Execution FSM-Based Processor Subsystem

Shreejal Bhattarai

December 3, 2025

Contents

1 Introduction **2**

2 System Architecture Overview **2**
2.1 Memory Map 2

3 Execution Finite State Machine **3**

4 Data Bus and Addressing Scheme **4**

5 ALU Design **4**
5.1 Matrix ALU 4
5.2 Integer ALU 4

6 Register File **5**

7 Instruction Memory **5**

8 Verification and Testing **5**
8.1 Waveform Results 6

9 Conclusion **6**

1 Introduction

This report presents the design, implementation, and verification of an Execution FSM-based processor subsystem developed using SystemVerilog. The project integrates multiple hardware modules, including a matrix ALU, an integer ALU, instruction memory, a register file, and a shared data bus, into a modular and scalable execution engine driven by a synchronous finite state machine (FSM). The goal of the design is to demonstrate correct instruction sequencing, coordinated data movement, and functional correctness for a small but representative instruction set.

The processor executes a test program that combines matrix operations, integer arithmetic, and conditional branching. The Execution FSM orchestrates instruction fetch, decode, operand access, execution, and writeback while enforcing a single-driver policy on a 256-bit shared data bus. The architecture is intended as a teaching and experimentation platform that can be extended with more advanced features such as pipelining and hazard detection.

2 System Architecture Overview

The processor subsystem is organized as a 256-bit wide architecture with multiple specialized computational and storage modules operating under a unified execution engine. The main modules are:

- Matrix ALU supporting matrix multiplication, scalar multiplication, addition, subtraction, and transposition
- Integer Math ALU for 16-bit integer operations
- Instruction ROM for program storage
- RAM and an internal register block for data and architectural state
- Central Execution Engine implemented as a finite state machine

All modules communicate over a shared 256-bit tri-stated data bus and a 16-bit address bus. The high-order address bits select the target module, and the lower bits provide an internal offset or memory index. Only one module is permitted to drive the data bus at a time, enforced through explicit enable signals and tri-state output logic in each module.

2.1 Memory Map

A 16-bit address bus is used, with the upper nibble encoding the module identifier and the remaining bits identifying an internal offset or register index. The memory map is summarized in Table 1.

Address	**Module**
0000h	Main Memory
1000h	Instruction Memory
2000h	Matrix ALU
3000h	Integer ALU
4000h	Register Block
5000h	Execution Engine

Table 1: Memory Map

The Execution Engine drives the address bus and control signals to select the appropriate module for each phase of instruction processing. Main memory and the register block store data operands and intermediate results, while the matrix and integer ALUs expose control and data registers in their assigned regions of the address space.

