was made through a systematic, three-phase methodology encompassing software toolchain modification, custom hardware accelerator design, and physical deployment on an FPGA platform.

A. Software Toolchain Extension and ISA Simulation

To integrate the custom `matmul` instruction, we first extended the RISC-V software for correctness and compatibility.

- **GNU Toolchain:** The relevant header files in the GNU Binutils were modified to include the matmul mnemonic. The instruction was defined in R-type format using a reserved `CUSTOM-0` opcode **(7'b0001011)**, allowing for seamless decoding in downstream hardware stages.
- **Spike Simulator:** A functional model of the `matmul` instruction was implemented in C within the **Spike ISA simulator**. This served as a software reference model for validating the semantic correctness of compiled C program.
- **C-to-Hex Toolchain Flow:** Test programs were written in C and `inline assembly` was used(via `asm volatile`) to invoke the custom instruction. These were compiled using the modified `riscv-gcc`, producing ELF binaries. The binaries were converted into Verilog-compatible '.hex' files using `objcopy` and `hexdump`, to initialize the instruction memory on the FPGA for testing.

B. Baseline Software Implementation of Matrix Multiplication

Before introducing hardware acceleration, the performance of a baseline matrix multiplication implemented entirely in software using the standard three-nested loop approach was analyzed. This implementation follows the pseudocode below:

