FPGA-Based Hardware Acceleration of Matrix Multiplication on RISC-V Using a Custom Systolic Array

Arjun Mallya, Anirudh Mittal, Shah Tirth

July 4, 2025

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 via a custom `matmul` instruction, with modifications made to both the GNU RISC-V toolchain and the Spike simulator 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 [**?**], 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 in advancing the performance of open-source RISC-V-based embedded computing platforms.

Contents

1 Introduction

RISC-V is an **open-source, modular ISA** known for its **simplicity** and **extensibility**. The **RV32IM** variant, featuring a **32-bit integer instruction set** and a classic **five-stage pipeline**, is ideal for **custom instruction integration** and experimentation with **hardware accelerators**.

To overcome the $\mathbf{O}(n^3)$ cost of matrix multiplication on **sequential processors**, we extend the **RISC-V ISA** with a **custom matmul instruction**. This instruction is supported end-to-end—from **software simulation** using a modified **Spike toolchain** to **hardware acceleration** via a **systolic array** integrated on **FPGA**—significantly reducing execution time while preserving pipeline compatibility.

2 Theoretical Background

This section outlines the theoretical foundations essential for understanding the design and implementation of a custom matrix multiplication instruction within the RV32IM RISC-V architecture, including the integration of a hardware accelerator and associated peripheral systems.

2.1 RISC-V Instruction Set Architecture (ISA)

RISC-V is an open-source, modular ISA designed to support a wide range of computing systems. Its clean, extensible architecture has led to widespread academic and industrial adoption. The base ISA is categorized by its register width and supported instruction types. In this project, we use the RV32IM variant, which supports 32-bit integer operations (RV32I) and integer multiplication and division extensions (M).

2.2 General Purpose Registers (GPRs)

RISC-V defines 32 general-purpose registers (x0–x31), each 32 bits wide in the RV32 architecture. Register x0 is hardwired to zero. The standard naming convention for these registers is as follows:

Register	ABI Name	Description
x0	zero	Constant zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–x7	t0–t2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–x11	a0–a1	Function arguments / return values
x12–x17	a2–a7	Function arguments
x18–x27	s2–s11	Saved registers
x28–x31	t3–t6	Temporaries

Table 1: RISC-V General Purpose Registers

2.3 Pipeline Architecture and Instruction Decoding

The RV32IM core used in our design is a classic 5-stage pipeline architecture comprising the following stages:

- **Instruction Fetch (IF):** Retrieves instructions from memory.
- **Instruction Decode (ID):** Decodes the fetched instruction and reads operands from the register file.

- **Execute (EX):** Performs ALU operations or address computation.
- **Memory Access (MEM):** Accesses data memory for load/store instructions.
- **Write Back (WB):** Writes results back to the register file.

In the decode stage, the instruction is identified based on the `opcode`, `funct3`, and `funct7` fields. Custom instructions use opcodes from the reserved `CUSTOM-0` to `CUSTOM-3` space.

2.4 RISC-V Instruction Formats

RISC-V defines several instruction formats to accommodate different types of operations within its reduced instruction set architecture. Each format specifies how the 32-bit instruction word is partitioned into opcode, register identifiers, immediate values, and function fields. The primary formats used in the RV32I base ISA are:

- **R-type:** Used for register-register operations (e.g., `add`, `sub`, `mul`)
- **I-type:** Used for immediate operations and loads (e.g., `addi`, `lw`)
- **S-type:** Used for store instructions (e.g., `sw`)
- **B-type:** Used for conditional branches (e.g., `beq`, `bne`)
- **U-type:** Used for upper immediate operations (e.g., `lui`, `auipc`)
- **J-type:** Used for jump instructions (e.g., `jal`)

Format	Fields (bit positions)	Use Case
R-type	[31:25] funct7, [24:20] rs2, [19:15] rs1, [14:12] funct3, [11:7] rd, [6:0] opcode	Register-register ALU ops
I-type	[31:20] imm, [19:15] rs1, [14:12] funct3, [11:7] rd, [6:0] opcode	Loads, immediate ALU ops
S-type	[31:25] imm[11:5], [24:20] rs2, [19:15] rs1, [14:12] funct3, [11:7] imm[4:0], [6:0] opcode	Stores
B-type	[31], [30:25], [11:8], [7] (combined as imm), [24:20] rs2, [19:15] rs1, [14:12] funct3, [6:0] opcode	Conditional branches
U-type	[31:12] imm, [11:7] rd, [6:0] opcode	Load upper immediate, AUIPC
J-type	[31], [30:21], [20], [19:12] (combined as imm), [11:7] rd, [6:0] opcode	Jumps

Table 2: RISC-V Instruction Formats

2.5 Our Custom `matmul` Instruction Format

The custom `matmul` instruction follows the R-type format, aligning with standard ALU operations. It uses the `CUSTOM-0` opcode space, which is reserved for user-defined extensions. This design ensures compatibility with the RISC-V decoder logic and avoids conflicts with standard instructions.

```
1 `define OPCODE_CUSTOM_0   7'b0001011
2 `define FUNCT3_MATMUL     3'b000
3 `define FUNCT7_MATMUL     7'b0000000
4
5 `define INST_MATMUL       {`FUNCT7_MATMUL, 5'h0, 5'h0, `FUNCT3_MATMUL, 5'h0,
     `OPCODE_CUSTOM_0}
6 `define INST_MATMUL_MASK  {7'h7F, 5'h0, 5'h0, 7'h7, 5'h0, 7'h7F}
```

Listing 1: Custom Matrix Multiplication Instruction Definition

In this instruction:

- `rd` stores the base address of matrix C (the result),
- `rs1` and `rs2` store the base addresses of matrices A and B, respectively.

