Unpipelined Processor

ECE 5367 Term Report

Haadi Farooqui, Ko Hernandez, Fatima Tahir, Antionio Rodriguez-Enciso, Jorge Penaloza, John Dinh, Garret Cain

University of Houston

Houston, Texas

*Abstract*— This project presents a simulation of a simple MIPS integer processor capable of executing MIPS binaries. The simulator, developed using C++, simulates a multi-cycle un-pipelined MIPS processor, considering instructions for data transfers, arithmetic logic operations, and control-flow instructions. The input file specifies initial register and memory contents, while the output file records the timing of instruction execution cycles and the final contents of registers and memory. The project aims to provide a hands-on understanding of MIPS architecture and processor simulation techniques.

# ***Keywords— MIPS, RISC, Computer Architecture, Simulated MIPS Un-Pipelined Processor, MIPS Integer Processor, University of Houston, ECE 5367***

1. Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) is a widely used instruction set architecture in embedded systems and educational contexts. Understanding MIPS architecture and its operation is crucial for computer engineering students. This project endeavors to provide a practical approach to comprehend MIPS processor behavior through simulation. By implementing a MIPS simulator, we can observe the execution of MIPS binaries and analyze processor behavior under different instructions.

1. methodology
2. *Identifying the Instruction Type*

After having defined the scope of the project we outlined the following objectives:

* The simulator reads the input file provided by the user. It parses the initial register and memory contents, storing them appropriately for simulation.
* The simulator executes each instruction from the code section of the input file. It follows the multi-cycle un-pipelined approach, where each instruction goes through different stages (IF, ID, EX, MEM, WB) sequentially.
  + Instruction Fetch (IF): Fetches the instruction from memory.
  + Instruction Decode (ID): Decodes the instruction and reads operand values from registers.
  + Execution (EX): Performs the arithmetic or logic operation specified by the instruction.
  + Memory Access (MEM): Accesses memory (e.g., for load or store instructions).
  + Write Back (WB): Writes the result back to the destination register.
* The simulator records the timing of instruction execution cycles and the final contents of registers and memory.
* After completing one round of simulation, the simulator prompts the user for further input, allowing multiple simulations if desired.

1. *Performing the Operations*

Once an instruction type is identified, the MIPS simulator performs specific operations according to the instruction's requirements. Register numbers, present in the same bit positions for both target and source registers, are converted from binary to decimal to facilitate calculations.

In R-Type instructions, bits 11-14 of the MIPS binary specify the destination register, where the result of the operation is stored. For an ADD instruction, the simulator calculates the sum of values in the target and source registers, updating the value in the destination register accordingly. Similarly, for a SUB instruction, it computes the difference between the target and source registers, updating the destination register accordingly. In the case of the SLT instruction, the simulator compares values in the source and target registers. If the source register value is less than the target register value, the destination register is set to "1"; otherwise, it is set to "0".

In the ADDI instruction, the 16 least significant bits represent an immediate value added to the source register. The simulator updates the value in the target register to the calculated sum. For LW and SW instructions, the 16 least significant bits represent an address offset added to the base address, which is the value in the source register. If the calculated memory address equals zero, the simulator inserts a new value in memory at that address, defined as 0. In LW instructions, the value in the target register is updated to the content of memory at the calculated address, simulating a load operation. In SW instructions, the memory content is updated to the value in the target register, simulating a store operation.

Control flow instructions like BEQ and BNE contain a 16-bit offset specifying the branching position if a condition is met. For BEQ, if the target register value equals the source register value, the simulator adds the offset to the current instruction position index, skipping or returning to the calculated position to execute the corresponding instruction. Similarly, for BNE, if the values are not equal, the offset is added to the current instruction position index for branching.

### *Logic*

By reading binary instructions from an input file and executing them in a simulated environment, the code simulates the actions of a MIPS processor. To classify MIPS instructions into R-type or I-type based on their opcodes, it begins by reading the file to extract register values, memory contents, and MIPS instructions. Each instruction is handled differently during execution: I-type instructions handle branching, and R-type instructions conduct arithmetic operations on registers. A pipeline representation that describes steps like instruction fetch, decode, execute, memory access, and write back is also included in the simulation. Each stage's timing data is output to offer insights into how the pipeline functions. The technique is a useful tool for comprehending MIPS instruction execution and pipeline behavior since it guarantees resilience by managing issues such as unsupported opcodes or missing registers.

### *Write To Output*

The program will append the instruction name to a vector each time it simulates an instruction operation. This vector functions as a list that records the operations that have taken place. The instruction sequence timing will be output using this list. Every time an instruction stage takes place, a counter that counts the number of clock cycles that occur during the simulation is started at one and updated. The program will output the phases of an instruction as iterates through the list of instructions that have been executed. The number of the clock cycle, the instruction number, and the stage of instruction that is happening during the clock cycle will all be output by the program.

Subsequently, the program will raise the counter and employ the identical procedure to output the remaining phases of an instruction. Every instruction step lasts for one clock cycle. Every instruction will produce the following output stages: execute (EX), decode (ID), and fetch (IF). The Write Back (WB) stage will also be output by R-type instructions. A LW instruction will output the three stages that are shared by all instructions, plus the Memory Access (MEM) and Write Back (WB) stages.

Following the first three stages, a SW instruction will additionally output the Memory Access (MEM) stage. There won't be any more stages output by BEQ and BNE.

1. CODE IMPLEMENTATION
   * 1. *‘parseRTypeInstruction’:*

Input: Binary value representing an R-type MIPS instruction.

Output: Populates the provided MIPS struct with opcode, register numbers, shift amount, and function code.

Logic: Extracts relevant fields from the binary instruction string using substrings and converts them to integers using stoi. Then assigns these values to the corresponding fields in the MIPS struct.

A screen shot of a computer

Description automatically generated

Figure 1: ‘parseRTypeInstruction’ Function

* + 1. ‘*parseITypeInstruction’:*

Input: Binary value representing an I-type MIPS instruction.

Output: Populates the provided MIPS struct with opcode, register numbers, and immediate value.

Logic: Similar to parseRTypeInstruction, but extracts different fields representing opcode, source and target registers, and immediate value.

A screen shot of a computer code

Description automatically generated

Figure 2: ‘parseITypeInstruction’ Function

* + 1. *’ parseInstruction’:*

Input: Binary value representing a MIPS instruction.

Output: Populates the provided MIPS struct based on the instruction type (R-type or I-type).

Logic: Determines the instruction type based on the opcode (first 6 bits of the binary instruction). Then calls either parseRTypeInstruction or parseITypeInstruction accordingly.

A screen shot of a computer code

Description automatically generated

Figure 3: ‘parseInstruction’ Function

* + 1. *’ ReadFile’:*

Input: Filename of the input file and a reference to the RM struct to populate.

Output: Populates the RM struct with register values, memory contents, and MIPS instructions.

Logic: Reads the input file line by line. Parses and stores register values until encountering the "MEMORY" section. Then parses and stores memory contents until encountering the "CODE" section. Finally, parses and stores MIPS instructions.

A screenshot of a computer program

Description automatically generated

Figure 4: ‘ReadFile’ Function

* + 1. *’executeRTypeInstruction’:*

Input: R-type MIPS instruction and the RM struct representing the processor state.

Output: Updates the processor state by executing the R-type instruction.

Logic: Extracts register names based on the instruction fields. Performs the specified operation (addition, subtraction, or comparison) using the values stored in the registers. Updates the destination register with the result.

A screenshot of a computer program

Description automatically generated

Figure 5: ‘executeRTypeInstruction’ Function

* + 1. *’ executeITypeInstruction2’:*

Input: I-type MIPS instruction and the RM struct representing the processor state.

Output: Updates the processor state by executing the I-type instruction.

Logic: Extracts register names based on the instruction fields. Performs the specified operation (load, store, or add immediate) using the values stored in the registers and memory. Updates the destination register or memory location with the result.

A screenshot of a computer program

Description automatically generated

Figure 6: ‘executeITypeInstruction2’ Function

* + 1. *’executeInstruction’:*

Input: MIPS instruction, RM struct representing the processor state, output file stream, program counter, cycle count, and instruction counter.

Output: Executes the MIPS instruction and updates the processor state and output file with timing information.

Logic: Determines the instruction type (R-type or I-type). Calls either executeRTypeInstruction or executeITypeInstruction2 accordingly. Updates the program counter and output file with timing information for each instruction stage.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

Figure 7: ‘executeInstruction’ Function

* + 1. *’simulateProcessor’:*

Input: Vector of MIPS instructions, RM struct representing the processor state, and output file stream.

Output: Simulates the execution of MIPS instructions and updates the processor state and output file with timing information.

Logic: Iterates through each MIPS instruction, calling executeInstruction for each. Updates the program counter, cycle count, and instruction counter. Writes timing information and final register/memory contents to the output file.

A screenshot of a computer program code

Description automatically generated

Figure 8: ‘simulateProcessor’ Function

* + 1. *’main’:*

Logic: Prompts the user for input and output file names. Opens the output file and checks for errors. Reads the input file and populates the RM struct. Simulates the processor and writes the results to the output file. Asks the user if they want to run the code again and repeats if requested. Ends the program when the user chooses not to run it again.

A screenshot of a computer program

Description automatically generated

Figure 9: ‘main’ Code

CONCLUSION

# For every test scenario that is presented, the program performs as anticipated. The user does not need to engage with the computer to direct the simulation process; all timing information is precise, and the application can accept various input files. Compared to utilizing several vectors to track the same information, using maps to update memory and register information in this simulation was a more efficient technique to maintain track of the necessary information. The user can see the path a MIPS instruction will follow through the architecture during execution by looking at the program's output.