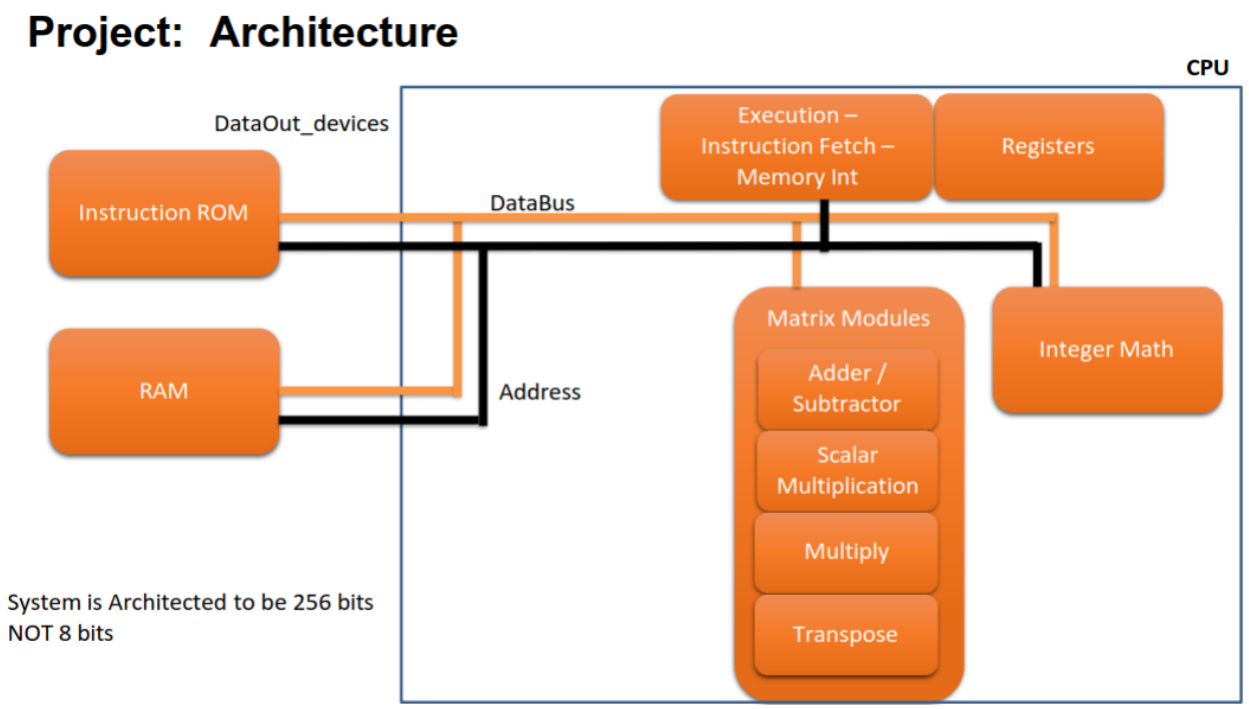

Figure 1: System Architecture

3 Execution Finite State Machine

The core of the processor is the Execution FSM, which sequences all instruction-level operations. The FSM is implemented as a synchronous state machine using SystemVerilog `enum` types for state encoding and `always_ff` and `always_comb` blocks for state update and next-state/control logic. On reset, the FSM enters an initial *Fetch* state with the program counter (PC) set to the start address of the instruction memory.

The main high-level states used by the FSM are:

- **Fetch**: Read the next instruction word from instruction memory using the current PC.
- **Decode**: Decode the opcode, source and destination register specifiers, and any immediate fields.
- **Operand Fetch**: Read operands from the register file or memory as required by the instruction type.
- **Execute**: Perform the requested operation in the integer or matrix ALU.
- **Writeback**: Write the result back to the destination register or memory location.
- **PC Update**: Increment the PC or apply a branch target, then return to **Fetch**.

State transitions occur on each rising clock edge. For straight-line code, the PC is incremented sequentially after each instruction, while for control-flow instructions the FSM evaluates branch conditions and either updates the PC to a branch target or falls through to the next instruction address. Simple stalls are introduced when an operation requires multiple cycles (for example, matrix multiplication), during which the FSM remains in the **Execute** state until the ALU asserts a done signal.

4 Data Bus and Addressing Scheme

The processor modules share a 256-bit tri-stated data bus that carries both instruction and data words. Each module exposes a 256-bit input port and a 256-bit output port, with the output port driven only when an associated enable signal is asserted by the Execution Engine. When not enabled, a module's data outputs are placed in a high-impedance state to avoid contention on the bus.

The 16-bit address bus is interpreted as follows:

- Bits [15:12]: Module identifier (matches the memory map)
- Bits [11:0]: Internal offset, memory index, or register address

Control signals include active-low read (`rd_n`) and write (`wr_n`) lines. During a read cycle, the Execution Engine drives the address and asserts `rd_n`, and exactly one selected module drives the data bus with the requested word. During a write cycle, the Execution Engine drives both address and data, asserts `wr_n`, and the selected module latches incoming data into the specified location.

5 ALU Design

Two primary ALU blocks implement the computational functionality of the processor: a Matrix ALU and an Integer ALU. Both units are memory-mapped and are controlled by writing operands and control fields to their registers, followed by asserting a start signal. Each ALU asserts a done flag when the operation completes, allowing the FSM to proceed.