When this instruction is detected in the Decode (ID) stage of the processor, the pipeline is stalled and control is handed over to the custom Matmul Wrapper module. This wrapper loads the operands from memory, interfaces with the Systolic Array for computation, and writes the resulting matrix back to the memory starting at the address specified by `rd`.

This setup allows for tight coupling between the custom hardware accelerator and the RISC-V core, ensuring low-latency interaction and efficient data movement through the memory system. It exemplifies the flexibility of RISC-V in supporting domain-specific architectural extensions.

2.6 Systolic Array and Hardware Acceleration

The systolic array is a 2D mesh of processing elements that enables high-throughput matrix multiplication. It operates on streamed data with inherent parallelism and is particularly efficient for dense matrix operations.

In our implementation, a Matmul Wrapper module handles the interface between the RISC-V core and the systolic array. It fetches the operand matrices from memory, streams them into the array, and writes the result matrix back to memory.

2.7 Performance Counters

To evaluate performance, a 64-bit hardware performance counter was integrated. This counter starts on execution and stops upon completion of the `matmul` operation. It can be accessed through custom Control and Status Registers (CSRs). The counter tracks the total number of clock cycles taken by a program, enabling accurate performance profiling and comparison between hardware-accelerated and software-only executions.

2.8 UART (Universal Asynchronous Receiver Transmitter)

UART is a hardware communication protocol that facilitates serial data transfer between the FPGA and a host PC. It converts parallel data from the processor into serial form and vice versa. UART was used in this project for debugging and verification purposes by transmitting matrix values and performance results from the FPGA to a PC terminal.

The UART setup includes:

- Baud Rate Configuration
- Data Register Interface
- Start/Stop Bit Synchronization

The UART was instantiated and mapped to specific memory addresses in the address space, enabling the processor to write data directly to it for transmission.

2.9 FPGA and Vivado Integration

The hardware design was implemented on a Xilinx Nexys4 DDR FPGA, an educational development board powered by the Artix-7 FPGA. The board includes:

- 15,850 logic slices
- 4,860 Kbits of block RAM
- 90 DSP slices

- On-board USB-JTAG and UART

We used Vivado Design Suite for synthesis, placement, routing, and timing analysis. During post-implementation, we observed timing violations due to default 100 MHz clocking. These were mitigated using Vivado's Clock Wizard IP to generate a 65 MHz derived clock that met all timing constraints, ensuring reliable execution.

Together, these components formed the theoretical and practical foundation for realizing the project goal of efficient hardware-accelerated matrix multiplication using custom RISC-V extensions.

3 Module Overview

3.1 Riscv Fetch

Purpose:

- Requests instructions from the instruction cache using the current Program Counter (PC).
- Handles stalls, cache responses, and flushes due to mispredicted branches or exceptions.
- Updates the PC, including branch redirection and privilege management.
- Buffers fetch responses using a skid buffer.
- Reports fetch errors such as access faults or page faults.

Inputs:

clk_i	Clock input signal. Synchronizes fetch logic.
rst_i	Active-high reset. Clears internal state.
fetch_accept_i	Downstream stage ready to accept fetched instruction.
icache_accept_i	Instruction cache ready to accept new fetch request.
icache_valid_i	Valid instruction from instruction cache.
icache_error_i	Error occurred during instruction fetch (e.g., access fault).
icache_inst_i[31:0]	Instruction data from cache.
icache_page_fault_i	Page fault indication during fetch.
fetch_invalidate_i	Invalidate current fetch (e.g., pipeline flush).
branch_request_i	Indicates a branch redirect.
branch_pc_i[31:0]	Target PC for branch.
branch_priv_i[1:0]	Privilege level of branch target.

Outputs:

fetch_valid_o	Instruction valid for decode.
fetch_instr_o[31:0]	Instruction to be decoded.
fetch_pc_o[31:0]	Program Counter of fetched instruction.
fetch_fault_fetch_o	Fetch error (e.g., access fault).
fetch_fault_page_o	Fetch page fault.
icache_rd_o	Request to read instruction cache.
icache_flush_o	Flush invalid instruction from cache.
icache_invalidate_o	Invalidate instruction cache line.
icache_pc_o[31:0]	PC address for cache read.
icache_priv_o[1:0]	Privilege level for PC.
squash_decode_o	Signal to squash instruction in decode stage.

3.2 Riscv Decode

Purpose:

- Translates raw 32-bit instruction data into processor control commands.
- Generates control signals for ALU, memory access, and branch units in later pipeline stages.
- Manages pipeline flow by stalling or squashing instructions based on control signals.
- Critical for identifying instructions related to matrix multiplication such as load, multiply, add, and branch.

Inputs:

clk_i	System clock (from riscv_core).
rst_i	System reset (from riscv_core).
fetch_in_valid_i	Indicates a new instruction is ready (from fetch).
fetch_in_instr_i	The 32-bit instruction data (from fetch).
fetch_in_pc_i	Instruction memory address (from fetch).
fetch_in_fault_fetch_i	Instruction fetch error flag (from fetch).
fetch_in_fault_page_i	Instruction fetch page fault flag (from fetch).
fetch_out_accept_i	Signals the Issue stage is ready.
squash_decode_i	Command to cancel the current instruction (from fetch).

Outputs:

fetch_in_accept_o	Signals this Decode stage is ready (to fetch).
fetch_out_valid_o	Indicates decoded output is valid (to issue).
fetch_out_instr_o	The decoded instruction (to issue).
fetch_out_pc_o	The instruction's PC (to issue).
fetch_out_fault_fetch_o	Fault flags, passed through (to issue).
fetch_out_fault_page_o	Page fault flag, passed through (to issue).
fetch_out_instr_exec_o	Control signal for ALU operations (to issue).
fetch_out_instr_lsu_o	Control signal for Load/Store operations (to issue).
fetch_out_instr_branch_o	Control signal for Branch/Jump operations (to issue).
fetch_out_instr_mul_o	Control signal for Multiply unit (to issue).
fetch_out_instr_div_o	Control signal for Divide unit (to issue).
fetch_out_instr_csr_o	Control signal for CSR access (to issue).
fetch_out_instr_rd_valid_o	Signals a register write-back is needed (to issue).
fetch_out_instr_invalid_o	Flags an illegal instruction (to issue).

3.3 Riscv Decoder

Purpose:

- Implements the RISC-V ISA decoding as a purely combinational logic block.
- Matches the input opcode against predefined bitmasks for all supported instructions.
- Categorizes instructions by asserting control flags (e.g., exec_o, lsu_o).
- Acts as the core control unit, directing instructions to the appropriate execution hardware.

Inputs:

valid_i	Indicates the input instruction is valid (from riscv_decode).

fetch_fault_i	Indicates a fetch error occurred (from riscv_decode).
enable_muldiv_i	Enables the M-extension (multiply/divide instructions) (from riscv_decode).
opcode_i	The 32-bit raw instruction word (from riscv_decode).

Outputs:

invalid_o	Asserted for unrecognized or illegal opcodes (to riscv_decode logic).
exec_o	Asserted for ALU instructions (to riscv_decode logic).
lsu_o	Asserted for Load/Store instructions (to riscv_decode logic).
branch_o	Asserted for Branch and Jump instructions (to riscv_decode logic).
mul_o	Asserted for Multiply instructions (to riscv_decode logic).
div_o	Asserted for Divide/Remainder instructions (to riscv_decode logic).
csr_o	Asserted for system and CSR instructions (to riscv_decode logic).
rd_valid_o	Asserted if the instruction writes a result to a register (to riscv_decode logic).

3.4 Riscv Exec

Purpose:

- Executes core operations based on decoded instruction information.
- Generates immediates of various types (I, B, J, U) with proper sign-extension.
- Performs ALU operations such as arithmetic, logic, and shifts.
- Calculates branch/jump target addresses.
- Determines branch prediction outcomes (taken/not taken).
- Forwards ALU results for register file writeback.
- Controls pipeline by signaling branch requests to Fetch/Decode stages.

Inputs:

clk_i	Clock input for sequential logic.
rst_i	Asynchronous active-high reset.
opcode_valid_i	Indicates if current instruction in execute stage is valid.
opcode_opcode_i[31:0]	Full 32-bit instruction opcode.
opcode_pc_i[31:0]	Program Counter value for current instruction.
opcode_invalid_i	Flag indicating if instruction is invalid (from Decode stage).
opcode_rd_idx_i[4:0]	Destination register index (rd).
opcode_ra_idx_i[4:0]	Source register A index (rs1).
opcode_rb_idx_i[4:0]	Source register B index (rs2).
opcode_ra_operand_i[31:0]	Value of first source operand (register file read).
opcode_rb_operand_i[31:0]	Value of second source operand (register file read).
hold_i	Pipeline stall signal (from memory hazards or data hazards).

Outputs:

branch_request_o	Signals Fetch/Decode stages about executing a branch instruction requiring PC update.
branch_is_taken_o	Indicates conditional branch is predicted taken.
branch_is_not_taken_o	Indicates conditional branch predicted not taken.
branch_source_o[31:0]	PC of the branch-causing instruction.
branch_is_call_o	Indicates a function call branch (e.g., JAL, JALR with rd = x1).
branch_is_ret_o	Indicates a function return branch (e.g., JALR to x1 with zero immediate).
branch_is_jmp_o	Indicates an unconditional jump branch.
branch_pc_o[31:0]	Calculated target PC for the branch.
branch_d_request_o	Delayed branch request signal (typically 1 cycle delay).
branch_d_pc_o[31:0]	Delayed branch target PC.
branch_d_priv_o[1:0]	Dummy privilege output (always 2'b00, no privilege handling here).
writeback_value_o[31:0]	ALU operation result to be written back to register file.

3.5 Arithmetic Logic Unit (ALU)

Purpose:

- Core component of the Execute (EX) stage in the RISC-V processor.
- Performs arithmetic and logical operations such as addition, subtraction, bitwise AND, OR, XOR, shifts, and comparisons.
- Uses control signals to select the operation to perform on two operand inputs.
- Produces a result used for calculations, branch decisions, and memory address computations.

Inputs:

alu_op[3:0]	ALU operation selector (opcode), controls the operation performed. Defined in riscv_defs.v.
alu_a_i[31:0]	Operand A, sourced from the register file.
alu_b_i[31:0]	Operand B, sourced from the register file.

Outputs:

alu_p_o[31:0]	ALU result after performing the selected operation on operands A and B.

Opcode Macros:

- ALU_ADD
- ALU_SUB
- ALU_AND
- ALU_OR
- ALU_XOR

- ALU_LESS_THAN
- ALU_LESS_SIGNED
- ALU_SHIFTL
- ALU_SHIFTR
- ALU_SHIFT_ARITH

3.6 Riscv Issue

Purpose:

- Acts as the control center between Decode and Execute stages.
- Routes decoded signals to appropriate modules while managing pipeline hazards.
- Decodes instructions into opcode, `rs1`, `rs2`, `rd` and reads required registers from the register file.
- Identifies hazards and forwards results to execution units to prevent stalls.
- Stalls the pipeline if forwarding is not possible (e.g., slow operations like division).
- Receives writeback values from execution modules and writes them to the appropriate registers.

