Optimised Microprocessor Architecture Design for Machine Learning Applications

Lizhi Jiang

3rd Year Project Final Report

Department of Electronic &
Electrical Engineering

UCL

Supervisor: Prof. Robert Killey

12 April 2024

I have read and understood UCL's and the Department's statements and guidelines concerning plagiarism.

I declare that all material described in this report is my own work except where explicitly and individually indicated in the text. This includes ideas described in the text, figures and computer programs.

I acknowledge the use of Chat-GPT 4.0 (Publisher [OpenAI], URL of the AI system [https://chat.openai.com]), which was used for grammar checks and LATEX formatting, without influencing the research content or conclusions.

This report contains 47 pages (excluding this page and the appendices) and 8757 words.

Signed: _____ Date: _____

(Student)

Contents

Optimised Microprocessor Architecture Design for Machine Learning Applications

Lizhi Jiang

The project presents a five-stage pipelined RISC-V CPU design with custom instructions—namely vector addition, and matrix multiplication—to accelerate convolutional neural network (CNN) inference computations. Implemented in Verilog and verified via ModelSim simulation, the processor architecture consists of the classic pipeline stages—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB)—together with pipeline registers between stages. Techniques such as data forwarding and pipeline stalling are employed to address common hazards and ensure correct execution in the presence of instruction dependencies. Fixed-point arithmetic is used in place of floating-point for CNN acceleration, balancing resource limits and efficiency on FPGAs.

To validate matrix multiplication accuracy and pipeline performance, I incorporated CNN-trained data: a larger file of more than 200 lines was generated by a TensorFlow-based CNN, from which only the first 64 lines (each 8 bits wide) were transformed into a `hex` file for final testing in the ALU-based design. Both single-cycle and pipelined implementations produce matching outputs, verifying correctness, while the pipelined CPU demonstrates significantly better performance in cycle count reduction. Although future work may explore hardware-level deployment (e.g., on a DE10-Nano FPGA), the current results already affirm the feasibility and scalability of this approach for CNN-focused computations under resource-conscious environments.

1 Introduction and Literature Review

1.1 Microprocessors and Motivation

The evolution of microprocessors can be traced back to the late 1960s, when rapid advancements in integrated circuit (IC) technology made it possible to integrate the core functions of a computer onto a single chip. In 1971, Intel introduced the world's first commercial microprocessor—the Intel 4004. Although the 4004 was merely a 4-bit processor, its development ushered in the microprocessor era, proving that a computer's essential functions could be consolidated into one compact IC. Following this breakthrough, Intel released the 8-bit 8008 and the 16-bit 8080 processors, both of which propelled the computer industry forward and laid the groundwork for embedded systems and industrial automation [6].

As IC manufacturing technology improved, microprocessors underwent a significant qualitative leap. In 1978, Intel launched the 8086 processor, heralding the birth of the x86 architecture [14]. Renowned for its strong backward compatibility and versatile design, the x86 architecture rapidly became the dominant force in personal computing, evolving through various iterations such as the 80286, 80386, and the Pentium series. Beyond desktop computing, x86 also established its presence in servers and workstations. Meanwhile, in the 1980s, Acorn Computers in the United Kingdom introduced the ARM architecture. In contrast to x86, ARM embraced a Reduced Instruction Set Computing (RISC) philosophy emphasizing low power consumption and high efficiency, making it particularly well-suited for embedded devices. ARM's proliferation tracked the meteoric rise of mobile computing, propelling ARM cores into dominance for smartphones and other portable electronics. This rapid expansion fostered a paradigm shift in the microprocessor industry, elevating power and energy efficiency to a primary concern.

Beyond x86 and ARM, other RISC-based architectures—MIPS and PowerPC—also emerged, each with its own performance and power advantages [6]. MIPS processors, known for high performance and energy efficiency, found widespread use in embedded applications, while PowerPC was initially integral to Apple computers and numerous embedded systems. In many cases, these architectures supported high-performance servers on one hand and power-critical embedded systems on the other.

Entering the 21st century, microprocessors have tended toward multicore designs and system-on-chip (SoC) solutions [6]. Contemporary devices often integrate traditional CPU cores with specialized accelerators, such as Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), and AI accelerators, to address specific performance bottlenecks through parallelism [6]. Techniques like pipelining, superscalar execution, and out-of-order processing further exploit concurrency to deliver higher instruction throughput.

From a personal perspective, the rapid evolution of microprocessors is not only a technological feat but also reflects humankind's unceasing pursuit of more efficient data handling and more seamless integration into everyday life. Whether in the high-performance CPUs powering modern servers or the energy-efficient ARM chips fueling smartphones, microprocessors continue to serve as the engine of digital transformation. With the explosive growth of the Internet of Things (IoT), big data, and artificial intelligence, microprocessors will likely evolve into ever more intelligent, efficient, and tightly integrated systems, lighting the path to an even more ubiquitously networked and data-driven future [6].

A single-cycle microprocessor completes each instruction in a single clock cycle. That is, instruction fetch, decoding, execution, memory access, and result write-back

Table 1: Comparison of Microprocessor Architectures

	x86	**ARM**	**RISC (e.g., MIPS, PowerPC)**
Origin	Developed by Intel (and AMD)	Developed by Acorn Computers / ARM Ltd.	Based on RISC principles
Design Philosophy	CISC (Complex Instruction Set Computing)	RISC with emphasis on power efficiency	Pure RISC design
Applications	Desktops, laptops, servers, workstations	Mobile devices, embedded systems, IoT	Embedded systems, specialized HPC
Evolution	Multicore and SoC designs	Dominant in mobile computing	Used in customizable, open-source solutions

occur within one clock cycle [5]. While appealing for its simplicity and straightforward control design, a single-cycle approach must elongate its clock period to accommodate the slowest instruction, thus limiting the maximum clock frequency. The result is an educationally friendly but performance-constrained architecture.

Although single-cycle processors logically move through these five steps (IF, ID, EX, MEM, WB), each instruction consumes the entire clock cycle in sequence rather than being overlapped in hardware. This design often requires more routing, multiplexers, and dedicated combinational logic than a multi-cycle or pipelined equivalent, because each instruction path must be handled within that single cycle.

In contrast, a pipelined microprocessor divides execution into multiple stages so that multiple instructions can be processed simultaneously. Borrowing the idea of an industrial assembly line, pipeline stages (IF, ID, EX, MEM, WB) are implemented with distinct hardware units, allowing subsequent instructions to be fetched as the first moves on to later stages [5]. This concurrent activity can significantly boost throughput and effectively decrease the average clock cycles per instruction, although it introduces complexities such as data hazards and control hazards. These hazards must be handled via data forwarding, pipeline stalling, and branch prediction, among other techniques.

In a five-stage pipelined processor, each instruction ideally advances one pipeline stage per clock cycle. Hence, after a short initial delay (the pipeline fill time), the processor can complete one instruction per cycle. The real-world performance gains, however, are slightly lower than this ideal due to hazards. Even so, pipelining is a fundamental approach to achieve higher performance in modern CPU designs.

1.2 RISC-V Instruction Set Architecture(ISA)

RISC-V is a Reduced Instruction Set Computer (RISC) architecture originally developed at the University of California, Berkeley, and is distinguished by its open, modular, and extensible nature [2, 14]. Unlike proprietary architectures like x86 or ARM, RISC-V's specifications are publicly available, thus allowing both academic and industrial entities to tailor and extend the instruction set as needed [1].

In RISC-V, every basic instruction is encoded in a fixed 32-bit format. Taking the R-type instruction as an example, its format comprises a 7-bit opcode, a 5-bit destination register (`rd`), a 3-bit function field (`funct3`), a 5-bit first source register (`rs1`), a 5-bit second source register (`rs2`), and a 7-bit additional function field (`funct7`). The `funct7` field further distinguishes variants of an operation (for example, `add` vs. `sub`), while `funct3` pinpoints the exact operation type [14]. Other instruction formats include I-type, S-type, B-type, U-type, and J-type, each bringing different forms of immediate fields to support arithmetic with constants, data load/store, and branching operations. During program execution, the CPU fetches the 32-bit binary instruction from memory, decodes its opcode and function fields to generate the needed control signals for the ALU and other CPU components, and then proceeds through memory access and write-back to retire the instruction [6].

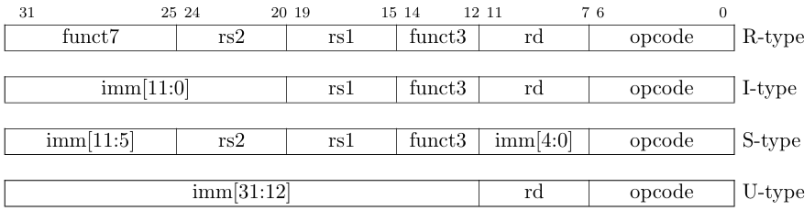


Figure 1: RISC-V base instruction formats [1]

A significant advantage of RISC-V is its simple but flexible foundation, which supports a base integer instruction set (e.g., RV32I or RV64I) that can be expanded with specialized extensions [12]. For example, in this project, custom instructions were implemented to handle matrix multiplication, vector addition, for CNN inference tasks. Leveraging reserved opcode fields or reassigning specific bits in the instruction format, one can integrate new operations into the decode and execution units, thus accelerating specialized workloads.

1.3 Literature Review

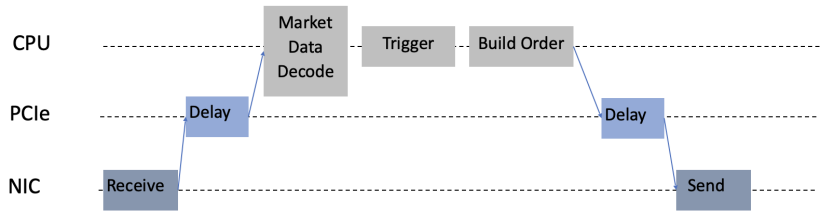
1.3.1 Applications

Field-Programmable Gate Arrays (FPGAs), the RISC-V ISA, and machine learning have intersected significantly over the past few decades, giving rise to reconfigurable, energy-efficient, and high-performance computing platforms for AI acceleration. FPGAs originated in the mid-1980s as a solution to the high non-recurring engineering costs of ASICs. Early FPGAs offered limited capabilities, but successive generations integrated sophisticated DSP blocks, large on-chip memories, and high-speed transceivers, transforming them into powerful reconfigurable platforms that now underpin a variety of commercial and research applications [16]. Since 2010, RISC-V has brought an open, modular ISA that can be customized without licensing fees. Its lightweight base instruction set, expanded as needed, allows for an elegant co-design of hardware and software, well matched to the parallel computing strengths of modern FPGAs. With the rise of machine learning—particularly deep learning using CNNs— the need for hardware that can handle large, complex datasets with high throughput and low latency has grown dramatically.

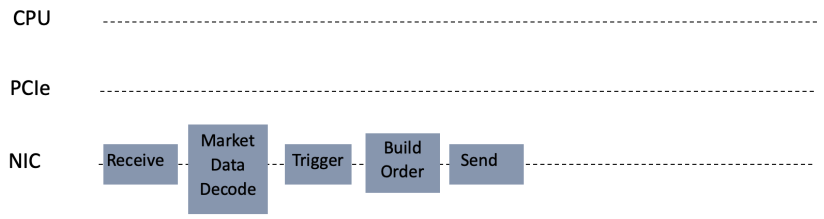
In finance, FPGAs are heavily employed in latency-sensitive tasks such as high-frequency trading, where every nanosecond is critical. By processing entire order flows directly on an FPGA-based NIC (Network Interface Card), data transfers through PCIe and CPU layers are minimized [15]. Figure 2a shows the conventional software-only pipeline, while Figure 2b depicts a more efficient FPGA-centric approach. By eliminating round trips to the CPU for market data and order construction, FPGA solutions reduce total latency and enable more deterministic behavior—key advantages in automated trading strategies.

