



CPU Simulator

MIPS Pipeline Architecture

Name: Raluca-Mihaela Adam
Group: 30422
Email: adam.ra.raluca@student.utcluj.ro

Teaching Assistant: Dragos Lazea



Contents

1	Introduction	3
1.1	Context	3
1.2	Problem and Motivation	3
1.3	Objectives and Proposed Solution	3
1.4	Project Timeline and Plan	4
2	Bibliographic Research	5
2.1	Importance of CPU Pipelining	5
2.2	MIPS Architecture and Pipelining	5
2.3	Existing Tools	5
2.4	Software Design Considerations	6
3	Analysis	7
3.1	MIPS Pipeline Architecture	7
3.1.1	Pipeline Structure and Stage Definition	7
3.1.2	Instruction Parallelism	8
3.2	Instruction Set Architecture	8
3.2.1	Instruction Formats and Encoding	8
3.2.2	Instruction Categories and Examples	9
3.2.3	Register Transfer Level (RTL) Description	9
3.2.4	Supported Instruction Set	11
3.3	Pipeline Hazard Mechanisms	11
3.3.1	Hazard Classification and Detection	11
3.3.2	Hazard Resolution Algorithms	12
3.4	Processor Components Architecture	13
3.4.1	Control Unit Design	13
3.4.2	ALU Operations and Functionality	13
3.4.3	Memory Hierarchy and Register Organization	14
3.4.4	Simulation Methodology	14
4	Design	16
4.1	System Architecture Overview	16
4.2	Use Case Diagram	16
4.3	Class Diagram and Component Design	17
4.3.1	Core Components	17
4.3.2	Architectural State Components	18
4.3.3	Pipeline Stages	18
4.4	Data Flow and Execution Sequence	18
4.5	User Interface Components	19
4.6	Component Interaction Summary	19

Chapter 1

Introduction

1.1 Context

This project aims to develop an educational CPU simulation software that mimics the 5-stage MIPS (Microprocessor without Interlocked Pipeline Stages) pipeline based on the 32-bit MIPS Instruction Set Architecture (ISA).

The simulator will visualize the flow of instruction through the MIPS pipeline, detect and display hazards, and help users understand the dynamics of parallel execution inside a CPU through a graphical user interface.

1.2 Problem and Motivation

In traditional computer architecture courses, students often struggle to understand the functioning of the CPU pipeline and the types of hazards that arise during instruction execution. Existing tools, such as MARS and SPIM simulators, allow users to run MIPS assembly programs but do not provide an intuitive visualization of how instructions propagate through the pipeline.

This project addresses this educational gap by providing a visual simulation software that demonstrates how instructions overlap inside a pipeline, how hazards occur, and how forwarding or stalling mechanisms resolve them in an interactive and easy-to-follow manner.

1.3 Objectives and Proposed Solution

The proposed application will consist of:

- An execution engine that simulates the MIPS instruction flow and pipeline stages.
- An intuitive graphical user interface for interactive visualization that allows users to either input custom instructions or select predefined programs.
- The possibility to simulate at least 15 MIPS instructions (R, I, and J types).
- Real-time visualization of the 5 pipeline stages (IF, ID, EX, MEM, WB).
- Display of registers, memory contents, ALU operations, and control flags.

1.4 Project Timeline and Plan

The development of the MIPS 32 Simulator will follow a structured timeline, as presented below:

- **Meeting 2:** Documentation phase — prepare the project introduction, bibliographic research, bibliography, and project plan. Establish the main objectives and overall development strategy.
- **Meeting 3:** Analysis and **Design Phase 1** — describe the system architecture, main modules, and component interactions. Provide an initial view of the implementation.
- **Meeting 4:** Updated design and **back-end implementation** — refine the architecture based on feedback and corrections. Begin coding the back-end, defining class structures, storage mechanisms, and pipeline simulation logic.
- **Meeting 5:** Finalize the back-end implementation. Begin developing the **front-end** to visualize the pipeline stages and display simulation data in real time.
- **Meeting 6: Testing and validation** — verify functionality through unit and integration tests.
- **Meeting 7: Final demo and presentation** — deliver the complete simulator, full documentation, and the final presentation showcasing the system's capabilities.

Chapter 2

Bibliographic Research

2.1 Importance of CPU Pipelining

The performance of a processor depends on factors such as clock speed, critical path length, and cycles per instruction (CPI). A single-cycle CPU executes one instruction per clock cycle, making the design simple but inefficient because not all instructions take the same amount of time. Pipelining addresses this limitation by allowing multiple instructions to be processed simultaneously at different stages. The pipeline architecture is designed for throughput, increasing CPU performance, but introduces hazards - structural, data, and control hazards - that must be detected and resolved for correctness.