Inputs:

clk_i	System clock.
rst_i	System reset.
fetch_valid_i	Indicates if a valid instruction is ready.
fetch_instr_i	The 32-bit raw instruction word.
fetch_pc_i	PC of the fetched instruction.
fetch_fault_fetch_i	Instruction fetch error from memory.
fetch_fault_page_i	Instruction page fault from MMU.
fetch_instr_exec_i	Decoded: standard ALU operation?
fetch_instr_lsu_i	Decoded: Load/Store operation?
fetch_instr_branch_i	Decoded: Branch/Jump operation?
fetch_instr_csr_i	Decoded: CSR operation?
fetch_instr_rd_valid_i	Decoded: Writes to register?
fetch_instr_invalid_i	Decoded: Invalid instruction?
branch_exec_request_i	Branch request signal from Execute.
branch_exec_is_taken_i	Indicates conditional branch predicted taken (from Execute).
branch_exec_is_not_taken_i	Indicates conditional branch predicted not taken (from Execute).
branch_exec_source_i	PC of the branch-causing instruction (from Execute).
branch_exec_is_call_i	Indicates a function call branch (from Execute).
branch_exec_is_ret_i	Indicates a function return branch (from Execute).
branch_exec_is_jmp_i	Indicates an unconditional jump branch (from Execute).
branch_exec_pc_i	Target PC for the branch (from Execute).
branch_d_exec_request_i	Delayed branch request (from Execute).
branch_d_exec_pc_i	Delayed target PC (from Execute).
branch_d_exec_priv_i	Delayed privilege info (from Execute).
writeback_exec_value_i	Result from standard ALU operations (from Execute).
exec_stall_request_i	Stall request from Execute (e.g., Matmul busy).

exec_rd_idx_i	Destination register index in Execute stage.
writeback_mem_valid_i	Indicates if memory access result is valid.
writeback_mem_value_i	Data read from memory for forwarding.
writeback_mem_exception_i	Memory access exception.
lsu_stall_i	Stall request from LSU (e.g., cache miss).
csr_result_e1_value_i	Value read from CSR.
csr_result_e1_write_i	CSR write enable.
csr_result_e1_wdata_i	Data to write to CSR.
csr_result_e1_exception_i	CSR-related exceptions (e.g., ecall).
branch_csr_request_i	Branch request from CSR.
branch_csr_pc_i	Target PC from CSR branch.
branch_csr_priv_i	Privilege level from CSR branch.
take_interrupt_i	Global interrupt pending.
d_stall	External debug stall signal from DataHazard.

Outputs:

fetch_accept_o	Signals that riscv_fetch is ready for the next instruction.
branch_request_o	Global branch signal to update the PC.
branch_pc_o	Target PC for the branch.
branch_priv_o	Privilege level for the branch target.
exec_opcode_valid_o	Valid instruction signal for riscv_exec.
opcode_opcode_o	Instruction word passed to riscv_exec.
opcode_pc_o	Instruction PC passed to riscv_exec.
opcode_invalid_o	Invalid instruction flag for riscv_exec.
opcode_rd_idx_o	Destination register index passed to riscv_exec.
opcode_ra_idx_o	Source register A index passed to riscv_exec.
opcode_rb_idx_o	Source register B index passed to riscv_exec.
opcode_ra_operand_o	Prepared operand A value for riscv_exec.
opcode_rb_operand_o	Prepared operand B value for riscv_exec.
lsu_opcode_valid_o	Valid signal for riscv_lsu.
lsu_opcode_opcode_o	Instruction word to riscv_lsu.
lsu_opcode_pc_o	Instruction PC to riscv_lsu.
lsu_opcode_invalid_o	Invalid instruction flag for riscv_lsu.
lsu_opcode_rd_idx_o	Destination register index to riscv_lsu.
lsu_opcode_ra_idx_o	Source register A index to riscv_lsu.
lsu_opcode_rb_idx_o	Source register B index to riscv_lsu.
lsu_opcode_ra_operand_o	Operand A to riscv_lsu.
lsu_opcode_rb_operand_o	Operand B to riscv_lsu.
csr_opcode_valid_o	Valid signal to CSR unit.
csr_opcode_opcode_o	Instruction word to CSR unit.
csr_opcode_pc_o	Instruction PC to CSR unit.
csr_opcode_invalid_o	Invalid instruction flag to CSR unit.
csr_opcode_rd_idx_o	Destination register index to CSR unit.
csr_opcode_ra_idx_o	Source register A index to CSR unit.
csr_opcode_rb_idx_o	Source register B index to CSR unit.
csr_opcode_ra_operand_o	Operand A to CSR unit.
csr_opcode_rb_operand_o	Operand B to CSR unit.
csr_writeback_write_o	CSR write enable signal.
csr_writeback_waddr_o	CSR write address.
csr_writeback_wdata_o	CSR write data.

csr_writeback_exception_o	CSR exception flag.
csr_writeback_exception_pc_o	PC causing CSR exception.
csr_writeback_exception_addr_o	Exception address for CSR.
exec_hold_o	Stall signal for riscv_exec pipeline register.
mul_hold_o	Stall signal for riscv_multiplier pipeline register.
interrupt_inhibit_o	Signal to temporarily block interrupts.

3.7 DataHazard Unit

Purpose:

- Implements a forwarding (or bypassing) mechanism to resolve read-after-write (RAW) data hazards without stalling.
- Checks if the operands needed by the instruction in the Decode stage are being produced by instructions currently in the Execute or Memory stages.
- If a dependency exists, it bypasses the register file and directly forwards the newer result from the Execute or Memory stage to the ALU's input.
- This forwarding avoids pipeline stalls on dependent instructions, which are common in tight loops and algorithms like matrix multiplication.

Inputs:

reset_i	System reset.
id_ra_index_w	Source register 1 index (rs1) in Decode stage.
id_rb_index_w	Source register 2 index (rs2) in Decode stage.
ex_rd_index_r	Destination register index (rd) in Execute stage.
mem_rd_index_w	Destination register index (rd) in Memory stage.
id_ra_value_r	Operand value 1 read from register file in Decode stage.
id_rb_value_r	Operand value 2 read from register file in Decode stage.
ex_alu_res_r	ALU result produced in Execute stage.
mem_wb_alu_result_r	ALU result from instruction in Memory stage.
mem_rdata_w	Data fetched from memory in Memory stage.
mem_access_w	Indicates memory load occurred in Memory stage.