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)
            C[i][j] += A[i][k] * B[k][j];
```

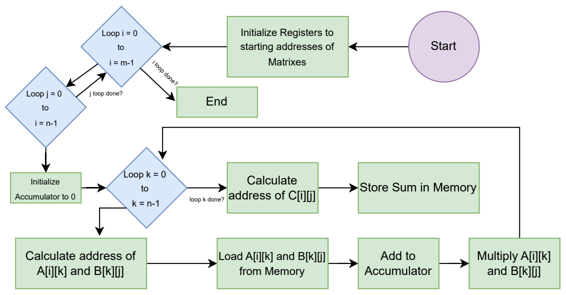

Start
Initialize Registers to starting addresses of Matrixes
Loop i = 0 to i = m-1
i loop done?
End
Loop j = 0 to i = n-1
j loop done?
Initialize Accumulator to 0
Loop k = 0 to k = n-1
loop k done?
Calculate address of C[i][j]
Store Sum in Memory
Calculate address of A[i][k] and B[k][j]
Load A[i][k] and B[k][j] from Memory
Add to Accumulator
Multiply A[i][k] and B[k][j]

Fig. 3. Flow diagram of mul-loop based multiplication on RISCV Core

This algorithm performs $\mathcal{O}(n^3)$ multiply-accumulate operations, which map to multiple standard `MUL` and `ADD` instructions in the RV32IM instruction set. When executed sequentially on the 5-stage pipelined core, this incurs a high latency due to frequent data memory accesses, register file updates, and limited instruction-level parallelism. This baseline provided both functional validation and a quantitative reference point for evaluating the effectiveness of the custom `matmul` instruction.

C. Hardware Accelerator Design in Verilog

The custom matrix multiplication accelerator consists of three primary Verilog modules:

- **Matmul Wrapper:** A finite-state machine (FSM) that coordinates data movement and control signaling.
- **Systolic Array:** A 2D grid of processing elements that perform parallel MAC operations.
- **Memory Arbiter:** Allows the wrapper to gain temporary access to the shared data memory for operand fetch and result writeback.

This hardware bypasses the main ALU and leverages localized, parallel computation, thereby drastically reducing the number of clock cycles required for large matrix sizes.

D. FPGA Synthesis, Timing Closure, and Verification

The complete system, including the core, custom instruction logic, memory units, and peripherals, was synthesized using the **Vivado Design Suite** [7]. Initial synthesis revealed timing violations due to excessive logic depth at the default 100 MHz clock. These violations were resolved by:

- Integrating the **Vivado Clocking Wizard IP** to derive a 67 MHz clock.
- Reconfiguring the synthesis strategy for optimized routing and register balancing.

Post-implementation static timing analysis confirmed that all setup and hold constraints were met at 67 MHz, enabling reliable and real-time operation.

TABLE I
PERFORMANCE METRICS (WITH `MATMUL` INSTRUCTION)

Metric	Value
Number of Clock Cycles	123
Clock Period	15 ns
Total Wall Clock Time	1845 ns

TABLE II
PERFORMANCE METRICS (WITHOUT `MATMUL` INSTRUCTION)

Metric	Value
Number of Clock Cycles	785
Clock Period	13 ns
Total Wall Clock Time	10 215 ns

E. *Runtime Verification and Output Capture via UART*

For physical runtime validation, we integrated a memory-mapped **UART module** configured at **9600 baud**. A **performance counter** was used to measure the execution latency. Upon completing the `matmul` operation, the contents of result matrix C and the final cycle count were transmitted over UART to a host PC terminal. This pipeline offered a seamless method for verifying both functional correctness and hardware-level performance.

V. EXPERIMENTAL RESULTS AND ANALYSIS

The proposed design was evaluated based on functional correctness, wall-clock performance, and power efficiency. [3] using synthesized Verilog designs.

A. *Functional Verification*

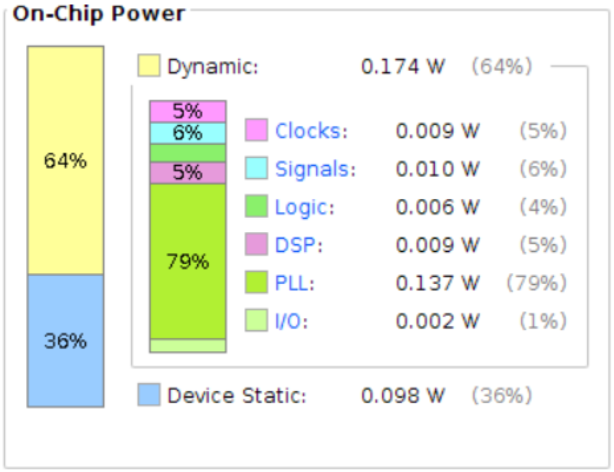
To ensure correctness, a baseline matrix multiplication for two 3×3 matrices was implemented both with and without the custom `matmul` instruction. The resulting output matrix from both approaches was transmitted over UART to a host PC and compared with the expected outputs. The outputs matched exactly, validating the accuracy of the hardware accelerator and the correctness of the instruction semantics.

B. *Performance and Power Comparison*