5.1 Matrix ALU

The Matrix ALU operates on 4×4 matrices with 16-bit elements. Matrices are packed into 256-bit words by concatenating all 16 elements in row-major order, resulting in a natural alignment with the shared bus width. The Matrix ALU supports the following operations:

- Matrix multiplication of two 4×4 matrices
- Scalar multiplication of a 4×4 matrix by a 16-bit scalar
- Matrix addition and subtraction
- Matrix transposition

Internally, the Matrix ALU includes dedicated registers for input matrices A and B, a scalar input, and an output matrix register. Matrix multiplication is implemented as a nested-product datapath that accumulates partial sums across multiple cycles; other operations (addition, subtraction, and transposition) require fewer cycles but still follow a handshaking protocol with the Execution Engine. Operation selection is driven by an opcode field in the ALU control register.

5.2 Integer ALU

The Integer ALU performs basic arithmetic and logical operations on 16-bit operands that are embedded within the 256-bit data words. For scalar instructions, the Execution Engine reads the relevant 16-bit fields from the register file, routes them to the Integer ALU, and then writes the 16-bit result back to the appropriate location in the destination word. The supported operation set includes:

- Addition and subtraction
- Bitwise AND, OR, and XOR
- Comparison operations for branch decisions

The Integer ALU produces status flags such as zero, negative, and greater-than, which are used by the FSM to implement conditional branches. The branch instruction set consists of BNE (branch if not equal), BEQ (branch if equal), BLT (branch if less than), and BGT (branch if greater than). Each branch instruction compares two register operands or a register and an immediate, then conditionally updates the PC.

6 Register File

The register file stores architectural state and intermediate values. It is organized as a set of 256-bit wide registers to match the bus width, allowing entire matrices or aggregates of 16-bit values to be transferred in a single cycle. The register file supports:

- Two synchronous read ports for operand fetch
- One synchronous write port for result writeback
- Register addressing compatible with the instruction encoding format

During the **Operand Fetch** state, the Execution FSM drives the register indices corresponding to the source operands and captures the output values on the next clock edge. During **Writeback**, the FSM asserts the write enable signal and supplies the destination register index and the result data. This separation of read and write phases simplifies timing and avoids write-after-read hazards within a single instruction.

7 Instruction Memory

Instruction memory is implemented as a read-only memory (ROM) preloaded with a test program. Each instruction is encoded in a fixed-width format that includes an opcode field, source and destination register specifiers, and optional immediate fields. Fields are extracted in the **Decode** state to determine which functional units and registers will be accessed.

Instruction fetch is controlled by the PC, which is maintained as an internal register of the Execution Engine. On each **Fetch** state, the address lines are driven with the instruction memory base plus the PC value, and the resulting instruction word is captured on the data bus. The PC is normally incremented by one after each instruction, but for branches and jumps it can be updated with an offset or absolute target computed from instruction fields.

8 Verification and Testing

Verification was performed using a provided top-level testbench and additional self-checking tests. The testbench generates the system clock, applies synchronous resets, and connects behavioral models for instruction and data memory. The test program is assembled into instruction memory and executed by the processor, while waveforms are collected to observe internal state transitions and data values.

The processor is required to execute a sequence of program-level tasks:

1. Matrix addition and scalar multiplication on 4×4 matrices
2. Integer arithmetic operations and comparisons on 16-bit values
3. Conditional branch evaluation using BNE, BEQ, BLT, and BGT instructions
4. Final matrix multiplication combining earlier computed results

For each task, simulation waveforms were examined to verify correct bus transactions, FSM state progression, ALU handshaking, and register updates. Special attention was given to branch instructions to confirm that both taken and not-taken paths resulted in correct PC values and