Outputs:

exe_ra_r	Corrected operand A value forwarded to Execute stage ALU.
exe_rb_r	Corrected operand B value forwarded to Execute stage ALU.

3.8 Hazard

Purpose:

- Detects the load-use data hazard, where an instruction needs data that a preceding load instruction has not yet fetched from memory.
- Compares the current instruction's source registers (rs1 or rs2) with the destination register (rd) of the immediately preceding instruction.
- Specifically targets cases where the preceding instruction is a load word (lw).
- If such a hazard is found, asserts the stall signal to pause the pipeline for one cycle, allowing memory access to complete.

- Ensures correctness in common dependency patterns, such as multiply-add sequences in matrix multiplication.

Inputs:

opcode_i	The instruction just fetched from memory (from mem_i_inst_i port).
clk_i	System clock.
reset_i	System reset.

Outputs:

stall	Single-bit signal to pause the pipeline if a load-use hazard is detected.

3.9 riscv_core

Purpose:

- Acts as the top-level "schematic" of the processor, instantiating and connecting all major functional blocks (fetch, decode, exec, lsu, multiplier, etc.).
- Defines the complete pipeline structure and orchestrates instruction and data flow across all pipeline stages.
- Connects all control and data paths, including forwarding logic and the stall signal from the DataHazard unit to the issue stage.
- Serves as the boundary between the CPU core and the external system.
- Manages external communication with instruction/data memory and interrupt controllers.

Inputs:

clk_i	System clock (from Testbench/System).
rst_i	System reset (from Testbench/System).
mem_d_data_rd_i	Data received from a load operation (from External Data Memory).
mem_d_accept_i	Data memory handshake signal: memory accepts new request.
mem_d_ack_i	Data memory handshake signal: read/write complete.
mem_d_error_i	Data memory error signal.
mem_i_inst_i	Instruction data fetched from memory (from External Instruction Memory).
mem_i_accept_i	Instruction memory handshake signal: memory accepts new request.
mem_i_valid_i	Instruction memory handshake signal: instruction valid.
mem_i_error_i	Instruction memory error signal.
intr_i	External interrupt request (from Testbench/System).
reset_vector_i	Address where execution starts after reset.
cpu_id_i	Identifier for the processor core.

Outputs:

mem_d_addr_o	Address for a memory operation (to External Data Memory).

data_wr_o	Data to be written to memory.
mem_d_rd_o	Read enable for data memory.
wr_o	Write enable for data memory.
mem_i_pc_o	Program Counter sent to fetch an instruction (to External Instruction Memory).
mem_i_rd_o	Read enable for instruction memory.
mem_d_cacheable_o	Indicates if data memory access is cacheable.
mem_i_cacheable_o	Indicates if instruction memory access is cacheable.
invalidate_o	Cache invalidate signal (to memory system).
flush_o	Cache flush signal (to memory system).

3.10 instruction_memory

Purpose:

- Represents the Instruction ROM (Read-Only Memory) in the Fetch stage of a RISC-V processor pipeline.
- Holds the program code and outputs a 32-bit instruction based on the Program Counter (PC).
- Accepts a word-aligned address and provides the corresponding instruction, either combinationally or synchronously.
- Always ready to serve fetch requests, supporting uninterrupted instruction flow unless externally stalled.

Inputs:

clk_i	Clock signal. Present for compatibility; unused in current combinational implementation.
reset_i	Reset signal. Included for extensibility; not used in current implementation.
iaddr_i[31:0]	Address input from the Program Counter (PC) to access the instruction memory.
ird_i	Instruction read enable. Present for future extensibility; unused in current implementation.

Outputs:

accept	Always set to 1, indicating readiness to serve instruction fetches.
irdata_o[31:0]	Instruction fetched from memory based on iaddr_i; sent to Decode stage.

3.11 data_memory

Purpose:

- Represents the data memory (RAM) used during the Memory Access (MEM) stage of the RISC-V processor pipeline.
- Supports both read and write operations with byte-level granularity.
- Handles accesses of various sizes—byte, half-word, and word—using byte-enable control signals.
- Interfaces with the processor pipeline to execute load/store instructions and manage data movement accordingly.

Inputs:

clk_i	Clock signal required for synchronous memory operations.
reset_i	Resets internal memory state and output signals.
daddr_i[31:0]	Word-aligned address for memory access, typically ALU output.
dwdata_i[31:0]	Data input used during store operations.
dsize_i[1:0]	Data size selector: 00 = byte (8-bit), 01 = half-word (16-bit), 10 = word (32-bit).
drd_i	Read enable signal, asserted for load instructions.
dwr_i	Write enable signal, asserted for store instructions.
dbe_w[3:0]	Byte-enable mask; each bit enables writing a corresponding byte in the 32-bit word.

Outputs:

drdata_o[31:0]	Data read from memory when drd_i is asserted.
accept	Handshake signal indicating readiness to perform a memory operation.
acknowledge	Signals completion of the current memory transaction.

3.12 riscv_csr

Purpose:

- Acts as the interface between the instruction pipeline and the CSR register file.
- Decodes and executes CSR-related instructions (e.g., CSRRW, ECALL, MRET).
- Detects early exceptions such as illegal CSR accesses or system calls.
- Prepares CSR data for writes, sets, or clears.
- Manages pipeline control signals including traps, returns, flushes, and interrupts.
- Interfaces with the MMU by providing configuration signals and flushing TLB.
- Asserts interrupt requests when enabled and pending.