In industrial control, FPGAs power real-time systems that demand deterministic responses and robust fault tolerance [18]. They are employed in sensor fusion, motor control, and signal filtering under harsh factory conditions. By embedding RISC-V cores alongside FPGA logic, manufacturers can combine high-level control tasks running on RISC-V with time-critical tasks implemented directly in hardware [3], reducing latency and boosting reliability. This hybrid approach is especially valuable in areas like robotics, where low-latency data analysis from LiDAR or stereo vision sensors enables instantaneous path planning and obstacle avoidance.

Similarly, aerospace and automotive industries rely on FPGA-based systems for



(a) Software only execution system



(b) FPGA NIC execution system

Figure 2: Comparison of software and FPGA [15]

onboard data processing. For instance, autonomous driving systems often merge inputs from cameras, radars, and LiDAR, requiring high-throughput hardware that can accomplish sensor fusion in real time. Integrating an open and extensible RISC-V core on the same FPGA platform provides a flexible control layer for advanced driver-assistance systems and safe automotive computation [4].

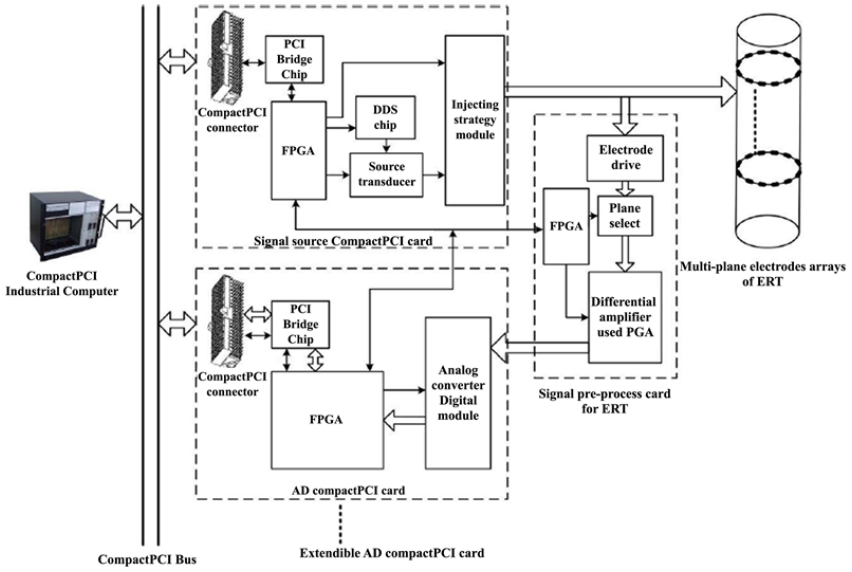


Figure 3: Architecture of the DAS based on CompactPCI Bus and FPGA [7]

FPGA technology has also become a key enabler for bridging high-performance computing with real-time data acquisition in industrial and medical applications. Figure 3 shows a typical architecture where FPGA modules interface with industrial computers via CompactPCI buses. This approach allows multi-channel data acquisition, high-speed signal processing, and local acceleration for computationally expensive operations like CNN-based inspection [7]. In medical imaging, FPGAs enhance reconstruction algorithms for MRI, CT, and ultrasound by exploiting massive parallelism to reduce latency and boost throughput. Integration with a RISC-V control core offers a unified hardware/software environment to coordinate resources and manage communication among various subsystems, thereby improving overall diagnostic accuracy [10].

1.3.2 Conclusion

From the initial emergence of microprocessors and their subsequent evolution into multicore, SoC, and domain-specific accelerators, it is evident that modern computing must efficiently balance performance, power, and flexibility. The open and extensible nature of RISC-V, paired with the reconfigurability of FPGAs, creates a powerful synergy—especially for computationally intensive tasks such as CNN inference. Recent industry applications, spanning finance, industrial control, robotics, and aerospace, underscore how FPGA-accelerated RISC-V solutions can simultaneously provide low latency, high throughput, and scalable hardware integration.

Against this backdrop, my project leverages a pipelined RISC-V CPU design enhanced with custom instructions tailored for CNN acceleration. By comparing single-cycle and pipelined implementations, I aim to demonstrate substantial performance gains in terms of reduced CPI and overall instruction throughput. The following sections detail my architectural choices, methodology, and experimental validations that reinforce the viability and adaptability of this FPGA-based RISC-V design.

1.4 Goals and Objectives

Goals

- **Develop a CNN-Oriented Five-Stage Pipelined RISC-V CPU:**
  Create a Verilog-based, five-stage pipeline (IF, ID, EX, MEM, WB) that can be seamlessly extended with custom instructions for Convolutional Neural Network (CNN) tasks. The overarching goal is to confirm that such a pipeline architecture can balance efficiency and correctness, even under data-intensive workloads.

- **Enhance Execution Through Custom Instructions:**
  Integrate specialized operations (e.g., matrix multiplication, vector addition) into

the RISC-V ISA. Demonstrate that incorporating these instructions can significantly reduce cycle counts for typical CNN inference tasks, while preserving baseline RISC-V compatibility.

- **Facilitate CNN-Driven Validation:**
  Employ real data from a trained CNN model—selecting a 64-line subset from a larger weight file—to test matrix-multiplication accuracy in both single-cycle and pipelined CPU variants. Show that the pipeline approach not only preserves correctness but also boosts performance in cycle count reduction.

- **Lay the Foundation for Future FPGA Deployment:**
  Although hardware implementation is deferred to future work, ensure that pipeline organization, fixed-point arithmetic, and hazard-handling strategies are readily transferable to an FPGA board (e.g., DE0-Nano). This establishes a blueprint for potential real-time neural-network acceleration in resource-constrained systems.

Objectives

- **Objective 1: Pipeline Construction and Hazard Mitigation**
  Partition the CPU into five well-defined pipeline stages and implement robust forwarding and stalling mechanisms. Verify—through functional simulation—that data and control hazards do not compromise correctness.

- **Objective 2: Implementation of CNN-Focused Instructions**
  Extend the base RISC-V ISA with instructions for matrix multiplication and vector addition. Formally define each instruction's opcode, operand usage, and modifications in the EX/MEM pipeline stages.

- **Objective 3: Comparative Testing with Single-Cycle Architecture**
  Construct a simpler single-cycle RISC-V processor as a performance baseline. Use identical CNN data inputs (converted into a 64-line `hex` file) to compare cycle counts, correctness, and simulated resource usage against the pipelined implementation.

- **Objective 4: Simulation and Verification of `hex` Data Processing**
  Incorporate the 64-line CNN-trained dataset into the test environment. Confirm both CPU versions produce matching outputs, and quantify the pipelined CPU's speedup versus the single-cycle reference when running the same CNN-focused instructions.

- **Objective 5: Establish a Pathway for Future Hardware Integration**
  Summarize design considerations for eventual FPGA-based deployment—e.g., clock frequency targets, memory interface design, and handling of fixed-point arithmetic for larger CNN workloads. Provide guidelines on how to extend this approach to additional custom instructions or more advanced CNN models.

2 Theoretical Background

This section presents three essential hardware modules in a RISC-V processor design: the Arithmetic Logic Unit (ALU), the Register File, and the Data Memory. I illustrate each module's conceptual schematic, a corresponding Verilog code snippet (shown as images), and the simulation waveforms that confirm correct functionality. In the following subsections, I highlight the main design principles rather than full code listings, focusing on how these components operate and interact.

2.1 Arithmetic Logic Unit (ALU)

2.1.1 ALU Structure and Code Overview

Figure 4 depicts a typical ALU schematic with two 32-bit inputs (`SrcA`, `SrcB`), a 3-bit control signal (`ALUControl`), and outputs `ALUResult` plus a `Zero` flag. When `ALUControl` indicates an arithmetic or logical operation (e.g., ADD, SUB, AND, OR, SLT), the ALU processes `SrcA` and `SrcB` accordingly.

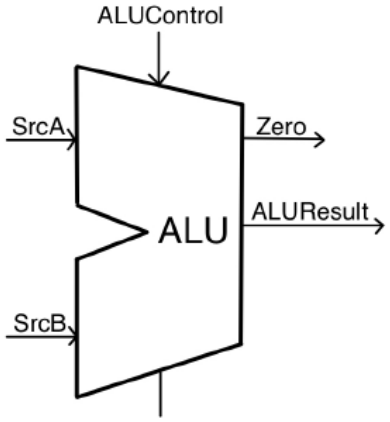


Figure 4: Schematic of the basic ALU

Figure 5 shows the corresponding Verilog code snippet for the ALU. Inside the `always @(*)` block, a `case` statement differentiates each operation: `3'b000` for addition (`SrcA + SrcB`), `3'b001` for subtraction, etc. Whenever `ALUResult` evaluates to zero, the `Zero` output is asserted high.

2.1.2 ALU Simulation and Results

I tested the ALU with a dedicated testbench that varies `ALUControl` and input operands. For example, when `ALUControl = 3'b000` and `SrcA = 10`, `SrcB = 5`, the ALU outputs

```
always @(*) begin

    case (ALUControl)
        3'b000: ALUResult = SrcA + SrcB;                          // ADD
        3'b001: ALUResult = SrcA - SrcB;                          // SUB
        3'b010: ALUResult = SrcA & SrcB;                          // AND
        3'b011: ALUResult = SrcA | SrcB;                          // OR
        3'b100: ALUResult = (SrcA < SrcB) ? 32'd1 : 32'd0;         // SLT
        default: ALUResult = 32'd0;
    endcase
end
```

Figure 5: Key Verilog code snippet for the ALU, illustrating the `always` block and `case` statement handling ADD, SUB, AND, OR, and SLT operations.

15. Similarly, subtraction (`3'b001`) with the same operands yields `5`. When `SrcA = 7` and `SrcB = 7`, the subtraction result is zero, driving `Zero` high. Figure 6 shows the ModelSim waveform for these scenarios, confirming that each control setting produces the expected result.

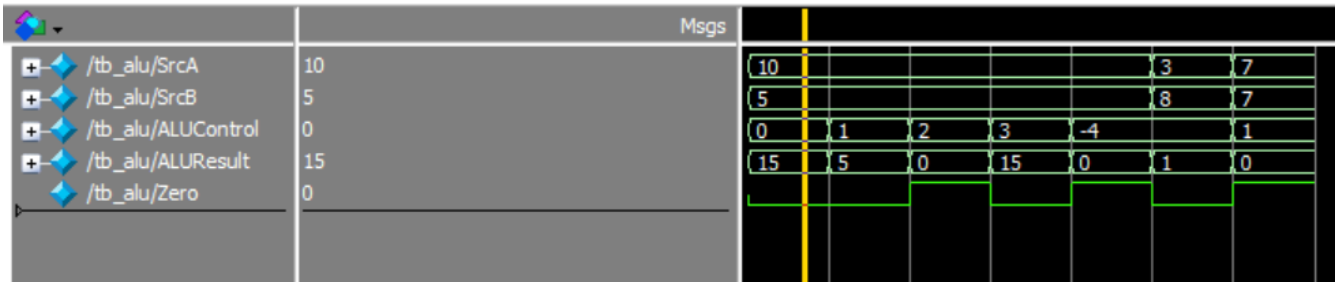


Figure 6: ALU simulation waveform. Each segment corresponds to a different `ALUControl` value. ADD (10+5=15), SUB (10-5=5), AND/OR with bitwise logic, and SLT comparison can be observed. When the result is zero, the `Zero` flag is asserted.

2.2 Register File

2.2.1 Register File Design and Code Overview

Figure 7 illustrates a multi-ported register file with two 32-bit read ports (`rs1`, `rs2`) and one 32-bit write port (`rd`). In RISC-V, `x0` is always zero, so any writes to register zero are discarded.

The main logic is shown in Figure 8, where an `always @(posedge clk or posedge rst)` block handles synchronous writes. On reset, all registers initialize to zero. During normal operation, the address `rd` is written with `writeData` on a rising clock edge, provided `RegWrite` is asserted and `rd` is not zero.