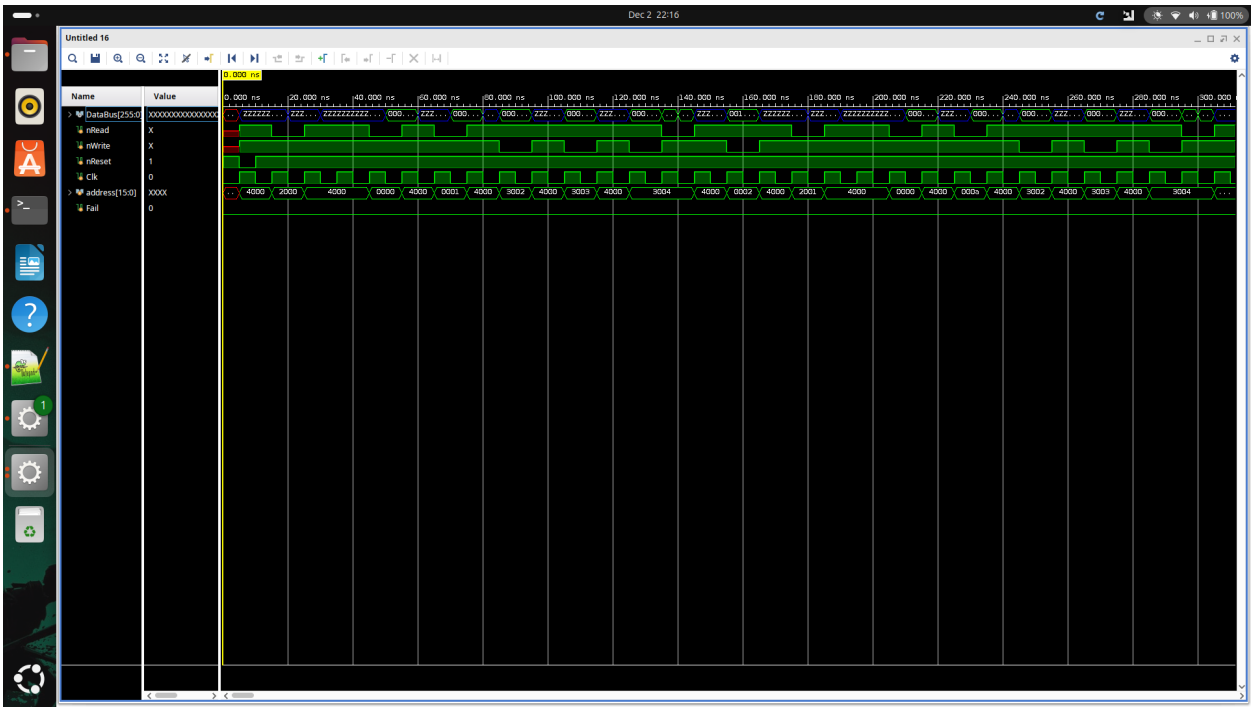

Figure 2: Waveform

control flow. Edge cases such as zero operands, negative operands, and equality conditions were explicitly tested.

8.1 Waveform Results

Waveform captures were collected for representative instructions and for each major program phase. Key observations include:

- Correct sequencing of FSM states from Fetch through Writeback
- Non-overlapping bus drive enables, confirming that only one module drives the data bus at a time
- Proper assertion of ALU done signals and corresponding FSM transitions
- Accurate PC updates on conditional branches and fall-through behavior when branch conditions are not met

These waveforms provide visual confirmation that the Execution Engine correctly coordinates memory accesses, computation, and writeback for both matrix and integer operations.

9 Conclusion

The project successfully implements a modular Execution FSM-based processor subsystem using SystemVerilog. The design demonstrates a scalable bus-oriented architecture, clean separation between control and datapath, and correct functional behavior for a mixed workload that includes matrix operations, integer arithmetic, and conditional branches. The FSM-based Execution Engine reliably sequences instruction fetch, decode, operand access, ALU execution, and writeback while enforcing a single-driver policy on the shared data bus.

Potential future enhancements include the introduction of a pipelined microarchitecture with hazard detection and forwarding, more sophisticated branch handling (such as prediction and delay slots), and a richer instruction set with support for additional matrix and vector operations. Further work could also explore synthesizing the design to FPGA hardware and measuring timing, resource utilization, and performance under different workloads.