2.2 MIPS Architecture and Pipelining

The MIPS Architecture is a reduced instruction set computer (RISC) design. Its pipeline implementation consists of 5 execution stages: Instruction Fetch, Instruction Decode, Execute, Memory, and Write Back, which enable the parallel processing of multiple instructions.

In a software simulator, true parallel execution is not possible; however, an event-based software architecture can create the illusion of concurrency while preserving accuracy in time and hazard detection.

2.3 Existing Tools

Several existing MIPS simulators are available:

- **MARS** [4] — A Java-based MIPS simulator commonly used in academia. It supports assembly code execution, but is limited by poor visualisation of the pipeline.
- **SPIM** [6] — One of the earliest MIPS simulators. It is focused mainly on instruction correctness rather than pipeline-level execution.

In [5], the authors present a comprehensive theoretical explanation of the MIPS pipeline and the types of hazards, but without an interactive or visual component. As a general observation, all of the existing solutions above suffer from a lack of visual representation. Thus, this project aims to provide a slightly different perspective by creating an interactive, educational simulator that focuses on how instructions move through the pipeline and how hazards affect performance.

2.4 Software Design Considerations

The simulator will follow object-oriented design principles such as modularity, encapsulation, and separation of concerns. The architecture will clearly separate:

- the execution engine (responsible for instruction handling and pipeline state),
- the data model (registers, memory, instructions), and
- the user interface layer (for visualization and user interaction).

Chapter 3

Analysis

This chapter presents the theoretical foundations for the MIPS CPU simulator, detailing the hardware concepts and architectural mechanisms that will be modeled. The analysis covers the MIPS pipeline structure, instruction execution mechanics, hazard detection, and resolution techniques that form the conceptual basis for the simulator implementation.

3.1 MIPS Pipeline Architecture

3.1.1 Pipeline Structure and Stage Definition

The MIPS pipeline employs a five-stage architecture designed to maximize instruction throughput by enabling concurrent processing of multiple instructions. As illustrated in Figure 3.1, each stage performs specialized operations while maintaining data flow through pipeline registers.

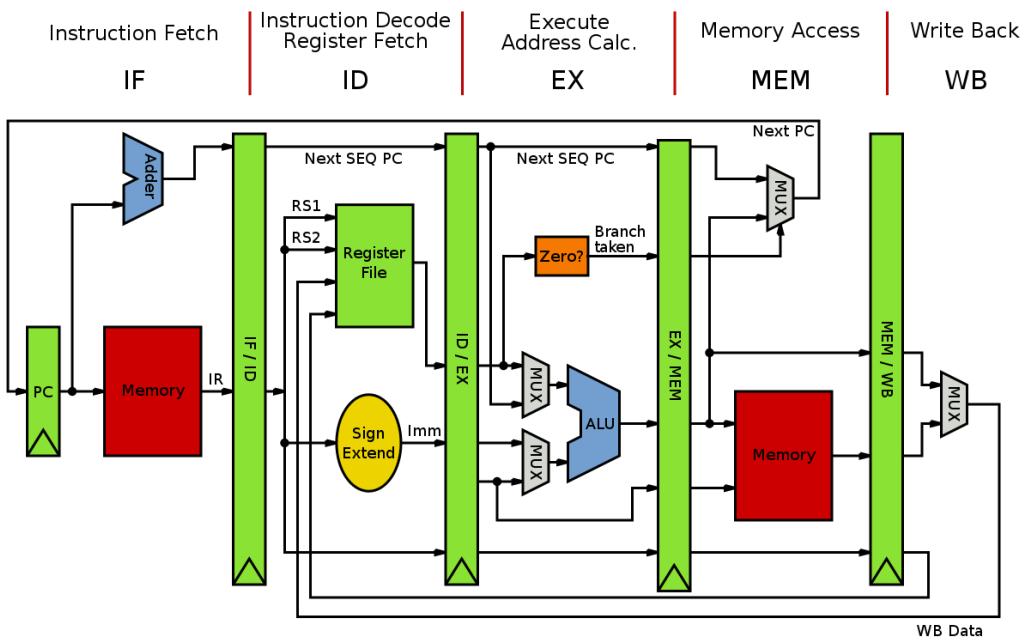


Figure 3.1: General structure of the 5-stage MIPS pipeline.