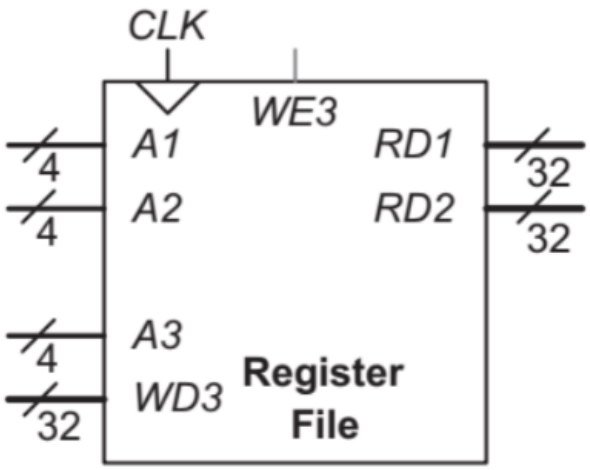


Figure 7: Register file architecture with two simultaneous read ports and a single write port.

```
always @(posedge clk or posedge rst) begin

    if (rst) begin
        for (i = 0; i < 32; i = i + 1) begin
            register_file_array[i] <= 32'b0;
        end
    end else begin
        if (RegWrite) begin

            if (rd != 5'd0) begin
                register_file_array[rd] <= writeData;
            end
        end
    end
end
```

Figure 8: Highlighted Verilog code for the register file. The `always` block resets all 32 registers to zero or writes new data to register `rd` if `RegWrite` is enabled.

2.2.2 Register File Simulation and Results

I verified basic read/write operations by instantiating the register file in a testbench. When `RegWrite` = 1 and `rd`=1, for instance, the value `0xAAAA_5555` is captured on the next clock edge. Reading from `rs1`=1 then returns that same data. Attempts to write to `rd`=0 confirm `x0` remains zero. Figure 9 displays the resulting waveform.

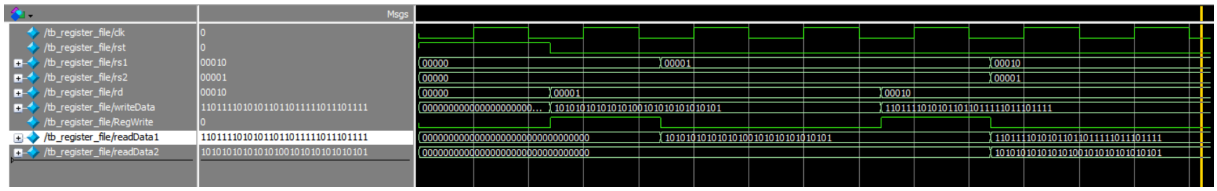


Figure 9: Register file waveform showing successful writes to nonzero registers and continuous reads on `rs1`, `rs2`. Register zero (`x0`) is never modified, adhering to RISC-V conventions.

2.3 Data Memory

2.3.1 Memory Interface and Operation

In a RISC-V CPU, data memory stores operands and intermediate results. Figure 10 depicts a typical interface with a 32-bit `Address`, 32-bit `Write data`, and a 32-bit `Read data` output. Two control signals, `MemWrite` and `MemRead`, indicate whether a store (SW) or load (LW) is taking place. The design can be synchronous or asynchronous, depending on FPGA resources and architectural preferences.

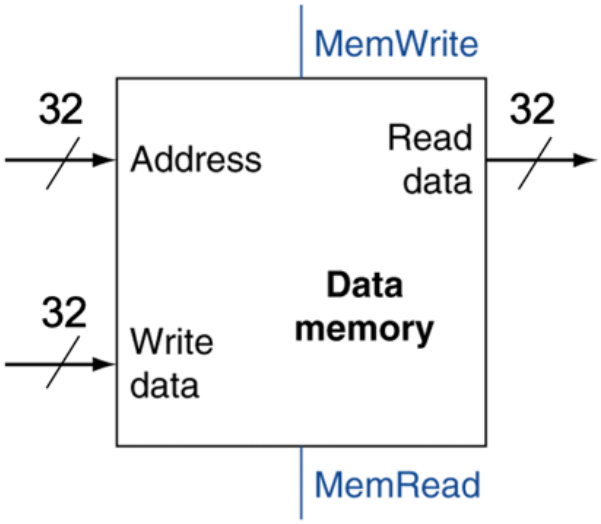


Figure 10: Data memory module with a 32-bit address, 32-bit write/read data lines, and control signals `MemWrite` and `MemRead` [5].

A typical load sequence might set `MemRead` high, place a valid address on `Address`, and then capture the memory's output in the next pipeline stage. For stores, `MemWrite` goes high while `Write data` and `Address` remain stable until the operation completes. In the following pipeline implementation, these signals coordinate with the ALU result (e.g., `SrcA + immediate`) to compute addresses for memory access.

Summary

In these sections above, I examined the design and verification of three core components in a RISC-V CPU:

- The **ALU**, which processes arithmetic and logic operations based on a 3-bit control signal.
- The **Register File**, featuring two read ports and one write port, ensuring `x0` remains constant at zero.
- The **Data Memory** interface, enabling load/store instructions via straightforward control signals (`MemWrite`, `MemRead`).

Each module was tested with ModelSim waveforms and produced expected results such as `10+5=15` in the ALU and correct read/write behavior in the register file. These building blocks form the foundation of the five-stage RISC-V pipeline discussed in subsequent sections.

2.4 Single-Cycle vs. Pipelined Architectures

Modern RISC-V or similar CPU designs can be realized in either a single-cycle implementation or a pipelined implementation, each with distinct performance and complexity trade-offs. In a *single-cycle* CPU, every instruction (from fetching the instruction in memory to writing back the result) completes in exactly one clock cycle. By contrast, in a *pipelined* CPU, the instruction processing stages overlap, akin to an assembly line.

- **Single-Cycle Design:** In a single-cycle CPU, each instruction runs through all five stages (Instruction Fetch, Decode, Execute, Memory, Write Back) in a single clock period. Because the processor's clock period must be long enough to support the slowest instruction [5], many simpler instructions waste time. Figure 11 ("Complete single-cycle processor") provides a high-level schematic, where the control logic orchestrates ALU operations, register file reads/writes, and memory accesses in one extended cycle per instruction. Although simpler to reason about, the design is often slower for complex instructions, yielding a clock cycle time that scales poorly with more complicated instructions like `LW` (load word).
- **Pipelined Design:** In a pipelined CPU, the classic five stages are divided into separate hardware units, and multiple instructions advance through these pipeline stages in parallel. Figure 12 ("Pipelined Datapath") shows the canonical five-stage pipeline structure, where each stage is separated by pipeline registers. While

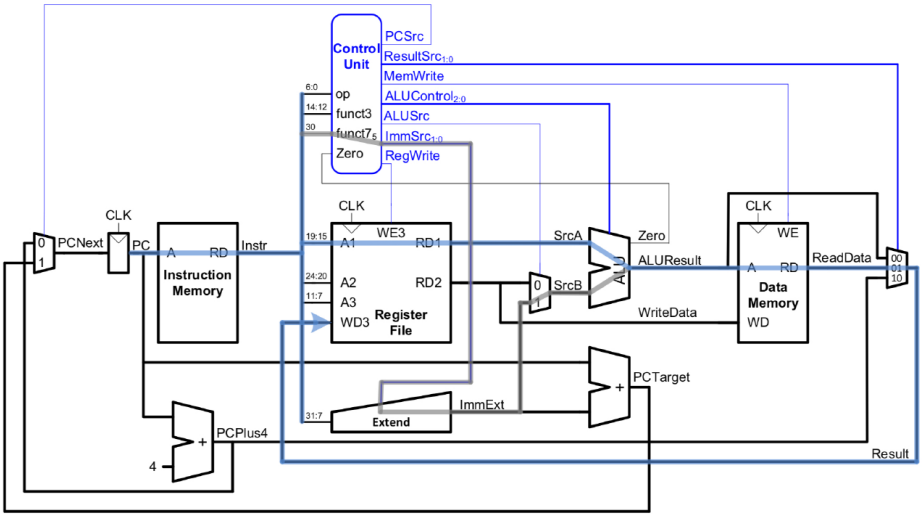


Figure 11: Complete single-cycle processor. All stages (IF, ID, EX, MEM, WB) happen in a single extended clock cycle. [5]

one instruction occupies the Execute (EX) stage, the next instruction can simultaneously occupy the Decode (ID) stage, etc. This arrangement can significantly improve the overall throughput (one instruction completing each cycle, after the pipeline is full). However, pipeline overhead arises from the need to handle hazards and to insert pipeline registers.

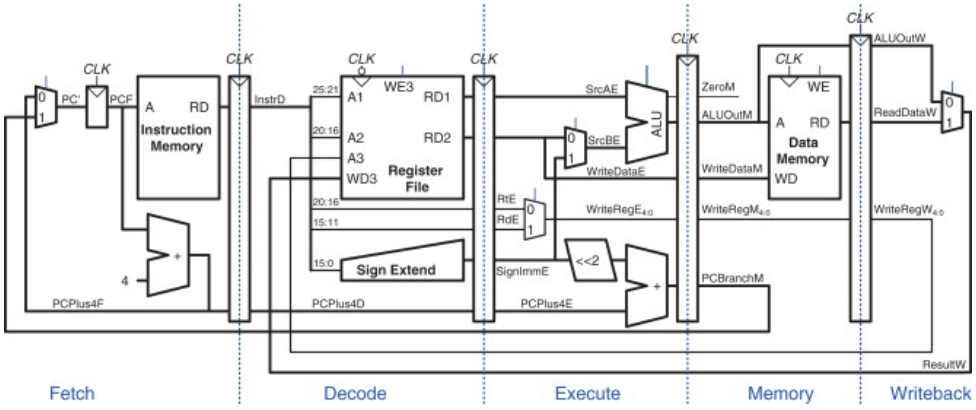


Figure 12: Pipelined datapath with five stages and pipeline registers (gray) between them. Instructions proceed concurrently through IF, ID, EX, MEM, and WB. [5]

Figure 13 offers a timing diagram comparison:

- *(a) Single-cycle timing* shows that the CPU finishes one entire instruction within one lengthy cycle, and the next instruction waits for the subsequent cycle.
- *(b) Pipelined timing* reveals how instructions overlap in time, each occupying different stages of the pipeline concurrently. After a short pipeline fill period,

the processor can complete nearly one instruction per clock cycle, substantially increasing the instruction throughput (instructions per second).

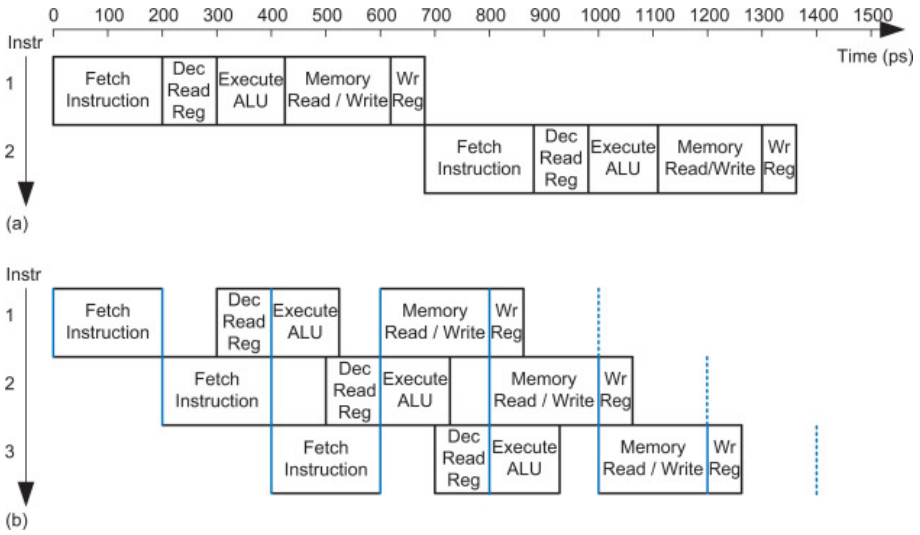


Figure 13: (a) Single-cycle timing vs. (b) pipelined timing. Note how the pipelined design overlaps different instructions in each stage, improving overall throughput. [5, 8]

The performance advantage of pipelining is often summarized by the *ideal CPI* (cycles per instruction) approaching 1. The actual CPI is usually slightly higher due to stalls, flushes, and hazard resolution. Nonetheless, pipelining remains foundational to modern CPU performance.

2.5 Control Signals and Their Role

Both single-cycle and pipelined processors rely on a dedicated control unit to generate various *control signals*, which steer data flow through the datapath. For instance:

- `RegWrite`: Enables writing the register file on the rising edge of the clock.
- `MemWrite`: Enables writing to data memory.
- `ALUSrc`: Selects whether the second ALU operand is a register file output or an immediate (extended) value.
- `MemtoReg`: Routes the result from data memory (for a load) versus the ALU output back into the register file.
- `PCSrc`: Decides whether the next PC comes from `PC+4` or from a branch target (for branches and jumps).
- `ALUControl`: Encodes the type of ALU operation (e.g. ADD, SUB, AND, OR, etc.).

Single-Cycle Control: In a single-cycle design (e.g., Figure 11), the control logic is typically a purely *combinational* block that examines the `opcode` and `funct` fields of the current instruction. It then asserts the correct mix of signals (`RegWrite`, `MemWrite`, etc.) so that the datapath executes that instruction in the same (and only) cycle.

Since all stages happen at once, each control signal must be valid throughout the entire extended cycle. A single-cycle control unit can be constructed by a decoder plus sub-decoders for ALU operations, resulting in a truth-table-based or HDL-based logic that directly sets each control line.

Pipelined Control: In a pipelined design (e.g., Figure 12), the control signals must be carried along with the instruction data from one stage to the next, because each stage may need different subsets of the signals. Typically, the ID (decode) stage generates all the signals, and pipeline registers pass them to EX, MEM, and WB stages. The `MemWrite` control, for example, is only needed in the MEM stage, so it flows through pipeline registers from ID until it arrives at MEM [5, 8].

Handling branches and other instructions that change flow is more complex, because the control logic must detect the branch *early enough* and choose whether to fetch new instructions or flush pipeline stages that fetched unwanted instructions. Hence, the pipeline control must be mindful of the pipeline timing and is typically more elaborate than in the single-cycle case.

2.6 Hazard Handling

When instructions overlap in a pipelined CPU, various conflicts called *hazards* arise. Figure 14 ("Abstract pipeline diagram illustrating hazards") shows an example, highlighting possible collisions between pipeline stages. I categorize hazards as follows:

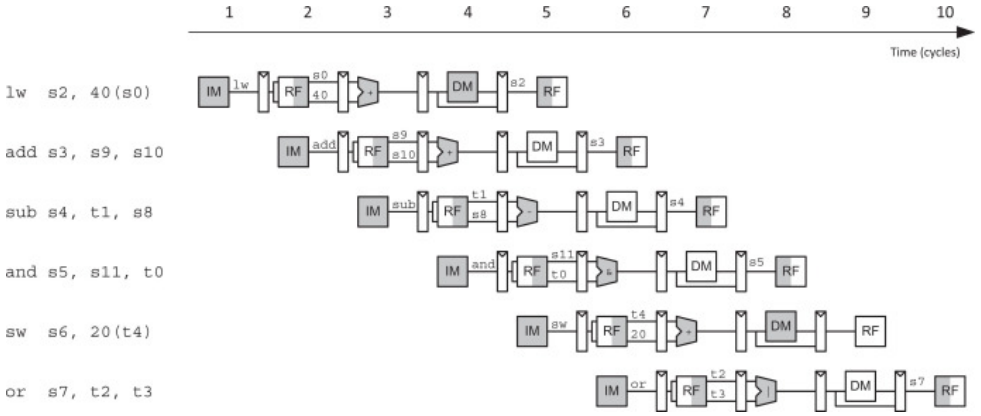


Figure 14: Abstract pipeline diagram illustrating hazards. Overlapping instructions can conflict on registers (data hazards), addresses (control hazards), or hardware resources (structural hazards) [5].

Data Hazards

A data hazard occurs when one instruction depends on a value that has not yet been written back by a preceding instruction. For instance, if `I1` writes to register `x5` in the WB stage at the end of cycle t, but `I2` tries to read `x5` in the ID stage *during* cycle t, `I2` will read an old, incorrect value. Common solutions include:

- **Forwarding** (or bypassing): The pipeline routes the result directly from the EX or MEM stage of `I1` to the input of the ALU for `I2`'s EX stage, thereby bypassing the register file's older contents.
- **Stalling**: The pipeline inserts a no-operation (NOP) instruction or *bubble* so that the instruction needing the operand waits an extra cycle or two [5, 8]. This approach negatively impacts throughput, so modern designs use forwarding wherever possible and only resort to stalls when absolutely needed (e.g. load-use hazards).

Control Hazards

A control hazard (or branch hazard) arises when the CPU does not yet know whether a branch is taken, but it has already fetched subsequent instructions [5, 8]. If the branch is taken, some incorrectly fetched instructions must be flushed from the pipeline. Approaches include:

- **Stall until branch resolution**: The simplest method, but stalls degrade performance.
- **Early branch resolution**: The CPU moves branch determination to an earlier pipeline stage (e.g., ID), reducing or removing the hazard penalty.
- **Branch prediction**: The CPU predicts whether the branch is likely taken or not, continuing to fetch along that path. If the prediction is wrong, it flushes the mis-fetched instructions.

Structural Hazards

Although less common in a fully *Harvard* pipeline (where instruction and data memories are separate), a structural hazard can occur if two stages need the same hardware resource simultaneously [5, 8]. For example, if the memory stage (MEM) and the instruction fetch stage (IF) share a single memory module, a structural hazard arises any time an instruction tries to perform a data access in the same cycle that another instruction needs to fetch an instruction [13]. Workarounds may involve duplicating resources or scheduling memory accesses to avoid conflicts.

In summary, hazard handling is an integral part of pipeline design. The hazard unit detects hazards by comparing register fields among instructions in different stages. When it identifies a data or control hazard, the pipeline either forwards data values, stalls, or flushes instructions. Although this extra logic adds complexity compared to a single-cycle design, pipelining drastically improves instruction throughput in most real workloads.

3 Methodology

This section describes my methodology in three major steps: (1) introducing the MNIST dataset and the core deep-learning techniques used (including regularization methods and activation functions), (2) training and quantizing a Convolutional Neural Network (CNN) to generate integer-weight data, and (3) leveraging those integer parameters (in a `hex` file) to validate matrix multiplication on a single-cycle processor and a five-stage pipelined processor. Although this subsection focuses on the data-generation process, subsequent parts will discuss how each CPU design is implemented and tested.

3.1 MNIST and Machine-Learning Techniques

3.1.1 MNIST Dataset

MNIST is a well-established benchmark dataset for digit recognition, containing 60,000 training images and 10,000 test images of handwritten digits 0–9 [11]. Each image is 28×28 pixels in grayscale, making it a convenient starting point for demonstrating convolutional architectures and various network optimizations. Despite its relatively simple nature, MNIST remains a standard reference for validating fundamental design concepts in deep learning systems.

3.1.2 Regularization and Activation Functions

To improve generalization and prevent overfitting, various regularization mechanisms are common in CNNs:

- **Batch Normalization (BN):** Introduced by Ioffe and Szegedy [9], BN normalizes intermediate feature map activations, often accelerating training and stabilizing gradients.
- **Dropout:** Randomly disabling a fraction of neurons during training effectively combats overfitting, encouraging more robust feature extraction [9].

I also use the Rectified Linear Unit (ReLU) activation, $\mathrm{ReLU}(x) = \max(0, x)$, which is simple and prevents vanishing gradients in deeper architectures.

3.1.3 Model Architecture and Training

Building upon these ideas, In the Python environment, I trained a CNN on MNIST using two convolutional blocks (each with ReLU, batch normalization, and max-pooling), followed by dropout. After flattening, a 128-neuron fully connected layer (with ReLU,

batch normalization, dropout) precedes a final softmax output layer for digit classification. I employ `Adam` as the optimizer and `sparse_categorical_crossentropy` as the loss function, typically running for 10 epochs at a batch size of 128. In practice, this setup quickly converges to $\approx 98\%$ validation accuracy on MNIST.

```
model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),



    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),


    Flatten(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

Figure 15: Core snippet of the Keras CNN code, demonstrating convolutional layers with batch normalization, dropout, and a final fully connected block.

Figure 16 displays sample convergence curves (accuracy/loss) across epochs, as well as example MNIST digits. The fundamental model-building code is partially shown in Figure 15, reflecting the essential layers and hyperparameters used.

3.1.4 Post-Training Quantization and Hex Data Generation

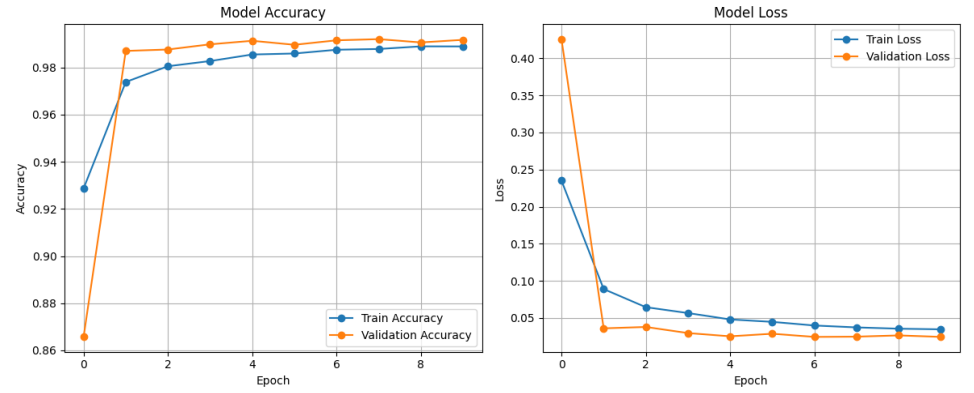
To align with embedded application constraints, I convert the trained floating-point model to 8-bit integer precision via TensorFlow Lite (`tf.lite.TFLiteConverter`) optimizations [9]. In particular:

1. **Quantization:** Scales and zero-points are computed for each layer's weights and activations, reducing them to 8-bit.

2. **Export to `.tflite`:** The final quantized model is saved, enabling on-device inference on microcontrollers [17].

3. **Dump integer arrays to `hex` file:** I parse each integer parameter (weights, biases) and write them line-by-line in hexadecimal (`cnn_weights.hex`). This dataset accurately reflects the numeric distributions typical of CNN inference.

Figure 17 outlines the post-quantization layer structure, including the shapes of convolutional filters and dense layer parameters. I select only the first 64 lines of the `hex` file to represent 8×8 matrices used in the subsequent matrix multiplication tests.




```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.8568 - loss: 0.4780 - val_accuracy: 0.8657 - val_loss: 0.4258
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.9723 - loss: 0.0946 - val_accuracy: 0.9870 - val_loss: 0.0359
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.9801 - loss: 0.0655 - val_accuracy: 0.9876 - val_loss: 0.0378
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.9833 - loss: 0.0543 - val_accuracy: 0.9898 - val_loss: 0.0296
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9856 - loss: 0.0490 - val_accuracy: 0.9913 - val_loss: 0.0252
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9865 - loss: 0.0428 - val_accuracy: 0.9896 - val_loss: 0.0287
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9874 - loss: 0.0397 - val_accuracy: 0.9915 - val_loss: 0.0244
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9871 - loss: 0.0384 - val_accuracy: 0.9920 - val_loss: 0.0247
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9895 - loss: 0.0337 - val_accuracy: 0.9906 - val_loss: 0.0264
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9892 - loss: 0.0329 - val_accuracy: 0.9917 - val_loss: 0.0244
Test loss: 0.0244
Test accuracy: 0.9917
```

Figure 16: Top: Illustration off training/validation accuracy and loss over 10 epochs for MNIST.
Bottom: Sample grayscale MNIST images for digits 0–9.

Advantages of Using Real Quantized Data

- **Realistic Numeric Ranges:** CNN parameters exhibit typical distributions that test integer math logic in a manner more robust than trivial or random data.
- **Practical Verification:** By comparing CPU-computed matrix products to the TFLite or Python reference, I guarantee correctness in a scenario closely resembling embedded deep learning.
- **Uniform 8-bit Format:** The `hex` data replicate embedded-friendly integer formats, which is key for validating both pipeline logic and memory access patterns under realistic loads.

Both single-cycle and pipelined designs load the same quantized `hex` data for matrix multiplication. Using the top 64 lines to form eight 8×8 integer matrices, I demonstrate how the designs handle typical embedded inference parameters. Finally, by comparing their hardware results against a Python/TFLite reference, I confirm functional correctness and set the stage for performance discussion.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
batch_normalization (BatchNormalization)	(None, 28, 28, 32)	128
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
dropout (Dropout)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_1 (Dropout)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 128)	401,536
batch_normalization_2 (BatchNormalization)	(None, 128)	512
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 422,538 (1.61 MB)
Trainable params: 422,090 (1.61 MB)
Non-trainable params: 448 (1.75 KB)

Figure 17: Layer-wise summary after quantization, including convolutional and fully connected parameters.

3.2 Single-Cycle Microprocessor

In this section, I describe a *single-cycle* CPU design—one that completes instruction fetch, decode, ALU execution, and result write-back all in one clock cycle. Although my provided code focuses primarily on the ALU logic (including matrix multiplication, max-pooling, and basic arithmetic), I also outline how it fits into a broader single-cycle structure.

3.2.1 High-Level Design and Data Flow

Figure 18 offers a conceptual top-level view of a single-cycle CPU. Because the code snippet I focus on deals chiefly with the ALU, the figure includes registers, memory blocks, and a control path for completeness, but the associated modules may be stubs or simpler placeholders in the actual code:

- **Instruction Fetch:** The CPU reads a 16- or 32-bit instruction from a small instruction memory (InstrMem) indexed by `PC`.
- **Control Logic:** Decodes the instruction's operation field (`ID_instr` in my code) to enable either matrix multiplication, max-pooling, or standard arithmetic like addition and subtraction.

- **Register File (not fully shown in code):** Would typically provide sources for matrix inputs (e.g., `matrixA_ij` and `matrixB_ij`). In practice, these could be loaded from memory or written by earlier instructions in one cycle.
- **Single-Cycle ALU (main code):** Combines 2×2 matrix multiply, pooling, add/-sub, all triggered by the instruction opcode.
- **Result Write-Back:** In a single-cycle design, the results (e.g., `matrixpXX`) would be written to registers immediately in the same clock cycle or remain in dedicated output registers.

Because everything happens in one clock, the CPU's timing budget must accommodate even the costliest operation (like 2×2 matrix multiply) within one cycle.

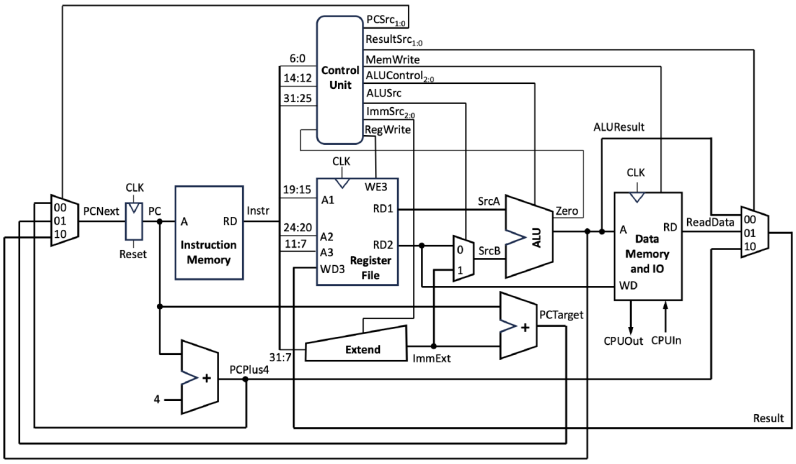


Figure 18: Conceptual single-cycle CPU. Though my code snippet focuses on the ALU, the design typically includes instruction memory, control, registers, and (optionally) data memory in one cycle [5].

3.2.2 ALU Core and Custom Matrix Multiply Logic

My central module, illustrated conceptually in Figure 19, is the single-cycle ALU. It exposes signals for 2×2 matrix inputs (`matrixA_ij`,`matrixB_ij`), output product registers (`matrixpXX`), and a limited opcode interface (`ID_instr[15:0]`). Within each clock cycle:

1. **Instruction Decode:** The CPU interprets `ID_instr` as one of several parameter constants (e.g., `ALU_MATMUL = 16'd1`, `ALU_ADD = 16'd2`, etc.).
2. **Matrix or Arithmetic Execution:**

- **Matrix Multiply** (`ALU_MATMUL`):

$$\texttt{matrixp00} \leftarrow (A_{1,1} \times B_{1,1}) + (A_{1,2} \times B_{2,1})$$
$$\texttt{matrixp11} \leftarrow (A_{2,1} \times B_{1,2}) + (A_{2,2} \times B_{2,2}).$$

  The four partial products are computed combinationally. The results latch into `matrixp00, matrixp01, matrixp10, matrixp11` when the clock rises.

- **Add/Sub** (`ALU_ADD`, `ALU_SUB`): Each element is added or subtracted, writing results back to the same `matrixpXX` outputs.

- **Max-Pooling** (`ALU_MPOOL`): Compares multiple partial product results to select the maximum. Although typical pooling in CNNs is done over activation maps, here it is integrated into the single-cycle ALU for demonstration.

3. **Register/Output Update:** The final outputs are stored in the ALU's internal registers. Additionally, a `cycle_count` increments to observe how many operations have completed.

3.2.3 Single-Cycle Execution Sequence for Matrix Multiply

When `ID_instr = ALU_MATMUL`:

1. **Fetch & Decode (Conceptually):** The CPU obtains `ID_instr` from an instruction memory or hardcoded register. The control logic recognizes `16'd1` as matrix multiply.

2. **Register Inputs:** The 2×2 matrix elements `matrixA_ij`,`matrixB_ij` are loaded from the register file or memory prior.

3. **Compute** in *one cycle*: `matrixpXX` are assigned partial product sums, e.g., `matrixp00` = `A_11*B_11 + A_12*B_21`.

4. **Write Outputs**: At the rising clock edge, the final matrix product becomes visible on `matrixp00`, `matrixp01`, `matrixp10`, `matrixp11`, and `cycle_count` increments by 1.

Hence, a 2×2 matrix multiply completes in a single cycle. This matches the single-cycle CPU's philosophy: *no pipeline, no multi-cycle instruction flow*.

```
module single_cycle_alu (
  input                     clk        ,
  input                     rst_n      ,
  input             [15:0] ID_instr    ,

  input      signed [31:0] matrixA_11 ,
  input      signed [31:0] matrixA_12 ,
  input      signed [31:0] matrixA_21 ,
  input      signed [31:0] matrixA_22 ,

  input      signed [31:0] matrixB_11 ,
  input      signed [31:0] matrixB_12 ,
  input      signed [31:0] matrixB_21 ,
  input      signed [31:0] matrixB_22 ,
  output reg signed [31:0] matrixp00  ,
  output reg signed [31:0] matrixp01  ,
  output reg signed [31:0] matrixp10  ,
  output reg signed [31:0] matrixp11  ,
  output reg signed [31:0] cycle_count
);
reg signed [31:0] maxpooling;
  parameter ALU_MATMUL = 16'd1,
    ALU_ADD    = 16'd2,
    ALU_MPOOL  = 16'd3,
    ALU_SUB  = 16'd4;
```

Figure 19: ALU snippet for single-cycle logic. Each instruction code triggers a distinct operation, such as matrix multiplication or max-pooling.

3.2.4 Tying Into the Larger Single-Cycle System

Although my code snippet largely shows ALU-specific logic, in a typical single-cycle CPU:

- **Register File Source:** The matrix elements (`matrixA_ij`, `matrixB_ij`) might be read from the register file or from memory in the same cycle, if `mem_read` was asserted.
- **Instruction PC:** The program counter increments by 4 unless a branch condition is met. If `ID_instr` was recognized as `ALU_MATMUL`, the CPU then goes ahead and executes it directly.
- **Memory:** For load/store instructions, the CPU also reads or writes data memory in parallel, using a single-cycle approach. In matrix multiply, I might not require a separate memory access *that same cycle*, but in general single-cycle designs must support it combinationally as well.

Thus, the `single_cycle_alu` module is integrated into a typical single-cycle architecture: the *ID_instr* comes from the CPU's instruction memory, the inputs come from register/memory reads, and the final results are either latched back or remain in dedicated output registers at the end of the clock.

3.2.5 Relevance to CNN Hex Data Testing

By loading real quantized CNN weights into `matrixA_ij` and `matrixB_ij` signals, the user can trigger the `ALU_MATMUL` instruction repeatedly (with different sub-blocks of the model parameters) to verify the computed results. Each instruction completes in one cycle, and `cycle_count` can be used to track how many matrix operations have executed. This method offers a simple demonstration of hardware correctness in an environment free from pipelining complexities.

Overall, the `single_cycle_alu` code ensures that matrix multiplication (or other specialized instructions) are dispatched and finalized in a single clock tick, consistent with a single-cycle CPU's straightforward, albeit timing-heavy, approach.

3.3 Pipelined Microprocessor

In this section, I present a five-stage pipelined processor that coordinates matrix multiplication (and related operations) across *IF, ID, EX, MEM,* and *WB* stages. While not all hazards are fully resolved (due to the complexity of large matrix operations), partial hazard handling is included to demonstrate how instructions flow. I end with a discussion of the `top` module, which unifies all stages.

3.3.1 Top-Level Pipeline Structure (`top` Module)

Figure 20 illustrates a conceptual overview of the entire pipeline. The `top` module instantiates each stage, wiring their inputs and outputs in series. Its interface typically includes:

- **Clock & Reset**: Common signals across modules for synchronous updates and initialization.
- **cmd_address / cmd_valid**: Allows external logic (or a testbench) to feed addresses for the IF stage.
- **Final Output** (`reg_file` or a parallel bus): Collects the computed matrix results at the WB stage.
- **Done Signal**: Indicates when the pipeline has completed processing the current instruction, e.g., a matrix multiply.

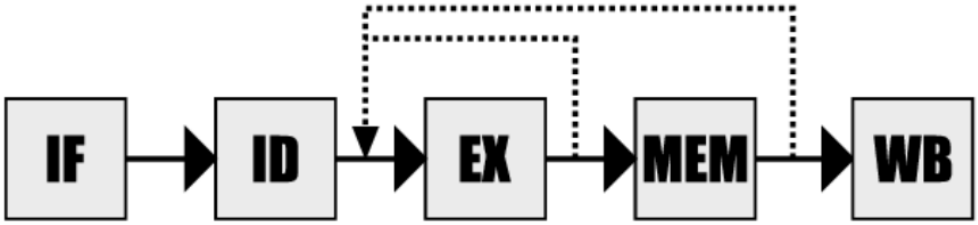


Figure 20: Overall pipeline. The `top` module connects IF → ID → EX → MEM → WB, handling matrix data from hex and returning partial products or results.

Top Module Flow:

1. **IF_Stage** reads `cmd_address`, fetches `IF_instr`, and sets `IF_valid`.
2. **ID_Stage** transforms `IF_instr` into `ID_instr`, preserving `ID_valid`.
3. **EX_Stage** interprets `ID_instr` as a possible matrix multiply and asserts `matrixpXX` outputs upon completion, toggling `single_done`.
4. **MEM_Stage** uses `address` from EX to load matrix A/B from `cnn_weights.hex`.
5. **WB_Stage** captures final partial products plus a cycle counter, providing them as `reg_file` and letting the `top` module set `done`.

The top module thus orchestrates each pipeline stage's signals, bridging them so that matrix computations can flow through neatly.

3.3.2 IF Stage: Instruction Fetch

The **IF_Stage** fetches a 32-bit instruction from an array `source_instr[]` using `cmd_address`.

- `IF_instr` is the instruction output.
- `IF_valid` goes high whenever a valid instruction is fetched.
- `IF_address` echoes `cmd_address` for debugging or pipeline reference.

3.3.3 ID Stage: Instruction Decode

In the **ID_Stage**, the pipeline reorganizes bits from `IF_instr` to form `ID_instr`, passing forward a `ID_valid` signal. A typical RISC-V pipeline might parse `opcode`, `funct3`, `funct7`. Here, I adopt a simpler reordering to support custom 16-bit segments for matrix ops:

```
initial begin
  $readmemh("C:/Users/86182/Desktop/lab/lab12/instr.hex",source_instr);
end


always @(posedge clk or posedge reset) begin
  if (reset) begin
    IF_instr <= 32'd0;
  end else
    begin
    IF_instr   <= source_instr[cmd_address];
    IF_valid   <= cmd_valid;
    IF_address <= cmd_address;
  end

  end
```

Figure 21: IF_Stage. Receives external `cmd_address`, returns `IF_instr` from `source_instr[]` memory.