Inputs:

clk_i	Clock input.
rst_i	Asynchronous active-high reset.
intr_i	External interrupt signal.
opcode_valid_i	Indicates if current instruction is valid.
opcode_opcode_i[31:0]	Instruction opcode.
opcode_pc_i[31:0]	Program Counter of the current instruction.
opcode_invalid_i	Invalid instruction flag.
opcode_rd_idx_i[4:0]	Destination register index.
opcode_ra_idx_i[4:0]	Source register A index or immediate.
opcode_rb_idx_i[4:0]	Source register B index.
opcode_ra_operand_i[31:0]	Operand value from source register A.
opcode_rb_operand_i[31:0]	Operand value from source register B.
csr_writeback_write_i	Writeback enable signal to CSR from later stage.
csr_writeback_waddr_i[11:0]	CSR address for writeback.
csr_writeback_wdata_i[31:0]	Data to write into CSR.

csr_writeback_exception_i[5:0]	Exception code from later pipeline stage.
csr_writeback_exception_pc_i[31:0]	PC of faulting instruction.
csr_writeback_exception_addr_i[31:0]	Address associated with exception.
cpu_id_i[31:0]	CPU identifier (mhartid).
reset_vector_i[31:0]	Reset vector for initial PC.
interrupt_inhibit_i	Signal to temporarily suppress interrupts.

Outputs:

csr_result_e1_value_o[31:0]	Value read from CSR or fault-related value.
csr_result_e1_write_o	Write enable for CSR result to register file.
csr_result_e1_wdata_o[31:0]	Data to be written to CSR.
csr_result_e1_exception_o[5:0]	Exception code generated by CSR module.
branch_csr_request_o	Indicates a trap or xRET branch should be taken.
branch_csr_pc_o[31:0]	Target PC for trap or return.
branch_csr_priv_o[1:0]	Privilege level after trap/return.
take_interrupt_o	Signal to trigger interrupt handling.
ifence_o	Triggers instruction cache flush (FENCE.I).
mmu_priv_d_o[1:0]	Effective privilege level for MMU.
mmu_sum_o	mstatus.SUM bit output.
mmu_mxr_o	mstatus.MXR bit output.
mmu_flush_o	Triggers TLB flush.
mmu_satp_o[31:0]	Current satp CSR value.

3.13 riscv_performance_counter

Purpose:

- Measures the number of clock cycles between program start and a halt instruction.
- Starts counting when the first instruction is fetched at PC 0x00000000.
- Stops counting when an instruction is fetched at PC 0x00000088.
- Outputs the total cycle count once halted.
- Indicates whether counting is active and whether measurement is complete.
- Flags detection of the halt instruction.

Inputs:

clk_i	Clock signal driving the performance counter.
reset_i	Asynchronous reset signal; clears internal state and counters.
pc_i[31:0]	Current Program Counter input used to detect start and halt points.
instruction_i[31:0]	Instruction at the current PC; not directly used for logic, but passed in.
instruction_valid_i	Indicates whether the instruction at the current PC is valid. Required to recognize start and halt conditions.

Outputs:

cycle_count_o[31:0]	Total number of cycles counted between start and halt.

counting_active_o	High when cycle counting is in progress.
measurement_done_o	High when halt condition is detected and cycle measurement is complete.
halt_detected_o	Asserted when halt instruction is fetched at PC 0x00000088.

3.14 matmul_controller

Purpose:

- Orchestrates matrix multiplication using a systolic array accelerator.
- Fetches matrix A and B elements from memory into internal buffers.
- Triggers the systolic array computation once data is loaded.
- Writes the resulting matrix C back to memory.
- Manages memory access and pipeline timing using an FSM with distinct states for each phase.

Inputs:

clk_i	Clock signal for synchronous logic.
rst_i	Asynchronous reset signal to initialize internal states and counters.
start_i	Start signal to initiate matrix multiplication.
addr_a_base_i[31:0]	Base memory address of matrix A.
addr_b_base_i[31:0]	Base memory address of matrix B.
addr_c_base_i[31:0]	Base memory address where the result matrix C is to be written.
mem_data_rd_i[31:0]	Data read from memory during matrix A/B fetch phase.
mem_gnt_i	Memory grant signal to indicate readiness to serve a request.
mem_ack_i	Memory acknowledgment signal indicating successful data transfer.

Outputs:

busy_o	High when the controller is busy performing matrix multiplication.
mem_addr_o[31:0]	Address sent to memory for reading matrix A/B or writing matrix C.
mem_rd_o	Read enable signal for memory interface.
mem_wr_o[3:0]	Write enable mask for memory writes (4 bits for byte-level granularity).
mem_data_wr_o[31:0]	Data to be written to memory during matrix C store phase.
mem_req_o	Request signal asserting memory access initiation.

3.15 systolic_array

Purpose:

- Implements a systolic array architecture for matrix multiplication.
- Takes input matrices matrix_A and matrix_B and produces the result matrix resMatrix.
- Uses time-skewed data movement to compute dot products in parallel over multiple cycles.

Inputs:

clk	Clock signal to drive computation steps and internal operations.
reset	Synchronous reset to clear state and output registers.
matrix_A	Input matrix A of size ROW_A × COL_A.
matrix_B	Input matrix B of size ROW_B × COL_B.
initiateCompute	Control signal to begin matrix multiplication operation.

Outputs:

computeDone	Asserted high when the computation of resMatrix is complete.
resMatrix	Output matrix containing the result of A × B, of size ROW_A × COL_B.

3.16 MemoryUartDumper

Purpose:

- Dumps contents of memory via UART in hexadecimal format.
- Converts 8-bit memory data to ASCII hex and transmits it serially.
- Sends a formatted header, spaces between bytes, and newlines after 16 bytes.

Inputs:

clk	System clock signal, typically 50 MHz.
reset	Active-high synchronous reset signal.
start_dump	Signal to begin the memory dump over UART.
mem_rdata	8-bit data input from memory at address mem_addr.

Outputs:

dump_in_progress	High while memory is being dumped. Returns low when complete.
mem_addr	Address line to fetch memory contents sequentially.
tx	UART serial transmit line carrying memory content in ASCII.
bytes_sent_debug	Debug counter of number of bytes transmitted.