The five stages operate as follows:

- **Instruction Fetch (IF):** Retrieves instructions from memory using the program counter and increments PC

- **Instruction Decode (ID):** Decodes instruction fields, reads register values, and generates control signals
- **Execute (EX):** Performs arithmetic/logical operations and calculates memory addresses
- **Memory Access (MEM):** Handles data memory reads/writes for load/store instructions
- **Write Back (WB):** Writes results back to the register file

3.1.2 Instruction Parallelism

The pipeline achieves performance gains through instruction-level parallelism, where multiple instructions occupy different stages simultaneously. Figure 3.2 demonstrates this overlapping execution pattern, showing how the pipeline maintains a steady flow of instruction completion once initially filled.

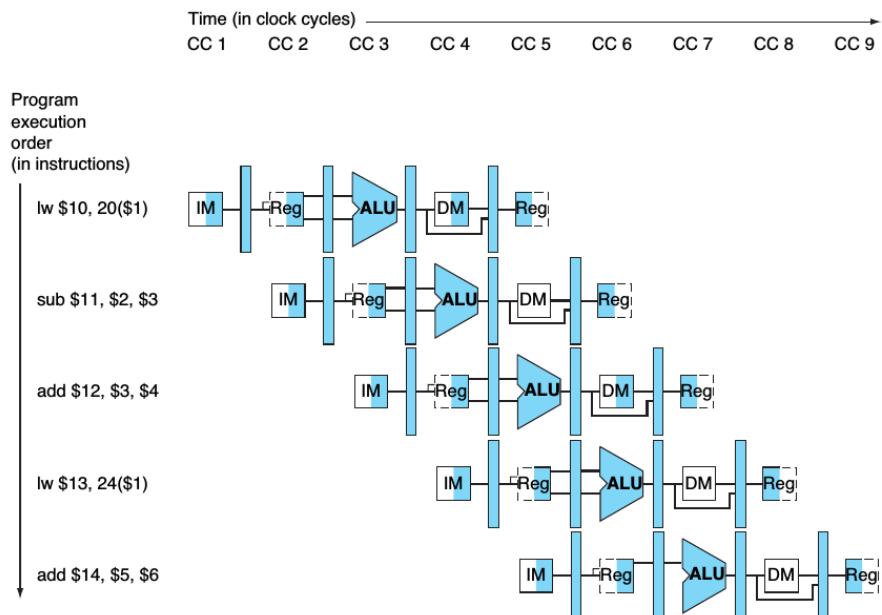


Figure 3.2: Concurrent instruction execution in the MIPS pipeline.

3.2 Instruction Set Architecture

3.2.1 Instruction Formats and Encoding

The simulator implements all three primary instruction formats, each with a fixed 32-bit encoding scheme as shown in Figure 3.3. This consistent formatting enables efficient decoding and execution across all instruction types.

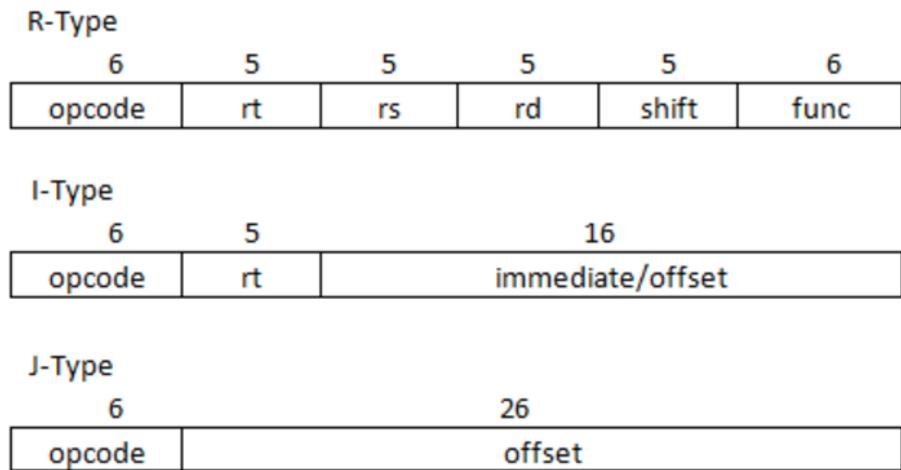


Figure 3.3: MIPS instruction encoding formats.

3.2.2 Instruction Categories and Examples

The supported instruction set includes:

R-type (Register Operations):

- ADD, SUB, AND, OR, SLT
- Example: `add $t0, $t1, $t2` - Register addition

I-type (Immediate/Memory Operations):

- ADDI, LW, SW, BEQ, BNE
- Example: `lw $t0, 4($sp)` - Memory load operation

J-type (Jump Operations):

- J, JAL
- Example: `j main` - Unconditional jump

3.2.3 Register Transfer Level (RTL) Description

The simulator models instruction execution at the register transfer level, describing the data movements and operations between pipeline stages for each instruction type.