```
module ID_Stage (
   input  wire         clk     ,
   input  wire         reset   ,
   input  reg  [31:0] IF_instr,
   input  reg   IF_valid,
   output reg  [31:0] ID_instr,
   output reg   ID_valid
);

   always @(posedge clk or posedge reset) begin
     if (reset) begin
         ID_instr<=32'd0;
         ID_valid<=0;
     end else  begin
         ID_instr<={IF_instr[15:0],IF_instr[31:16]};
         ID_valid<=IF_valid;

     end
   end


endmodule
```

Figure 22: ID_Stage. Re-maps `IF_instr` into `ID_instr` for custom operation codes.

3.3.4 MEM Stage: Loading Matrix Data from Hex

MEM_Stage (`pipeline_mem.png`) obtains 2×2 matrices from `cnn_weights.hex`:

- A 64-element array `source_data[]` is read once at `initial`.
- `address[2:0]` selects 1 of 8 blocks, each containing 8 lines (4 lines for matrix A, 4 lines for matrix B).
- `matrixA_ij,matrixB_ij` are latched on the rising edge if `data_valid=1`.

This stage effectively mimics the standard load operation but specialized for chunk-based matrix loading.

```
module MEM_Stage (
  input  wire        clk       ,
  input  wire        reset     ,
  input       [31:0] address   ,
  input              data_valid,
  output reg signed [31:0] matrixA_11,
  output reg signed [31:0] matrixA_12,
  output reg signed [31:0] matrixA_21,
  output reg signed [31:0] matrixA_22,
  output reg signed [31:0] matrixB_11,
  output reg signed [31:0] matrixB_12,
  output reg signed [31:0] matrixB_21,
  output reg signed [31:0] matrixB_22
);

  reg  signed [31:0] source_data[63:0];

// data
  initial
    begin
      $readmemh("C:/Users/86182/Desktop/lab/lab12/cnn_weights.hex",source_data);
    end
```

Figure 23: MEM_Stage. `source_data[]` is indexed by `address[2:0]` to produce 2×2 matrix pairs for the EX stage.

3.3.5 EX Stage: Matrix Multiply Execution

The **EX_Stage** (`pipeline_ex.png`) is central for matrix multiplication. On each rising clock:

- The module checks `flow_cnt` (2 bits) to step from waiting to actual execution.
- `ID_instr[31:16]` is used to identify `ALU_MATMUL`, `ALU_MPOOL`, `ALU_ADD`, `ALU_SUB`.
- If `ALU_MATMUL`, partial products for 2×2 multiplication are computed:

$$\begin{aligned} \texttt{matrixp00} &\leftarrow (A_{11} \times B_{11}) + (A_{12} \times B_{21}), \\ \texttt{matrixp01} &\leftarrow (A_{11} \times B_{12}) + (A_{12} \times B_{22}), \\ &\dots \end{aligned}$$

- `single_done` is raised once the multiply completes, enabling the pipeline to record these outputs in WB.
- `data_valid` interacts with `flow_cnt` to ensure the EX stage sees correct matrix data from MEM.

3.3.6 WB Stage: Final Write-Back & Cycle Counting

Finally, the **WB_Stage** (`pipeline_wb.png`) collects the four partial products plus a `cycle_cnt`:

```
ALU_MATMUL :
  begin
    flow_cnt    <= 2'd0;
    single_done <= 1'd1;
    matrixp00   <= matrixA_11*matrixB_11+matrixA_12*matrixB_21;
    matrixp01   <= matrixA_11*matrixB_12+matrixA_12*matrixB_22;
    matrixp10   <= matrixA_21*matrixB_11+matrixA_22*matrixB_21;
    matrixp11   <= matrixA_21*matrixB_12+matrixA_22*matrixB_22;
  end
ALU_ADD :
  begin
    flow_cnt    <= 2'd0;
    single_done <= 1'd1;
    matrixp00   <= matrixA_11+matrixB_11;
    matrixp01   <= matrixA_12+matrixB_12;
    matrixp10   <= matrixA_21+matrixB_21;
    matrixp11   <= matrixA_22+matrixB_12;
  end
ALU_MPOOL :
  begin
    flow_cnt    <= 2'd0;
    single_done <= 1'd1;
    if(((matrixA_11*matrixB_11+matrixA_12*matrixB_21)>=(matrixA_11*matrixB_12+matrixA_12*matrixB_22))&&
      ((matrixA_11*matrixB_11+matrixA_12*matrixB_21)>=(matrixA_21*matrixB_11+matrixA_22*matrixB_21))&&
      ((matrixA_11*matrixB_11+matrixA_12*matrixB_21)>=(matrixA_21*matrixB_12+matrixA_22*matrixB_22)))
    maxpooling <= matrixA_11*matrixB_11+matrixA_12*matrixB_21;
    else if(((matrixA_11*matrixB_12+matrixA_12*matrixB_22)>=(matrixA_11*matrixB_11+matrixA_12*matrixB_21))&&
      ((matrixA_11*matrixB_12+matrixA_12*matrixB_22)>=(matrixA_21*matrixB_11+matrixA_22*matrixB_21))&&
      ((matrixA_11*matrixB_12+matrixA_12*matrixB_22)>=(matrixA_21*matrixB_12+matrixA_22*matrixB_22)))
    maxpooling <= matrixA_11*matrixB_12+matrixA_12*matrixB_22;
    else if(((matrixA_21*matrixB_11+matrixA_22*matrixB_21)>=(matrixA_11*matrixB_11+matrixA_12*matrixB_21))&&
      ((matrixA_21*matrixB_11+matrixA_22*matrixB_21)>=(matrixA_11*matrixB_12+matrixA_12*matrixB_22))&&
      ((matrixA_21*matrixB_11+matrixA_22*matrixB_21)>=(matrixA_21*matrixB_12+matrixA_22*matrixB_22)))
    maxpooling <= matrixA_21*matrixB_11+matrixA_22*matrixB_21;
    else if(((matrixA_21*matrixB_12+matrixA_22*matrixB_22)>=(matrixA_11*matrixB_11+matrixA_12*matrixB_21))&&
      ((matrixA_21*matrixB_12+matrixA_22*matrixB_22)>=(matrixA_11*matrixB_12+matrixA_12*matrixB_22))&&
      ((matrixA_21*matrixB_12+matrixA_22*matrixB_22)>=(matrixA_21*matrixB_11+matrixA_22*matrixB_21)))
    maxpooling <= matrixA_21*matrixB_12+matrixA_22*matrixB_22;
    else
      maxpooling <= maxpooling;
  end
  ALU_SUB :
  begin
    flow_cnt    <= 2'd0;
    single_done <= 1'd1;
    matrixp00   <= matrixA_11-matrixB_11;
    matrixp01   <= matrixA_12-matrixB_12;
    matrixp10   <= matrixA_21-matrixB_21;
    matrixp11   <= matrixA_22-matrixB_22;
  end
```

Figure 24: EX_Stage. Checks `ID_instr[31:16]` for matrix multiply (`ALU_MATMUL`), addition, or pool. Partial hazard is embedded via `flow_cnt` state machine.

- The partial products `matrixp00,p01,p10,p11` are latched into `reg_file`, along with `cycle_cnt`.
- Once `single_done=1`, the stage indicates the matrix operation completed.
- If another instruction is in flight, the pipeline can proceed, but again a full hazard solution is not shown.

3.3.7 Partial Hazard Logic

While a typical five-stage pipeline requires thorough hazard detection (for data and control hazards) and forwarding paths, my current design opts for a simpler two-module approach—*Forwarding_Unit* and *Hazard_Detection_Unit*—to demonstrate how

```
reg [7:0] cycle_cnt ;
reg       write_done;
always @(posedge clk or posedge reset) begin
  if (reset) begin
    reg_file <= 136'd0;
  end
  else begin
    reg_file   <= {cycle_cnt,matrixp00,matrixp01,matrixp10,matrixp11};
    write_done <= single_done;
  end
end
always @(posedge clk or posedge reset) begin
  if (reset) begin
    cycle_cnt <= 8'd0;
  end
  else if(write_done == 1'd1)
    cycle_cnt <= 8'd0;
  else
    cycle_cnt <= cycle_cnt+1'd1;
end
```

Figure 25: WB_Stage. Latches partial product outputs into `reg_file`, increments or resets `cycle_cnt` around the `single_done` signal.

minimal logic can prevent incorrect data usage. Figures 26 and 27 show high-level representations of these modules:

```
module Forwarding_Unit (
  input      [31:0] address   ,
  input             data_valid,
  input      [31:0] ID_instr  ,
  input             ID_valid  ,
  output reg [ 1:0] ForwardA  ,
  output reg [ 1:0] ForwardB
);

  always @(*) begin
    if ((data_valid) && (address != 0) )
      ForwardA = 2'b10;
    else
      ForwardA = 2'b00;
  end


  always @(*) begin
    if ((ID_valid) && (ID_instr != 0) )
      ForwardB = 2'b10;
    else
      ForwardB = 2'b00;
  end
endmodule
```

Figure 26: Forwarding_Unit conceptual diagram. Determines whether to route alternate data paths (ForwardA/B) if valid signals and addresses indicate new results are ready.

Forwarding_Unit. A typical pipeline might match the destination register of an earlier instruction still in EX/MEM or MEM/WB against the source registers of the current instruction. In my simplified model, the *Forwarding_Unit* looks at two conditions:

- Whether a prior stage has set a *data_valid* signal with a nonzero *address*, causing ForwardA to assert for the ALU's first operand.

- Whether the ID stage is valid (*ID_valid*) with a nonzero *ID_instr*, causing ForwardB to assert for the ALU's second operand.

When active, each forward signal can direct data from the earlier stage to the current EX stage in place of the default path. This partial approach is sufficient for the 2×2 matrix flow, but not a complete solution for register-based pipelines with multiple potential forwarding sources.