3.17 uart_tx

Purpose:

- Implements a UART transmitter for serial communication.
- Transmits 8-bit data over a single line at a fixed baud rate.
- Follows standard UART format: 1 start bit, 8 data bits, 1 stop bit, no parity.

Inputs:

clk	System clock used to time bit transmissions. Designed for 50 MHz.
reset	Active-high reset signal. Resets state, counters, and output.

data	8-bit parallel data to be serialized and transmitted.
tx_start	Pulse signal indicating a new byte should be transmitted.

Outputs:

tx	UART transmit line. Outputs serialized data (start, 8-bit data, stop).
tx_busy	High during transmission of a byte, low when ready for new data.

3.18 riscv_top

Purpose:

- Top-level RISC-V SoC module integrating clock wizard, performance counter, UART dumper, instruction/data memory, and core processor.
- Drives a 16-bit LED display for visual feedback on execution and status.
- Manages startup delay and memory dump functionality post-reset.

Inputs

Name	Description
clk_100mhz_i	100 MHz input clock signal.
rst_i	Asynchronous reset signal.
intr_i	External interrupt signal input to the RISC-V core.

Outputs

Name	Description
tx_o	UART transmit line used by the MemoryUartDumper to send dumped memory contents.
led_o [15:0]	LED output showing system status, cycle count, and halt information.

3.19 Memory Dump UART Python Script

Purpose:

- Automatically identifies the correct COM port connected to the Nexys 4 DDR board.
- Reads the memory dump transmitted via UART.
- Displays both raw hexdump (little-endian) and interpreted 32-bit word values (big-endian).

Python Script:

```
import serial
import time
import sys
import re

# find_nexys_port() function is unchanged and remains excellent.
def find_nexys_port():
    import serial.tools.list_ports
    ports = serial.tools.list_ports.comports()
    nexys_ports = []

```

```
    print("Searching for COM ports...")
    for i, port in enumerate(ports):
        if any(keyword in port.description.lower() for keyword in
               ['digilent', 'ftdi', 'usb serial', 'nexys']):
            nexys_ports.append(port)

    if nexys_ports:
        print(f"\nFound {len(nexys_ports)} potential Nexys 4 DDR port(s):")
        for i, port in enumerate(nexys_ports):
            print(f"  {port.device} - {port.description}")

        try:
            higher_port = max(nexys_ports, key=lambda p: int(re.search(r'(\d
+)$', p.device).group(1)))
            print(f"\n--> Automatically selecting higher numbered port: {
higher_port.device}\n")
            return higher_port.device
        except (AttributeError, ValueError):
            print("\n--> Could not determine higher port, selecting first
found.\n")
            return nexys_ports[0].device

    if not ports:
        print("No COM ports found!")
        return None

    print("\nCould not automatically identify Nexys port.")
    for i, port in enumerate(ports):
        print(f"  {i}: {port.device} - {port.description}")
    try:
        choice = int(input("Enter the number of the port to use: "))
        return ports[choice].device
    except (ValueError, IndexError):
        print("Invalid selection. Exiting.")
        return None

def process_and_print_buffer(buffer):
    """
    Takes the raw text buffer, prints a clean hexdump, and then prints the
    interpreted 32-bit integer values.
    """
    if not buffer:
        return

    print("\n========================= MEMORY DUMP RECEIVED
==========================")

    # 1. Sanitize the input and get a clean list of bytes
    payload = buffer.split("Memory Dump:\r\n", 1)[-1]
    hex_string = re.sub(r'[^0-9A-Fa-f]', '', payload, flags=re.IGNORECASE)
    byte_list = [hex_string[i:i+2].upper() for i in range(0, len(hex_string)
, 2)]

    # 2. Print the raw hexdump first (as seen in memory)
    print("--- Raw Little-Endian Memory Content ---")
    for i in range(0, len(byte_list), 16):
        chunk = byte_list[i:i+16]
        address = f"0x{i:04X}: "
        hex_part = " ".join(chunk)
        print(f"{address}{hex_part}")

```

```
    # 3. Process and print the interpreted 32-bit words
    print("\n--- Interpreted 32-bit Words (Big-Endian) ---")
    print("Address    | Word 0 (Hex)   | Word 1 (Hex)   | Word 2 (Hex)   | Word 3 (Hex)")
    print("           | Word 0 (Dec)   | Word 1 (Dec)   | Word 2 (Dec)   | Word 3 (Dec)")
    print("-----------+----------------+----------------+----------------+----------------")

    for i in range(0, len(byte_list), 16):
        chunk = byte_list[i:i+16]
        address = f" 0x{i:04X}    |"
        hex_line_parts = []
        dec_line_parts = []

        for j in range(0, len(chunk), 4):
            word_bytes_little_endian = chunk[j:j+4]
            if len(word_bytes_little_endian) == 4:
                word_bytes_big_endian = word_bytes_little_endian[::-1]
                hex_str = "".join(word_bytes_big_endian)
                hex_line_parts.append(f" 0x{hex_str:<12}")
                decimal_val = int(hex_str, 16)
                dec_line_parts.append(f" {decimal_val:<14}")

        print(address + "".join(hex_line_parts))
        print("           |" + "".join(dec_line_parts))
```

4 Introduction to RISC-V GNU Toolchain and Spike ISA Simulator

The RISC-V ecosystem provides open-source tools to write, compile, simulate, and test RISC-V assembly and C/C++ programs. Two key components in this flow are:

- **GNU RISC-V Toolchain:** Includes the compiler `riscv32-unknown-elf-gcc`, assembler, linker, and utilities like `objcopy`. It allows compilation of standard C/C++ programs targeting the RISC-V architecture.
- **Spike ISA Simulator:** Spike is the official RISC-V ISA simulator. It supports functional simulation and is ideal for debugging compiled RISC-V binaries.