R-type Instructions RTL Description

For R-type instructions such as `add $rd, $rs, $rt`:

- IF: $IR \leftarrow \text{Mem}[\text{PC}]$, $\text{PC} \leftarrow \text{PC} + 4$
- ID: $A \leftarrow \text{Reg}[\$rs]$, $B \leftarrow \text{Reg}[\$rt]$, Decode opcode
- EX: $\text{ALUOutput} \leftarrow A \oplus B$ where \oplus is the ALU operation
- MEM: No memory operation
- WB: $\text{Reg}[\$rd] \leftarrow \text{ALUOutput}$

I-type Load/Store RTL Description

For load instructions such as `lw $rt, offset($rs)`:

IF: $IR \leftarrow \text{Mem}[PC]$, $PC \leftarrow PC + 4$
ID: $A \leftarrow \text{Reg}[\$rs]$, Sign-extend immediate
EX: $\text{ALUOutput} \leftarrow A + \text{SignExt}(\text{imm})$
MEM: $LMD \leftarrow \text{Mem}[\text{ALUOutput}]$
WB: $\text{Reg}[\$rt] \leftarrow LMD$

For store instructions such as `sw $rt, offset($rs)`:

IF: $IR \leftarrow \text{Mem}[PC]$, $PC \leftarrow PC + 4$
ID: $A \leftarrow \text{Reg}[\$rs]$, $B \leftarrow \text{Reg}[\$rt]$, Sign-extend immediate
EX: $\text{ALUOutput} \leftarrow A + \text{SignExt}(\text{imm})$
MEM: $\text{Mem}[\text{ALUOutput}] \leftarrow B$
WB: No register writeback

Branch Instructions RTL Description

For branch instructions such as `beq $rs, $rt, label`:

IF: $IR \leftarrow \text{Mem}[PC]$, $PC \leftarrow PC + 4$
ID: $A \leftarrow \text{Reg}[\$rs]$, $B \leftarrow \text{Reg}[\$rt]$, $\text{Target} \leftarrow PC + (\text{imm} \ll 2)$
EX: $\text{Condition} \leftarrow (A = B)$, If Condition then $PC \leftarrow \text{Target}$
MEM: Pipeline flush if branch taken
WB: No register writeback

Jump Instructions RTL Description

For jump instructions such as `j target`:

IF: $IR \leftarrow \text{Mem}[PC]$, $PC \leftarrow PC + 4$
ID: $\text{Target} \leftarrow (PC_{31:28} \parallel (\text{address} \ll 2))$
EX: $PC \leftarrow \text{Target}$
MEM: Pipeline flush
WB: No register writeback

Pipeline Register Transfers

The data flow between stages is maintained through pipeline registers:

IF/ID: IR, PC+4
ID/EX: A, B, SignExt(imm), Control signals
EX/MEM: ALUOutput, B, Control signals
MEM/WB: LMD, ALUOutput, Control signals

This RTL description provides the formal specification for instruction execution that guides the simulator implementation, ensuring accurate modeling of the MIPS pipeline behavior.

3.2.4 Supported Instruction Set

The simulator implements 18 core MIPS instructions covering arithmetic, memory, and control flow operations:

Type	Instruction	Description
R-type Instructions (Register Operations)		
ADD	add rd, rs, rt	Integer addition
SUB	sub rd, rs, rt	Integer subtraction
AND	and rd, rs, rt	Bitwise AND
OR	or rd, rs, rt	Bitwise OR
SLT	slt rd, rs, rt	Set less than (signed)
I-type Instructions (Immediate/Memory Operations)		
ADDI	addi rt, rs, imm	Add immediate
LW	lw rt, offset(rs)	Load word from memory
SW	sw rt, offset(rs)	Store word to memory
BEQ	beq rs, rt, label	Branch if equal
BNE	bne rs, rt, label	Branch if not equal
ORI	ori rt, rs, imm	Bitwise OR immediate
ANDI	andi rt, rs, imm	Bitwise AND immediate
SLTI	slti rt, rs, imm	Set less than immediate
J-type Instructions (Jump Operations)		
J	j target	Unconditional jump
JAL	jal target	Jump and link
JR	jr rs	Jump to register

Table 3.1: Complete instruction set supported by the simulator.

3.3 Pipeline Hazard Mechanisms

3.3.1 Hazard Classification and Detection

Pipeline hazards represent situations when the next instruction cannot be executed in the following clock cycle. Three types of hazards can be distinguished:

- **Structural Hazards:** Arise from resource conflicts; when two instructions attempt to use the same hardware component at the same time. An example is shown in Figure 3.4, where the Load and Instruction 3 both need to access the memory in the same cycle.

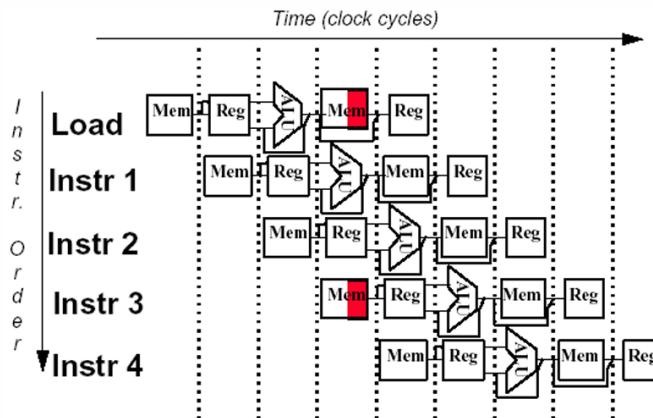


Figure 3.4: Simultaneous memory access of two instructions in different stages.

- **Data Hazards:** Occur when instructions depend on results from previous instructions still in the pipeline. Figure 3.5 presents a Read-After-Write (RAW) hazard where the `and` instruction needs \$2 in cycle 3 (during ID stage) or cycle 4 (during EX stage), but the value won't be available until cycle 5.

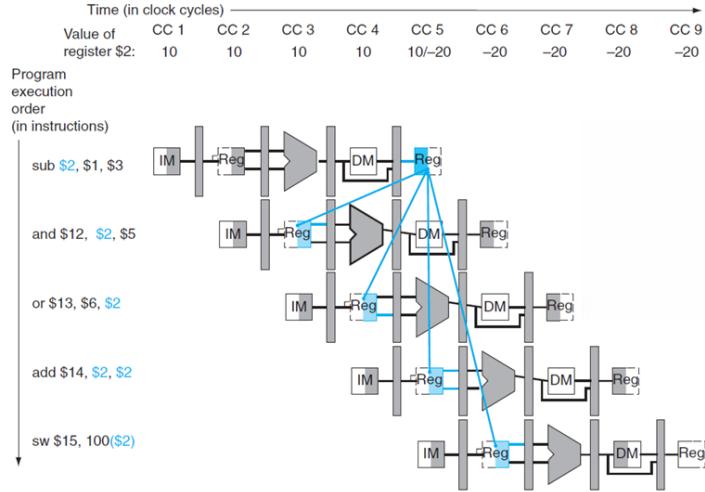


Figure 3.5: RAW Data Hazard.

- **Control Hazards:** Result from branch and jump instructions altering the program flow; attempt to make a decision about program control flow before the condition has been evaluated and the new program counter target address computed. An example is detailed below:

```

Cycle 1: beq $t0, $t1, label      # Branch instruction
Cycle 2: add $t2, $t3, $t4        # Instruction after branch
Cycle 3: lw $t5, 0($t6)          # Second instruction after branch
Cycle 4: label: add $t7, $t8, $t9 # Branch target
    
```

3.3.2 Hazard Resolution Algorithms

The simulator implements all three hazards presented above, as well as several resolution strategies to maintain pipeline efficiency:

Pipeline Stall Algorithm:

1. Identify unresolvable data or structural hazards
2. Insert pipeline bubbles by disabling register writes and PC updates
3. Maintain pipeline state until hazard resolves

Forwarding Algorithm:

1. Detect data dependencies between instructions in EX/MEM/WB and ID stages
2. Route results directly from producing stages to consuming stages
3. Bypass register file access when newer data is available

Branch Prediction Strategy:

1. Assume branch not taken for simple prediction
2. Flush pipeline if prediction proves incorrect
3. Redirect instruction fetch to correct target address
4. **Penalty:** 2 clock cycles lost when branch is taken

3.4 Processor Components Architecture

3.4.1 Control Unit Design

The control unit generates control signals across all pipeline stages based on decoded information from the instruction opcode and/or function field. Its operation follows this sequence:

1. Analyze opcode and function fields during ID stage
2. Generate stage-specific control signals (RegWrite, MemRead, MemWrite, ALUOp, etc.)
3. Propagate control signals through pipeline registers
4. Deactivate signals for instructions that don't use specific stages

The data-path with control signals is depicted in Figure 3.6. Each of the described steps can be observed in blue.

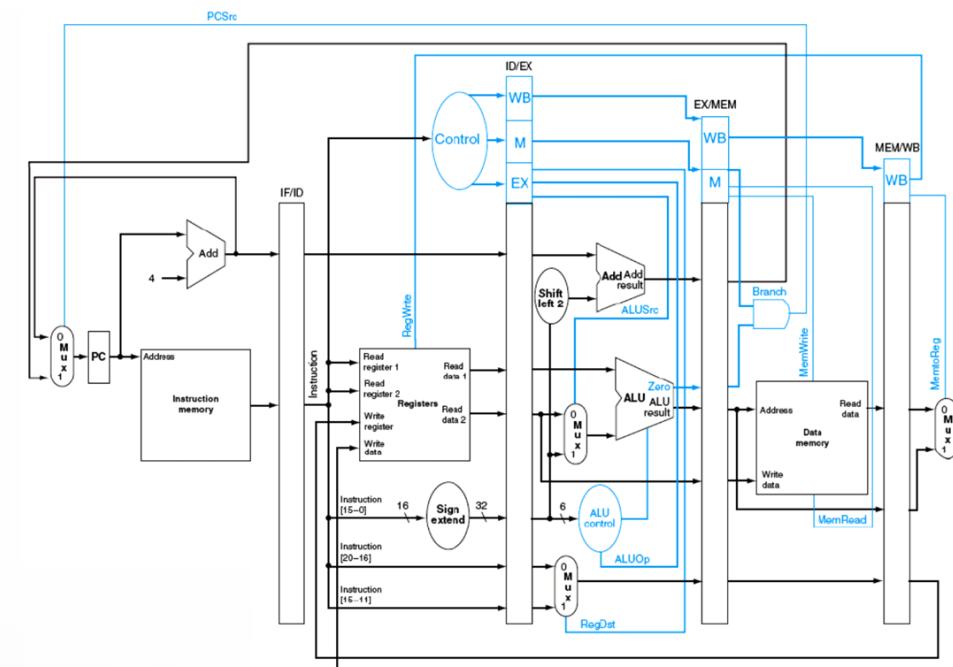


Figure 3.6: MIPS Pipeline Data-Ptah with control signals.

3.4.2 ALU Operations and Functionality

The Arithmetic Logic Unit performs computations based on control signals and operand inputs. Key operations include:

- Arithmetic: Addition, subtraction with overflow detection

- Logical: AND, OR, XOR, NOR operations
- Comparison: Set-less-than for signed and unsigned values
- Shift: Logical and arithmetic shift operations

3.4.3 Memory Hierarchy and Register Organization

The memory subsystem uses separate instruction and data memories following the Harvard architecture principle to avoid structural hazards. This separation allows simultaneous instruction fetch and data memory access in different pipeline stages.

Register File Organization

The MIPS architecture features 32 general-purpose 32-bit registers.

The register file organization supports pipeline operation through:

- Dual read ports for simultaneous reading of two source operands during ID stage
- Single write port for register updates during WB stage
- The `$zero` register is hardwired to value 0 for efficient constant generation
- Register bypass logic to handle RAW data hazards between pipeline stages

Memory Organization

The memory system features:

- Byte-addressable 32-bit address space
- Word-aligned memory accesses (addresses divisible by 4)
- Selectable endianness
- Separate instruction and data memories to prevent structural hazards
- Memory-mapped I/O regions for input/output operations

3.4.4 Simulation Methodology

The simulator models the MIPS pipeline using a discrete, clock-driven approach. Each clock cycle represents a complete synchronization point where all pipeline stages update their state and data flow concurrently.

Simulation Approach

- **Clock-driven Execution:** The simulation advances one clock cycle at a time.
- **Pipeline Synchronization:** All stages (IF, ID, EX, MEM, WB) execute in parallel, maintaining consistent instruction flow.
- **Event Handling:** During each cycle, instructions progress through stages, hazards are managed, architectural state is updated, and the visualization reflects the new system state.

Clock Cycle Execution Sequence

Each clock cycle involves the following steps:

1. Propagate instructions through pipeline stages (IF → ID → EX → MEM → WB)
2. Detect and resolve hazards using forwarding or stalling
3. Handle control flow changes and flush mispredicted instructions
4. Update registers and memory with computed results
5. Refresh the visualization and increment the simulation clock

Chapter 4

Design

This chapter presents the design and architectural structure of the MIPS pipeline simulator. Building on the theoretical concepts described in Chapter 3, this section outlines how those ideas are implemented in software.

4.1 System Architecture Overview

The simulator follows a modular, layered architecture composed of three main layers:

- **Simulation Engine Layer** – Implements the core logic of instruction execution and hazard management.
- **Data Model Layer** – Represents the architectural state of the processor, including registers, memory, and pipeline registers.
- **Presentation Layer** – Provides visualization and interaction for the user through graphical components.

The layered approach promotes separation of concerns, making the design easy to maintain and extend in the future.

4.2 Use Case Diagram

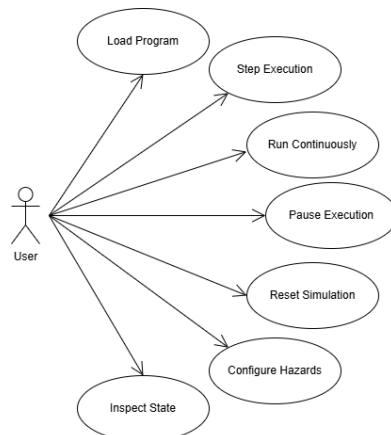


Figure 4.1: Figure 1. Use case diagram showing user interactions with the simulator.

As illustrated in Figure 4.1, users can load a MIPS program, execute it step-by-step or continuously, inspect the pipeline state, and enable or disable hazard resolution mechanisms. The main interactions include program loading, execution control, state inspection, and simulation reset.

4.3 Class Diagram and Component Design

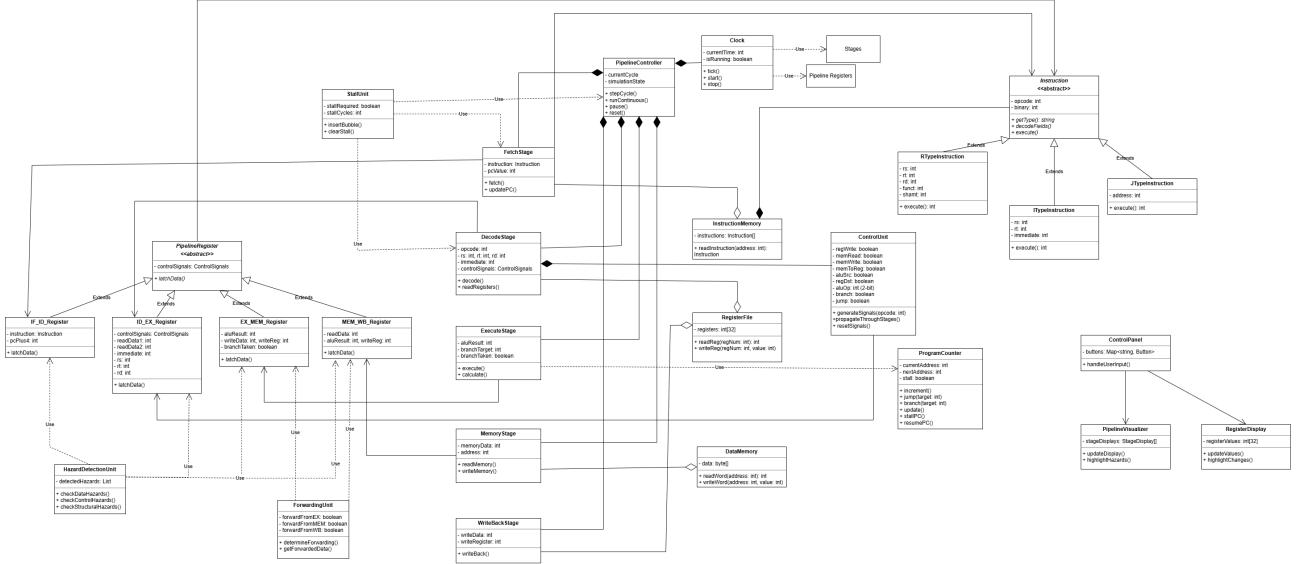


Figure 4.2: Figure 2. Class diagram showing main system components and their relationships.

Figure 4.2 illustrates the system's class structure. The simulator is centered around the **PipelineController**, which manages clock cycles, pipeline stage coordination, and hazard handling. Each pipeline stage—**FetchStage**, **DecodeStage**, **ExecuteStage**, **MemoryStage**, and **WriteBackStage**—is represented by a dedicated class responsible for its specific functionality.

4.3.1 Core Components

PipelineController

Coordinates overall execution, synchronizing pipeline stages each cycle. It interfaces with the hazard detection and forwarding units to ensure correct instruction flow.

HazardDetectionUnit, ForwardingUnit, and StallUnit

These units collectively handle instruction dependencies and pipeline control. The **HazardDetectionUnit** identifies data, control, and structural hazards during each clock cycle. When a hazard cannot be resolved through data forwarding, the **StallUnit** introduces pipeline stalls to preserve correct execution order. The **ForwardingUnit** handles read-after-write (RAW) hazards by routing results directly between stages whenever possible.

Pipeline Registers

Classes such as **IF_ID_Register**, **ID_EX_Register**, **EX_MEM_Register**, and **MEM_WB_Register** hold intermediate data between stages, encapsulating control signals and operand values for the next clock cycle.

4.3.2 Architectural State Components

RegisterFile

Stores the 32 general-purpose registers. Provides simultaneous read ports and one write port for concurrent access during pipeline execution.

Memory Subsystem

Implemented as `InstructionMemory` and `DataMemory` classes, maintains separation between instruction fetch and data access paths.

ProgramCounter

Tracks the address of the instruction currently being fetched. Updated each cycle or redirected in case of branch instructions.

ControlUnit

Generates the control signals required by each instruction type, influencing multiplexers, ALU operations, and memory control lines.

4.3.3 Pipeline Stages

Each stage encapsulates its corresponding MIPS pipeline function:

- **FetchStage:** Retrieves the next instruction from `InstructionMemory`.
- **DecodeStage:** Decodes instruction fields and reads operand values from the `RegisterFile`.
- **ExecuteStage:** Performs arithmetic or logical operations through the ALU.
- **MemoryStage:** Accesses or updates data memory for load and store instructions.
- **WriteBackStage:** Writes computed results back into the register file.

4.4 Data Flow and Execution Sequence

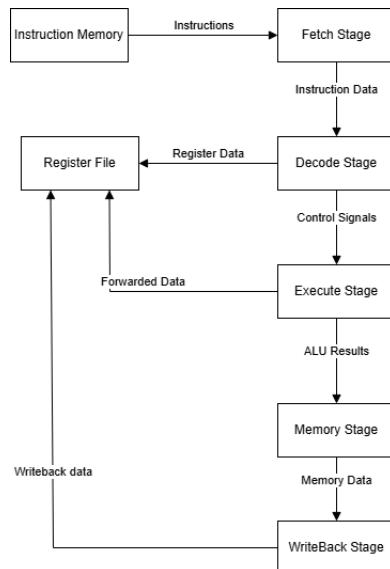


Figure 4.3: Figure 3. Data flow diagram illustrating information exchange between components.

As depicted in Figure 4.3, the simulation proceeds in a sequence of discrete clock cycles managed by the `PipelineController`. Each cycle involves:

1. Fetching the next instruction.
2. Decoding and preparing operands.
3. Executing ALU or branch operations.
4. Accessing memory (if required).
5. Writing results to registers.

Hazard detection and forwarding operate concurrently during each cycle to maintain correctness.

4.5 User Interface Components

The visualization layer presents the internal state of the simulator in real time. It includes:

- **PipelineVisualizer:** Displays the content and status of each pipeline stage.
- **RegisterDisplay:** Shows current register values and highlights recent changes.
- **ControlPanel:** Provides execution controls for stepping, running, pausing, and resetting the simulation.

These components are updated after every clock cycle, ensuring synchronized feedback between the simulation engine and the user interface.

4.6 Component Interaction Summary

During each clock cycle, the following sequence occurs:

- The `PipelineController` triggers all stage modules.
- Pipeline registers propagate intermediate results.
- The `HazardDetectionUnit` and `ForwardingUnit` adjust data paths as needed.
- Architectural state components (`RegisterFile`, `Memory`, `ProgramCounter`) are updated.
- Visualization components refresh to display the new state.

Bibliography

- [1] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. 5th. Morgan Kaufmann, 2014.
- [2] *Computer Architecture Lecture 8: Mihai Negru, UTCN*. <https://mihai.utcluj.ro/wp-content/uploads/ca/lectures/Lecture08.pdf>. [accessed October 2025].
- [3] *Computer Organization and Design*. <https://harttle.land/2014/02/17/computer-design-processor.html>. [accessed October 2025].
- [4] *MARS MIPS Simulator*. <https://computerscience.missouristate.edu/mars-mips-simulator.htm>. [accessed October 2025].
- [5] *MIPS Pipeline - Cornell: Computer Science*. <https://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/09-pipelined-cpu-i-g.pdf>. [accessed October 2025].
- [6] *SPIM MIPS Simulator*. <https://pages.cs.wisc.edu/~larus/spim.html>. [accessed October 2025].

Distributed systems (DS)