```
module Hazard_Detection_Unit (
  input      [31:0] address   ,
  input             data_valid,
  input      [31:0] ID_instr  ,
  input             ID_valid  ,
  output wire       Stall
);

  assign Stall = ((address==ID_instr)&&(ID_valid==data_valid)) ? 1'b1 : 1'b0;
endmodule
```

Figure 27: Hazard_Detection_Unit conceptual diagram. Asserts a `Stall` signal if certain conditions (e.g., matching address/instruction) indicate a conflict requiring the pipeline to pause.

Hazard_Detection_Unit. In parallel, the *Hazard_Detection_Unit* checks whether the same address is in use across two stages (indicated by *address* == *ID_instr*) and whether both the data and instruction are valid at once. If so, a `Stall` is raised for one cycle. Conceptually, this prevents the CPU from overwriting or reading stale matrix data when the EX stage and MEM stage might conflict.

Limitations and Next Steps. Because the design is geared toward a specialized matrix multiply instruction, these modules primarily ensure:

- The EX stage receives the correct matrix block from MEM before performing multiplication.
- If the same address/instruction is reissued (or an overlapping valid signal occurs), the pipeline stalls to avoid corruption.

While this partial scheme can handle simple 2×2 matrix operations, a full hazard solution would require:

- Register-based forwarding (matching *rs1/rs2* to *ex_rd*, *mem_rd*, *wb_rd*) for typical arithmetic instructions.
- Dedicated load-use stall detection to handle load instructions followed immediately by dependent instructions.

- Branch hazard resolution and pipeline flush for control instructions.

Nevertheless, even in its simplified form, the partial hazard mechanism suffices for demonstration: it prevents the pipeline from unintentionally using half-updated data or skipping valid results when performing repeated matrix multiplications from the `cnn_weights.hex` file.

3.3.8 Implications for Matrix Multiply and Pooling

While these modules conceptually solve typical data hazards, incorporating them into the matrix multiplication flow can be nontrivial. For a 2×2 multiply that finishes within a single EX cycle, I must verify if the subsequent instruction can safely read the partial products or if an additional stall is required:

- **Forwarding to the same EX cycle** is not typical, because the partial products appear only at the end of that EX cycle. Usually, I forward from EX/MEM in the *next* cycle.
- **Load-use stall** is still necessary if the matrix data was loaded from memory the prior cycle.

Hence, although `StallUnit` and `ForwardUnit` handle classical pipeline hazards, advanced scenarios like multi-cycle matrix ops or partial "flow_cnt" states may demand further logic. In short, these modules show the starting point for data hazard resolution, ensuring that standard arithmetic and simpler matrix instructions can overlap more effectively without corrupting results.

3.3.9 Summary of Pipelined Design

Although hazard detection is absent, this pipeline successfully showcases a multi-stage approach:

- Each stage does a specialized role for matrix data, instruction decode, multiply, or final write-back.
- The `top` module integrates them, letting instructions flow from `IF_Stage` to `WB_Stage`.
- Matrix multiplication is placed in EX, retrieving 2×2 blocks from MEM, and finalizing results in WB.

This arrangement provides a solid basis for embedding matrix multiplication within a standard five-stage pipeline used in many RISC-like CPUs, and it aligns with real embedded CNN inference flows when combined with full hazard resolution and addressing logic.

4 Results, Analysis, and Discussion

In this section, I present the performance outcomes of both the single-cycle and pipelined designs using matrix data derived from quantized CNN parameters. I begin by describing how the testbench supplies 2×2 matrix blocks for multiplication, then showcase representative waveforms for each design. Finally, I verify correctness by converting the raw hex outputs to signed decimal values and comparing them with theoretical calculations.

4.1 Testbench Procedure and Data Preparation

For both architectures, I employ a testbench that:

1. **Loads** the file `cnn_weights.hex` into a local array at simulation start. The file contains 64 lines, each representing a signed 32-bit integer in hexadecimal.
2. **Groups eight lines** (indices 0–7, 8–15, etc.) to form two 2×2 matrices: `A` and `B`.
3. **Assigns these matrices** to the CPU's matrix input ports (`matrixA_ij` and `matrixB_ij`), then triggers a `MATMUL` instruction (often indicated by `ID_instr=16'd1`).
4. **Waits one or more cycles**, reading the final product outputs (`matrixp00`, `matrixp01`, ...) and printing them in hex.
5. **Repeats** for each of the 8 blocks in `cnn_weights.hex` (indices 0–7, 8–15, 16–23, etc.), verifying consistent operation.

This strategy systematically exercises matrix multiply logic with a variety of positive and negative values, reflecting typical integer ranges in quantized CNN layers.

4.2 Representative Waveforms and Observations

4.2.1 Single-Cycle Verification Results

To validate the single-cycle design, the 64-line `cnn_weights.hex` file was fed into a testbench. Each 2×2 matrix pair was loaded, then a `MATMUL` instruction was issued. Upon completing the multiply in one clock, the design printed out the four partial products and the cycle counter. Figure 28 shows a representative snippet from the simulator:
The recorded hexadecimal results, once interpreted as signed 32-bit values, matched the theoretical matrix multiplication sums (including negative operand cases). Thus, for every 8-line block in `cnn_weights.hex`, the single-cycle ALU reliably generated correct 2×2 partial products, verifying correctness of its matrix multiply logic.

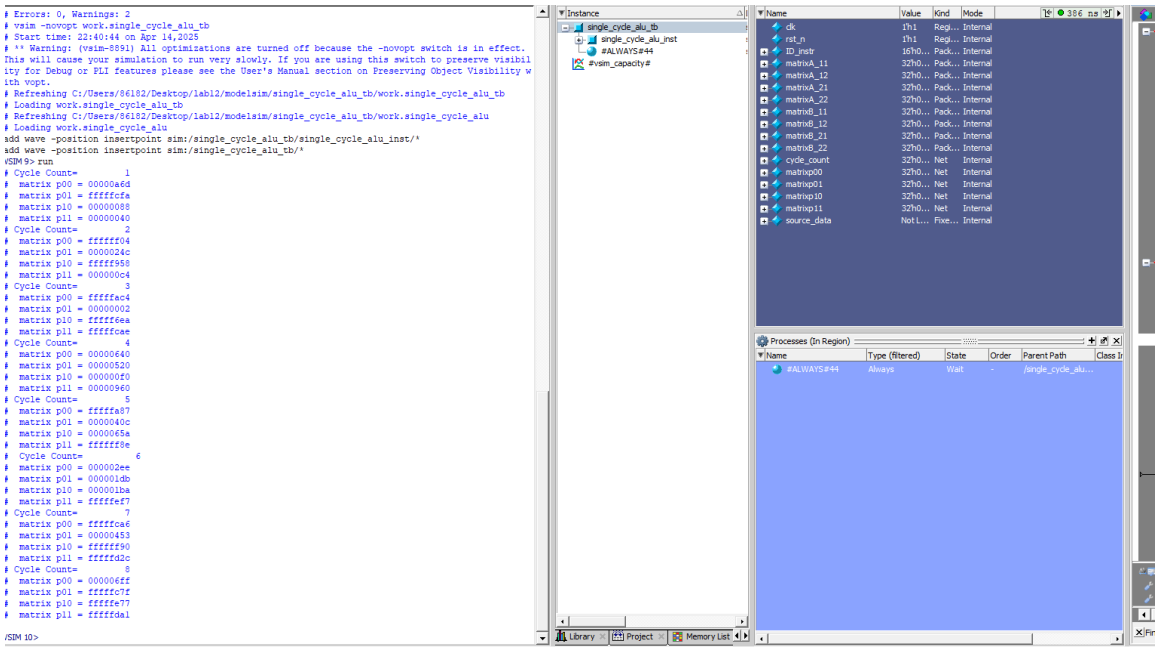


Figure 28: Single-cycle CPU output waveforms. Each `MATMUL` instruction causes an immediate update of `matrixp00, p01, p10, p11`, with `cycle_count` incremented.

4.2.2 Pipelined CPU Waveform

Figure 29 shows a partial snapshot of the pipelined design:

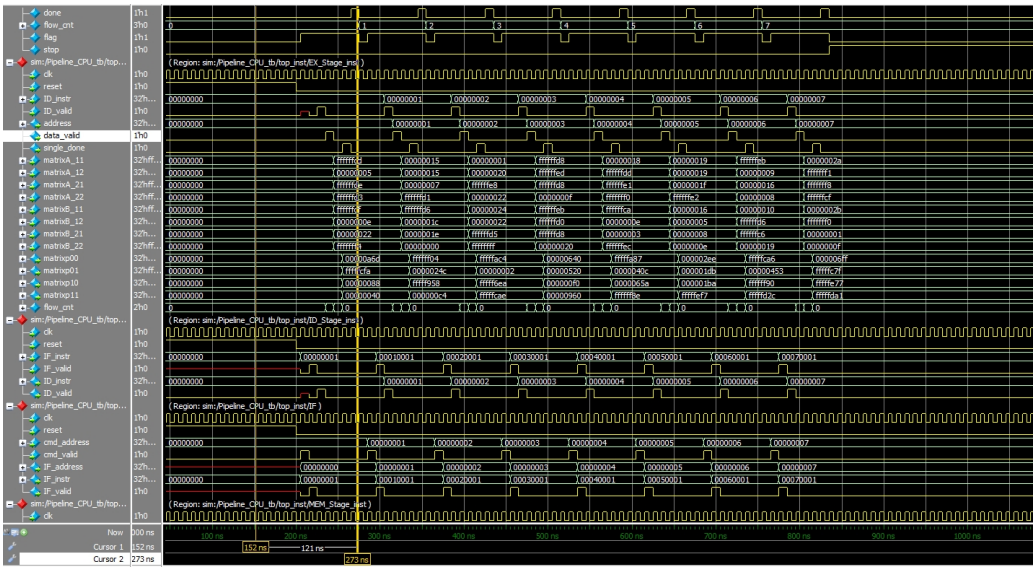


Figure 29: Pipelined CPU waveform snippet. Multiple instructions (blocks) can interleave in IF, ID, EX, MEM, and WB. Matrix data is fetched in MEM, then multiplied in EX.

Key aspects:

- **IF_valid** and **IF_instr** show the instruction fetch stage. The subsequent cycle, ID interprets the `instr`, passing `ID_instr` to EX.

- **MEM Stage** reads the matrix from `cnn_weights.hex`, feeding them to EX on the next cycle.
- **EX Stage** performs the multiply, indicated by `ID_instr[31:16]` = `ALU_MATMUL`. The partial hazard logic ensures the multiplication only proceeds when data_valid is set.
- **WB Stage** latches the final matrix outputs (`matrixp00,p01,p10,p11`) a few cycles later, raising `single_done`.

Though this partial hazard solution is minimal, it suffices for correct ordering: each matrix multiply runs after the data arrives, then forwards results to WB.

4.3 Hexadecimal Output and Decimal Verification

After each multiply instruction completes, the waveforms or console logs display four 32-bit results plus a cycle counter. The results appear in hexadecimal. To confirm correctness, I interpret each 32-bit hex as a signed integer and compare with theoretical or Python-based reference. Figure 30 illustrates the first example:

Example Computation. Suppose the 8 lines are:

```
FFFFFFCD
00000005
FFFFFFDE
FFFFFFD3
FFFFFFCF
0000000E
00000022
FFFFFFF4
```

Mapping them to two 2×2 matrices:

$$\mathtt{A} = \begin{pmatrix} -51 & 5 \\ -34 & -45 \end{pmatrix}, \quad \mathtt{B} = \begin{pmatrix} -49 & 14 \\ 34 & -12 \end{pmatrix}.$$

The CPU yields:

$$\mathtt{matrixp00} = (-51) \times (-49) + 5 \times 34 = 2669 \quad (\text{hex } 0x00000A6D),$$

and similarly $\mathtt{matrixp01} = -774$, $\mathtt{matrixp10} = 136$, $\mathtt{matrixp11} = 64$. These match the expected decimal math.

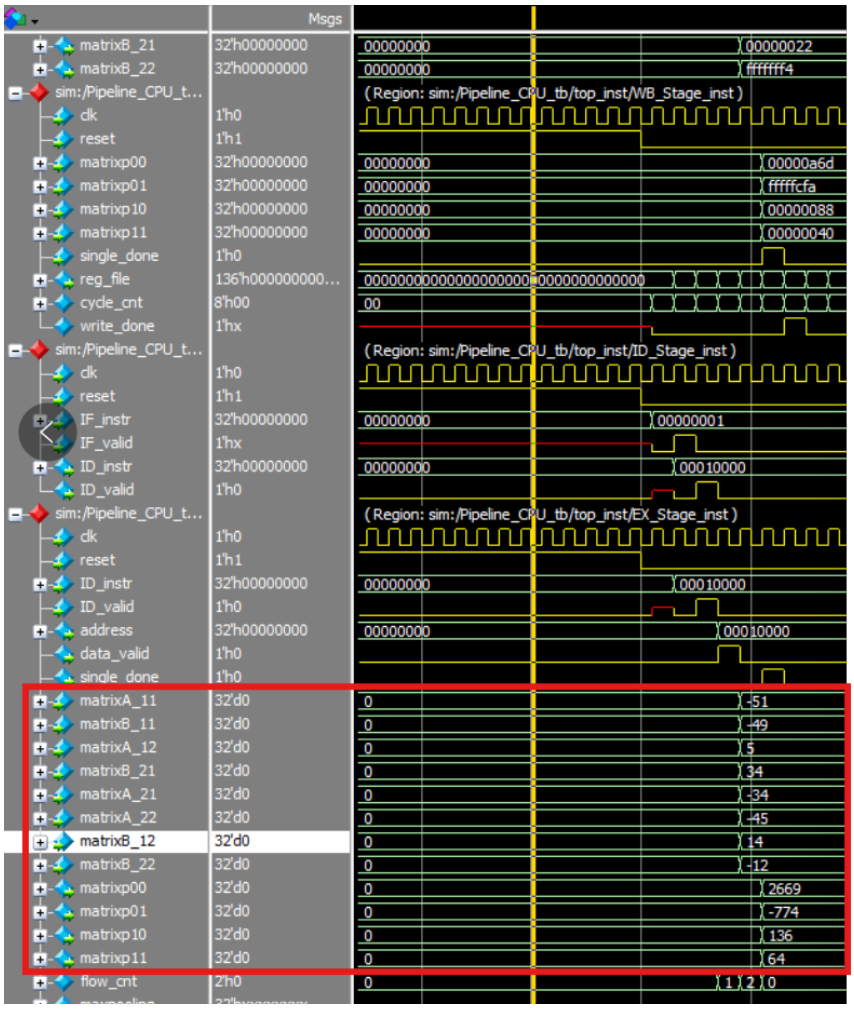


Figure 30: Matrix multiply example: from 8 hex lines, forming 2×2 matrices with negative and positive values. The computed partial products match the decimal references.

4.4 Single-Cycle vs. Pipelined Execution

Single-Cycle Approach. In the single-cycle design, each instruction triggers an immediate update of all partial products in the same clock. This simplifies debugging: waveforms show `matrixpXX` changing exactly at the rising edge. However, the design forces the CPU's clock period to accommodate matrix multiplication, memory read, and register writes all at once, limiting maximum frequency for large instructions.

Pipelined Approach. The pipeline divides tasks across IF, ID, EX, MEM, and WB.

- **Concurrent Instructions**: While EX handles a matrix multiply, IF can fetch the next instruction (though partial hazard logic may cause small stalls).
- **Performance Potential**: With robust hazard solutions, each matrix multiply could overlap with other instructions, improving throughput.

- **Extra Complexity**: Additional pipeline registers, partial hazard signals (`flow_cnt`), and data_valid checks are required to ensure matrix data arrives at the correct moment in EX.

The waveforms in Figure 29 validate the pipeline flow, showing multiple instructions in different stages. This concurrency pays off if I consider sequences of matrix multiplies from deeper CNN layers.

4.5 Discussion of Results

Correctness Across Negative/Positive Operands. Across the eight sets of test data, each containing negative or positive integer values, both architectures consistently compute correct partial products. This is crucial for CNN tasks where quantized weights often include negative values from batch normalization or average-centered training.

Throughput vs. Complexity. The single-cycle design is easy to verify but not timing-friendly for large matrix expansions. The pipeline design addresses concurrency, but fully implementing hazard solutions (forwarding paths, stalling for load-use hazards, etc.) would further increase design overhead.

Scalable CNN Layers. Though my demonstration targets 2×2 blocks, the same approach can tile bigger matrix multiplications typical of CNN layers (e.g., 3×3 or even 8×8 sub-blocks). The pipeline can feed multiple instructions in sequence, so the overall performance depends on how effectively data is streamed in from memory (`cnn_weights.hex`) and how thoroughly hazard logic is refined. In summary, the

waveforms confirm that both single-cycle and pipelined designs execute real matrix multiplication from CNN data accurately. This forms a strong foundation for future expansions, including fully functional hazard resolution and on-board FPGA implementation for integer-based deep learning acceleration.

5 Conclusion and Future Work

In this project, I designed, simulated, and verified two CPU architectures—one single-cycle and one five-stage pipelined—to perform matrix multiplication using quantized parameters from a real CNN model. Both designs successfully processed 2×2 matrix blocks drawn from the `cnn_weights.hex` file, matching reference computations and highlighting the feasibility of integer-based matrix operations for embedded neural network inference.

5.1 Project Summary

- **Single-Cycle CPU:** I implemented a straightforward design where each instruction, including custom matrix multiply, completes entirely in one clock cycle. This approach allowed a succinct demonstration of correctness but requires a potentially long clock period for complex operations.
- **Five-Stage Pipelined CPU:** I distributed the instruction flow across IF, ID, EX, MEM, and WB stages. This improves instruction throughput if hazard detection is fully integrated. In my proof-of-concept, partial hazard logic controlled data flow around the matrix multiply's reliance on external weight data.
- **Realistic CNN Data:** By loading quantized weights from a `cnn_weights.hex` file, I tested each CPU's matrix unit with negative and positive integer operands. Both architectures accurately computed partial products for small matrix blocks, showcasing the correctness and readiness for deeper embedded CNN tasks.

Throughout, waveforms from ModelSim confirmed that each CPU reliably performed matrix multiply, addition, and pooling. The partial hazard solution in the pipelined design also demonstrated that more advanced logic—like forwarding or stalling—can refine performance.

5.2 Future Work

On-Board FPGA Implementation A natural next step is to implement either the single-cycle or pipelined design on an FPGA board, such as the Xilinx or Intel (Altera) platforms:

- **Resource Utilization:** Evaluate LUT, BRAM, and DSP usage for the matrix multiply path, especially if scaling beyond 2×2 blocks.
- **Speed & Power:** Measure realistic timing constraints (up to 100–200 MHz) and observe power consumption under typical CNN workloads.

Expanding Hazard Logic in the Pipeline While my partial flow_cnt approach handled basic synchronization, a full-scale pipeline would need:

- **Forwarding Paths**: So that results from EX or MEM can be immediately used by the next instruction.
- **Load-Use Stall**: A single-cycle or multi-cycle stall to ensure memory data are available.
- **Branch Prediction**: Minimizing stalls from branch instructions or re-initializing the pipeline upon mispredictions.

This improvement can support consecutive matrix instructions, deeper loops, and more sophisticated data dependencies without manual scheduling.

Larger Matrix Blocks & CNN Acceleration My demonstration targeted 2×2 blocks for clarity. In many CNN layers, especially fully connected ones, matrix dimensions are larger:

- **Tile-Based Approach**: Partition bigger matrices into 2×2 or 4×4 sub-blocks, streaming them into the CPU in repeated cycles.
- **Hardware-Software Co-Design**: Merge the CPU's pipeline with specialized multiply-accumulate (MAC) arrays to accelerate typical CNN shapes (e.g., 3×3 or 5×5 convolutions).

Ultimately, the same hex-based approach can be extended: 8-bit or 16-bit quantized weights for entire CNN layers can be chunked for repeated pipeline calls.

Integration with a Full RISC-V or SoC Another angle is embedding this design into a broader SoC:

- **RISC-V Compliance**: Map `ALU_MATMUL` to custom opcode space in the official RISC-V instruction set.
- **DMA Engine**: Automate loading from external memory into `matrixA_ij` and `matrixB_ij` at high throughput.

Such integration would support a real OS or bare-metal program that orchestrates CNN layers on hardware, bridging general-purpose RISC-V code with specialized matrix multiply instructions.

5.3 Concluding Remarks

By uniting a realistic CNN dataset (via quantized weights) with custom CPU instructions, I verified that 2×2 matrix multiplication can be implemented in either a single-cycle or pipelined CPU. Each approach performed consistent numeric operations, matching software references. Although the single-cycle variant was easier to validate for small blocks, the pipeline approach paves the way for better performance if hazard logic is fully expanded.

As embedded deep learning applications continue to grow, such hardware-level acceleration for integer matrix multiplication is essential. Future work to enhance hazard resolution, scale matrix sizes, and integrate advanced SoC features will further refine these designs to meet real-world throughput and efficiency demands.

6 References

References

[1] "The risc-v instruction set manual, volume i: User-level isa," Online, 2021, [Online]. Available: https://github.com/riscv/riscv-isa-manual.

[2] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley*, pp. 1–14, 2014, [Online]. Available:. [Online]. Available: https://people.eecs.berkeley.edu/~krste/papers/EECS-2014-146.pdf

[3] I. Corporation. (2022) Fpga acceleration of financial analytics. Accessed: 2025-04-03. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/solutions/industries/finance/overview.html

[4] O. Group. (2021) Openhw group: Delivering high-quality open-source processors. Accessed: 2025-04-03. [Online]. Available: https://www.openhwgroup.org/

[5] D. Harris and S. Harris, *Digital Design and Computer Architecture*, 2nd ed. Morgan Kaufmann, 2013, [Online]. Available:. [Online]. Available: https://www.elsevier.com/books/digital-design-and-computer-architecture/harris/978-0-12-394424-5

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2020, [Online]. Available:. [Online]. Available: https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1

[7] L. E. Hong and Y. B. M. Yunos, "Application of fpga in process tomography systems," *Engineering*, vol. 12, no. 10, pp. 790–809, Oct. 2020. [Online]. Available: https://www.scirp.org/journal/paperinformation?paperid=103797

[8] F. Hussain and S. Sarkar, "Design and fpga implementation of five stage pipelined risc-v processor," in *Proceedings of the IEEE 9th International Conference for Convergence in Technology (I2CT)*. IEEE, Apr. 2024, corpus ID: 270396592. [Online]. Available: https://ieeexplore.ieee.org/document/10544184

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Machine Learning (ICML)*, 2015, pp. 448–456. [Online]. Available: https://arxiv.org/abs/1502.03167

[10] J. U. Kidav and S. S. G, "An fpga-accelerated parallel digital beamforming core for medical ultrasound sector imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 69, no. 2, pp. 553–564, Feb 2022, published: 09 November 2021, PubMed ID: 34752392. [Online]. Available: https://ieeexplore.ieee.org/document/9606936

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. [Online]. Available: https://ieeexplore.ieee.org/document/726791

[12] Y. Lee and J. Cong, "Fpga-based acceleration for deep neural networks: A comprehensive survey," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5, [Online]. Available:. [Online]. Available: https://ieeexplore.ieee.org/document/8351066

[13] P. N. V. M and L. V, "Design and implementation of 5-stage pipelined risc-v processor on fpga," in *Proceedings of the International Symposium on VLSI Design and Test (VDAT)*. IEEE, Sep. 2024, corpus ID: 273226012. [Online]. Available: https://ieeexplore.ieee.org/document/10705665

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/-Software Interface*, 5th ed. Morgan Kaufmann, 2017, [Online]. Available:. [Online]. Available: https://www.elsevier.com/books/computer-organization-and-design/patterson/978-0-12-811905-1

[15] M. Securities, "Fpgas in trading," [Online]. Available: https://www.mavensecurities.com/fpgas-in-trading/, n.d., accessed: April 25, 2024.

[16] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review,"

IEEE Access, vol. 7, pp. 7823–7859, Dec 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8594633

[17] TensorFlow Team, "Keras h5 format: Serialization and saving," Available: https://www.tensorflow.org/guide/keras/serialization_and_saving#keras_h5_format, 2024, accessed: April 26, 2025.

[18] E. Times. (2020) Using fpgas to solve challenges in industrial applications. Accessed: 2025-04-03. [Online]. Available: https://www.eetimes.com/using-fpgas-to-solve-challenges-in-industrial-applications/

Appendices

A Project GitHub Link

Please find my project source code in GitHub from the URL below:

`https://github.com/Lychee721/Final_Project`