4.1 Getting Started with Docker Container

To simplify the setup process, a pre-built Docker container is available with the GNU Toolchain and Spike already installed.

```
docker pull arjunmallya/ubuntu:withmatmul
docker run -it --rm arjunmallya/ubuntu:withmatmul bash
```

Listing 2: Pull and run the Docker container

This opens a shell with the RISC-V toolchain and Spike pre-installed.

4.2 Compiling and Converting RISC-V C Code

4.2.1 Step 1: Navigate to Working Directory

```
cd /opt/cpp_nn_riscv
```

4.2.2 Step 2: Create and Compile a C File

Create a file named `test.c` with any RISC-V-compatible C code. For example:

```
int main() {
    return 42;
}
```

Listing 3: Simple C Program

Compile it using:

```
riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -nostdlib -Ttext=0x0 -o test.elf
    test.c
```

- `-march=rv32i`: Targets base integer RISC-V ISA
- `-mabi=ilp32`: Uses 32-bit integer ABI
- `-nostdlib`: Omits linking the standard library
- `-Ttext=0x0`: Places code starting at address 0x0

4.2.3 Step 3: Generate Binary and Hex Files

Convert the ELF output to binary and then to hexadecimal:

```
riscv32-unknown-elf-objcopy -O binary test.elf test.bin
hexdump -v -e '1/4 "%08x\n"' test.bin > test.hex
```

4.2.4 Step 4: Clean and Format Hex for Verilog Use

To prepare the hex for use in Verilog simulations:

```
cat test.hex | tr -d '\n\r' | fold -w8 > test_clean.hex
```

Each line of `test_clean.hex` will now be 8 characters (32 bits), suitable for Verilog memory initialization.

4.3 Summary of Commands

Task	Command/Tool
Compile C file	`riscv32-unknown-elf-gcc`
Convert to binary	`riscv32-unknown-elf-objcopy`
Generate hex	`hexdump`
Format hex for Verilog	`tr`, `fold`
Run Docker container	`docker run`

Table 39: Toolchain Workflow Summary

These tools provide a robust flow for writing, compiling, and testing custom C/C++ programs for RISC-V architectures.

5 Integration of Custom Matmul instruction to GNU Toolchain and Spike Simulator

The objective of this project is to extend the RISC-V instruction set with a custom `matmul` instruction. This instruction is intended to accelerate matrix multiplication operations, which are fundamental to

applications such as neural networks. This document details the initial phase of integration into the Spike simulator, where the instruction operates on matrices of fixed, predefined dimensions.

The `matmul` instruction is designed as an R-type instruction with the following intended behavior for this initial phase:

- **Operation:** Integer vector-matrix multiplication (`output_vector = input_vector * weight_matrix`).
- **Operands in GPRs:**
  - `rs1`: Holds the base memory address of the input vector (array of `int8_t`).
  - `rs2`: Holds the base memory address of the weight matrix (array of `int8_t`).
  - `rd`: Initially holds the base memory address for the output vector (array of `int32_t`). After execution, `rd` is updated with a status code (0 for success).
- **Data Types:** Input elements are `int8_t`, weight elements are `int8_t`, and output/accumulator elements are `int32_t`.

6 Modifications to RISC-V GNU Toolchain

To enable the assembler and disassembler to recognize the new `matmul` instruction, the following modifications were made:

6.1 opcodes/riscv-opc.c

```
{"matmul", 0, INSN_CLASS_I, "d,s,t", 0x0200000b, 0xfe00707f, match_opcode, 0 },
```

Listing 4: Changes to `matmul` in `riscv-opc.c`

7 Modifications to RISC-V Spike Simulator

7.1 riscv/encoding.h

```
#define MATCH_MATMUL 0x0200000b
#define MASK_MATMUL  0xfe00707f

#ifdef DECLARE_INSN
  DECLARE_INSN(matmul, MATCH_MATMUL, MASK_MATMUL)
#endif
```

Listing 5: Changes to `riscv/encoding.h`

7.2 riscv/insns/matmul.h

```
reg_t addr_a = RS1;
reg_t addr_b = RS2;
reg_t addr_c = RD;

// Read dimensions from registers x14, x15, x16
size_t M = STATE.XPR[14];
size_t N = STATE.XPR[15];
size_t K = STATE.XPR[16];

for (size_t i = 0; i < M; ++i) {
    for (size_t j = 0; j < N; ++j) {
        int32_t sum = 0;
        for (size_t k = 0; k < K; ++k) {
            int32_t a = MMU.load<int32_t>(addr_a + (i * K + k) * sizeof(int32_t));
```

```
            int32_t b = MMU.load<int32_t>(addr_b + (k * N + j) * sizeof(int32_t));
            sum += a * b;
        }
        MMU.store<int32_t>(addr_c + (i * N + j) * sizeof(int32_t), sum);
    }
}

WRITE_RD(addr_c);
```

Listing 6: Updated dynamic dimension implementation

7.3 riscv/riscv.mk.in

```
matmul \
```

7.4 riscv-isa-sim/disasm/disasm.cc

```
DEFINE_RTYPE(matmul);
```

Listing 7: Addition to disassembler

7.5 Rebuild Instructions

```
../configure --prefix=/opt/riscv
make clean
make -j$(nproc)
make install
```

8 Current Status and Testing

With the above modifications, the `matmul` instruction has been successfully integrated with dynamic dimension support.

A C++ test program was developed:

- Initializes matrices A, B, and C in memory using `int32_t`.
- Loads base addresses of A (into a0), B (into a1), and C (into a2).
- Loads M, N, K into a4, a5, a6 respectively.
- Executes: `matmul a2, a0, a1`.
- Verifies that C = A * B and checks the returned address.

The test confirms correct decoding, register usage, and memory-based matrix multiplication.