The number of clock cycles and the estimated dynamic on-chip power for both the standard three-loop matrix multiplication method and our hardware-accelerated `matmul` instruction were measured. Tables I and II present the performance metrics for a 3×3 matrix multiplication over frequencies 67 MHz and 77 MHz respectively, representing the performance optimization. For power comparison, a common frequency of 67 MHz was taken, recording on-chip power of 0.272 W (Fig. 4) and 0.25 W (Fig. 5) respectively, with and without matmul.

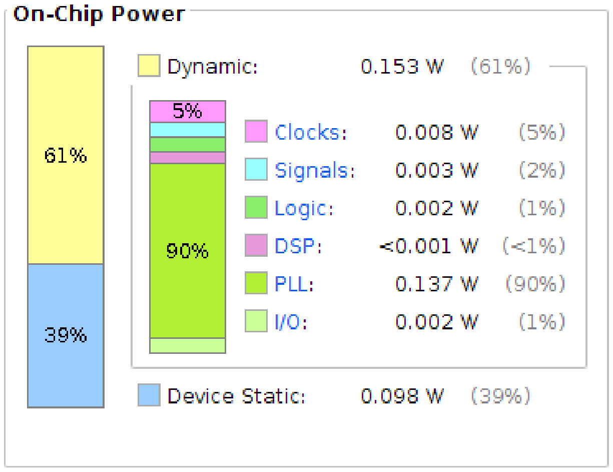
C. *Wall Clock Time Trend Across Matrix Dimensions*

We further evaluated how execution time scales with matrix dimensions using both approaches(Fig. 6). The matrix dimensions are defined as A(M×K), B (K×N), and output C (M×N). The results are presented in Table III.



Dynamic:		0.174 W	(64%)
	Clocks:	0.009 W	(5%)
	Signals:	0.010 W	(6%)
	Logic:	0.006 W	(4%)
	DSP:	0.009 W	(5%)
	PLL:	0.137 W	(79%)
	I/O:	0.002 W	(1%)
Device Static:		0.098 W	(36%)

Fig. 4. Power and Energy Consumption of RISCV with matmul



Dynamic:		0.153 W	(61%)
	Clocks:	0.008 W	(5%)
	Signals:	0.003 W	(2%)
	Logic:	0.002 W	(1%)
	DSP:	<0.001 W	(<1%)
	PLL:	0.137 W	(90%)
	I/O:	0.002 W	(1%)
Device Static:		0.098 W	(39%)

Fig. 5. Power and Energy Consumption of RISCV without matmul

VI. CONCLUSION AND FUTURE WORK

This work presents a complete design, simulation, and hardware realization of a custom `matmul` instruction in a RISC-V-based system, backed by a systolic array accelerator. Our results demonstrate that the proposed approach significantly reduces the effective wall clock time for matrix multiplication while retaining instruction-level abstraction. The successful deployment on an FPGA and verification using UART and cycle counters highlight the practicality of using custom instructions for domain-specific acceleration.

Future work includes integrating the accelerator into a full neural network pipeline and extending the RISC-V toolchain for automatic matmul substitution. We also aim to co-simulate with Spike for hardware-software testing.

VII. ACKNOWLEDGEMENT

This work was supported by the Summer Research Internship Program (SRIP) 2025 , under the guidance of Prof. Joycee M. Mekie and the Micro-electronics and VLSI Research Lab at the Indian Institute of Technology Gandhinagar.

TABLE III
WALL CLOCK TIME (NS) VS. MATRIX DIMENSIONS

M,N,K	Cycles (Matmul)	Time (ns)	Cycles (Loop)	Time (ns)
3,3,3	123	1845	785	10215
3,3,2	95	1330	564	7332
3,3,1	67	938	342	4104
2,2,3	85	1190	344	4472
2,2,2	71	923	266	3458
2,2,1	55	715	164	1968

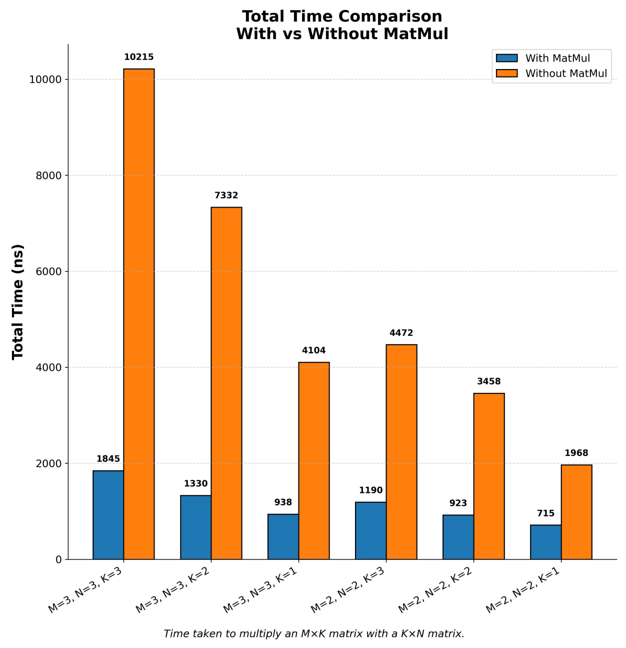


Fig. 6. Bar Graph depicting the time consumption across different matrices

REFERENCES

[1] RISC-V International, “The official risc-v github organization,” GitHub, 2024, accessed on: 2024-07-02. [Online]. Available: https://github.com/riscv

[2] The RISC-V Foundation, “Spike, the RISC-V ISA simulator,” GitHub repository, 2024, accessed on: July 4, 2025. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[3] Digilent Inc., *Nexys 4 DDR Reference Manual*, Digilent Inc., 2024, accessed on: 2024-07-02. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual

[4] G. Erfan, V. Bouac, A. Abdennebi, J.-M. Portal, and L. Torres, “A risc-v-based accelerator for general-purpose in-memory-computing with spintronic memristors,” in *2024 IEEE 25th Latin American Test Symposium (LATS)*. IEEE, 2024, pp. 1–6.

[5] H. T. Kung and C. E. Leiserson, “Systolic arrays (for VLSI),” *Sparse Matrix Proceedings 1978*, pp. 256–282, 1979.

[6] A. Waterman, K. Asanović *et al.*, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, RISC-V Foundation, 2019, document Version 20191213. [Online]. Available: https://riscv.org/technical/specifications/

[7] Xilinx, Inc., “Vivado design suite user guide: Synthesis (UG901),” Xilinx, Inc., Tech. Rep., 2023, v2023.1. [Online]. Available: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis