MIPS Pipeline Processor

ECE 5367

Spring 2025

Juan Aguilar (ID:2331246)

Introduction

Modern technology is built upon fundamental computer architecture, which has evolved throughout the years. The central processing unit is often regarded to be the “heart” or “brain” of any computer to ever exist. The project presented aims to simulate a MIPS processor and its 5-stage pipeline, with data and control hazard prevention. Each instruction in a program undergoes the Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write-Back (WB). Though the instruction may not make use of each stage, it must undergo the same process.

Data hazards are typically associated with the R-type and I-type instructions such as ADD, SUB, LW, and SW. These hazards exist when a current instruction depends on the result of a previous instruction that has yet to reach the write-back stage. This poses an issue since the current instruction is working with outdated information, leading to incorrect results. In this simulation, the hazard is tackled by delaying the current instruction with a “stall”, which allows the dependent instructions to wait until the previous instructions are complete.

Control hazards relate to the I-type branching instructions. These instructions create a hazard when it is not clear whether the branching will be executed. Because the pipeline will introduce a new instruction each clock cycle, it can be difficult to determine whether those instructions are to execute once the branching is resolved. To combat this, a prediction is made, where it is assumed that the instructions following the branch are correct, and the branching is not. If the branching decision is positive, then the instructions under the prediction are “flushed” from the pipeline, and the correct path is resumed.

The simulation is implemented in C++, where an input file contains the register values, memory contents, and the 32-bit binary-coded instructions. The simulation makes use of the vector data structure to represent the data in the input file, where the simulator checks each instruction with previous ones and handles hazards as needed. Once the simulation ends, the final checks for flushes and branching are made before the output. The objective is to properly parse the input file to simulate the pipeline with data and control hazards, along with a user-friendly console interface.

Design

The simulation initializes by asking the user to enter their input file name. This allows the user to choose the instructions or register values they wish to test. A small check is made to ensure the file opens; if it does not open properly, an output message is generated, and the program ends. It is expected that the input file is correctly formatted, so the logic for parsing immediately follows the file opening phase, as seen in figure 1.

A screen shot of a computer program

AI-generated content may be incorrect.

*Figure 1: File Opening Phase*

The parsing phase begins by extracting each line within the input file, where three keywords are looked for: “REGISTERS”, “MEMORY”, and “CODE”. These keywords determine how the information contained within the section is extracted. As shown in figure 2, the register section pulls the string containing the register location, and the register value. The information is then placed into the register vector, where it is already initialized to represent the thirty-two registers in a MIPS processor. Once the line in the file reaches another keyword, it will break from the current logic and enter the correct one instead. Memory information contains an address and value, which is converted from a string to an integer, and is then stored into the memory vector.

A computer screen shot of code

AI-generated content may be incorrect.

*Figure 2: Register and Memory Parsing*

Once the parsing function reaches the code keyword, it will start to translate the binary strings into their respective instructions. A structure is created to contain the information decoded, which consists of the instruction type, name, opcode, register values, shift amount, function, and address/immediate values. Based on the opcode, that is, the first six bits, the function enters the extraction for either the R-type or I-type decoding. Once the decoding is finished, the information is stored into the instruction vector. Figures 3 and 4 showcase the instruction structure and the code string parsing logic.

A computer screen shot of a program code

AI-generated content may be incorrect.

*Figure 3: Instruction Structure*

A computer screen shot of a program code

AI-generated content may be incorrect.

*Figure 4: Code String Parsing*

One of the goals the project aims to tackle is to detect hazards within the pipeline. This is done by checking the previous instruction relating to the current instruction. Since there are two format types used in the simulation, it is important to know how the instruction can affect future instructions. Since control hazards only include branches for this test case, the focus will be on detecting data hazards. Near the end of each round, the execution for each instruction is finalized, and placed into its correct address if it applies.

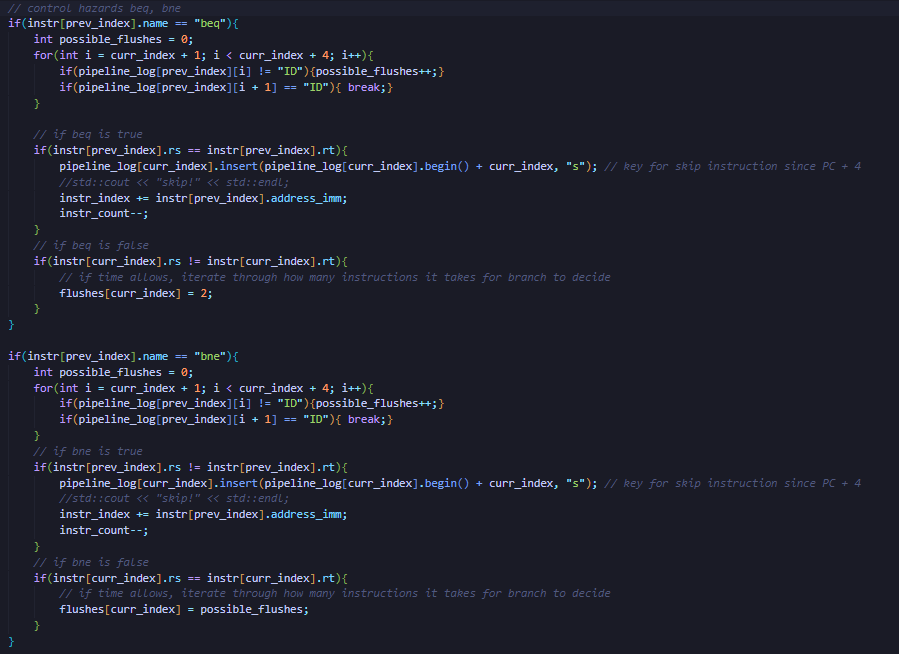
For data hazard handling, each instruction is compared to the previous instruction, and based on the function, the program will know whether to stall the current instruction or continue with the next stage. As seen in figure 5, ADD, SUB, and SLT are checked for dependencies. If that is the case, it will stall the current instruction, until the previous instruction reaches the write-back stage. Similarly, ADDI, and LW will have its destination register checked for dependencies, which will result in the stalling of the current instruction. Note since SW does not modify registers, it does not need to be explicitly checked.

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 5: Data Hazard Handling*

For control hazards there are only two instructions to consider, BEQ and BNE. In essence, the simulator will check to see if the previous instruction is a branch, thus a prediction is needed. The assumption is that the branch is not executed, so we assume the following instructions are correct. Once the branching is resolved in the execution stage and the assumption is correct, the program continues. However, if the branching is executed, then the instructions that followed the initial assumption are flushed from the pipeline. This means that there are at least two instructions that get into the pipeline under the assumption, which can be flushed. Due to the structure of the branching instruction, the PC register will change its address, leading to a skip in some of the instructions, depending on the branching immediate value. Thus, it’s important to update the index that the pipeline is currently in, to preserve correct information. Within the simulation, an instruction that is skipped, is marked “s”, so that the output will not include it, and since the index of the pipeline skips it as well, the instruction never gets executed. Figure 6 highlights the logic implemented for control hazards.



*Figure 6: Control Hazard Handling*

Figure 7 demonstrates how each instruction is executed at the end of each round the simulation reaches. At the end of each round, the user is asked if they would like to simulate another round. Though the simulation is meant to be finished, it can be useful for seeing how the pipeline progresses per round. Furthermore, if there are no instructions left to simulate, the program will automatically output a message, notifying the user that the pipeline can no longer be simulated. This allows the program to begin the final output process.

A screen shot of a computer screen

AI-generated content may be incorrect.

*Figure 7: Instruction Execution at Write-Back Stage*

Once the simulation is completed, or once the user decides to stop, the program will immediately start the output process. Since the pipeline is initiated in a 2D vector, the columns, or clock cycles are detected first. Once that is done, it must undergo skipping/flushing checks. If the program detects an instruction marked “s”, it’s output must be skipped. Thus, the final output only cares for the instructions that aren’t marked or were never executed in the pipeline itself. Once the pipeline output is done, the registers and memory that are not zero valued in the vectors will be part of the output. Thus, the program is complete, and a new file “output.txt” appears in the project folder. Refer to figure 8 for code implementation.

A screen shot of a computer code

AI-generated content may be incorrect.

*Figure 8: Final Output*

Results

Figures 9 and 10 display the input file and the output file assuming the simulation goes through all rounds. The input file covers 6 instructions, consisting of instructions LW, ADD, SW, BNE, and ADDI. By tracing the instructions by hand and comparing them with the output, it is shown to be a successful project. Not only are the registers and memory correct, but the pipeline itself is in the correct order and the clock cycles properly represent what is happening. Since the branching was executed as instruction five, ADDI, is skipped. Thus, the final output does not contain instruction five at all. Stalls are properly placed in instructions 2 and 3 as they have dependencies on register being modified in earlier instructions. Figure 11 is harder to interpret since 2 rounds of simulation may not cover hazard checks properly. It can be said that instruction 3 may suffer from a data hazard since it does not stall, meaning that partial simulation does not tell the whole story.

A number of numbers and a list of ones

AI-generated content may be incorrect.

*Figure 9: Input File*  
A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 10: File Output (Final Simulation)*

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 11: File Output (Partial Simulation)*

Conclusion

Based on the test case, the full simulation takes care of control and data hazards that appear in the pipeline. The input file is read correctly, and the vectors take care of updating register and memory values as needed. A user-friendly interface is created successfully and allows the user to understand what the information means. Some weak points in the project include the partial simulation not updating the pipeline correctly, as it does not have the opportunity to check for hazards, until the next instruction is introduced. There are also limitations on the types of instructions it currently supports. Some things to consider in the future would be to implement an ARM pipeline using Verilog and verify the simulation using System Verilog, for a wider range of instructions.