Legacy MIPS Assembler/Disassembler

Debugging and Revamping the MIPS-Translatron 3000

1. Introduction

Congratulations! You've just been hired as a junior firmware engineer at ByteForge Systems, a company specializing in embedded computing solutions for high-performance robotics and automation.

Your first assignment? Debug and restore an essential internal tool, *MIPS-Translatron 3000*; a MIPS assembler/disassembler that was once the pride of the company but has since fallen into disrepair.

Here's the catch: The original developer, "Old Jim," retired five years ago, leaving behind a cryptic and buggy codebase with little documentation, inconsistent variable names, and a few "temporary fixes" that became permanent. You've been handed this mess with a simple directive: *"Fix it ASAP."*

ByteForge Systems is on the verge of launching its latest product, the Titan-9 Industrial Automation Unit, which relies on a custom MIPS-based microcontroller for its real-time control logic. However, the final firmware validation process requires the *MIPS-Translatron 3000* to accurately translate compiled machine code back into assembly to verify instruction integrity.

Unfortunately, during initial tests, the tool produced bizarre outputs—registers randomly swapping, opcodes misinterpreted, and in one case, an instruction that somehow asked the CPU to make coffee (we wish).

With the Titan-9 launch *just weeks away*, your team is counting on you to debug, clean up, and improve the tool before it's too late. No pressure.

2. Project Setup

Download the *MIPS-Translatron* codebase from here.

2.1. MacOS Setup

Mac setup steps can be found here.

2.2. Windows Setup

For Windows users, please check the attached file for Project Setup with VS Code
Windows Users Project Setup with VS Code.docx

3. Project Deliverables

The project is broken down into two phases. Each phase has a deliverable, see below.

3.1. Bug Fixing & Feature Enhancement

In this phase of the project, your primary task is to take the existing assembler/disassembler codebase and bring it up to a functional and maintainable state. This will require extensive debugging, refactoring, and documentation to ensure that future engineers can easily understand and work with the tool. The current version of the MIPS-Translatron 3000 is plagued with issues, including incorrect opcode translations, faulty register mappings, and a general lack of clear organization. Your job is to fix these errors systematically and improve the tool's overall reliability.

Before Jim retired, he fully tested and documented three functions: **ADD, BNE, ADDI**. These functions were implemented with proper formatting, clear logic, and thorough inline comments explaining each step of the translation process. These three functions serve as a *gold standard* for how an instruction should be correctly processed within the assembler/disassembler. As you work through debugging and implementing new instructions, you should *refer to these functions as a guide* to ensure consistency in structure, error handling, and efficiency.

As you work through the code, you will also need to document your changes. This includes **adding detailed comments to explain complex logic, renaming ambiguous variables and functions for clarity, and writing a brief summary of any modifications made**. The goal is to ensure that anyone picking up the project in the future can quickly grasp how the assembler/disassembler works without having to reverse-engineer poorly written code.

Once the core functionality is restored, you will *extend the tool's capabilities* by **implementing support for three new MIPS instructions: MULT, ORI, and SLTI**. The specifics of these instructions are provided below. Your task will be to integrate them into both the assembler and disassembler, ensuring they are correctly encoded into machine code and decoded back into human-readable assembly.

Assembly Instruction	Instr. Format	op op/funct	Meaning	Comments
`mult $rs, $rt`	R	0/24	`Hi, Lo = $rs * $rt`	rd = coprocessor register (e.g. epc, cause, status)
`ori $rt, $rs, imm`	I	13	`$rt = $rs | imm`	Logical OR, unsigned constant
`slti $rt, $rs, imm`	I	10	`if($rs<imm) $rt = 1; else $rt = 0`	

By the end of this phase, you should have a fully functional assembler/disassembler that not only translates MIPS instructions accurately but also includes additional functionality to support new instructions.

3.2. Final Testing & Robustness Improvements

ByteForge's testing division has encountered a serious issue that could jeopardize the Titan-9 launch. A batch of compiled firmware has been corrupted due to a faulty EEPROM writer, causing *bit flips* in certain machine instructions. These errors could lead to unintended behavior in the Titan-9's control system, and it is now your responsibility to analyze and correct the issue. Your task is to **take the provided corrupted machine code, identify where the bit flips have occurred, and restore the original instructions.**

The corrupted machine code can be found here. Imagine this file was Terabytes long–what would you do to deal with this size file? Suggest an automated and scalable solution to this problem **(implementing & documenting it = extra credit–see section 4.1)**.

To accomplish this, you will need to carefully inspect the binary representation of each instruction and compare it to valid opcode and register formats. By analyzing patterns and detecting anomalies in the bit sequences, you can determine which fields—such as opcodes, registers, or immediate values—have been altered. Once the errors have been identified, you must apply the necessary corrections to restore the proper machine instructions. After fixing the bit flips, you will **run the corrected machine code through your disassembler to verify that it translates back into the expected assembly instructions.**

4. Deliverables:

Deliverable 1 (4%): A **fully corrected codebase with clear comments highlighting your edits and/or new implementations**. This is derived from the corrupted machine instructions that you were provided (do not be like Old Jim)..

Deliverable 2 (4%): Machine code where the corrected bits are **highlighted**, as well as the **recompiled assembly code** from running the fixed machine code through your disassembler from Deliverable 1. Compile all this information and figures in a *single document (follow lab report style, this document should be readable)*.

Deliverable 3 (2%): Create a **presentation** explaining your bug fixes, documentation process, debugging process, highlighting how you identified bugs, and bit flips in the corrupted machine code demonstrating how your assembler/disassembler successfully converts the corrected instructions.

Notes: *Each team member must have their first and last names listed next to the part of the project that they were responsible for (this is mandatory). All team members should be fully prepared to present their work and discuss it with peers/instructor, and answer questions.*

4.1. Bonus Challenge: Implementing a Test Bench

An additional challenge is to design a **test bench** that systematically verifies the correctness of your assembler/disassembler. A proper test bench should take a set of valid assembly instructions (irrespective of file size), convert them into machine code using the assembler, and then run the resulting binary back through the disassembler to ensure that the output matches the original assembly.

To make the test bench more robust, consider incorporating automated validation by comparing expected outputs with actual results and flagging any discrepancies. Additionally, the test bench should include cases that test edge conditions, such as immediate value limits, register overflows, and rare instruction formats. A well-designed test bench will serve as a **valuable tool** for ensuring that your assembler/disassembler remains functional and reliable, even when modifications are made in the future.