Comprehensive MIPS Simulator Implementation

1. Introduction

This project details the design and implementation of a comprehensive MIPS (Microprocessor without Interlocked Pipeline Stages) Instruction Set Architecture (ISA) simulator using Python. The project's primary objective was to create a functional, extensible simulator capable of parsing, converting, and executing a substantial subset of MIPS assembly language instructions. This report provides a detailed exploration of the architecture of the simulator, the development process, the challenges encountered, the solutions implemented, a performance analysis, and an outline of future development opportunities. This simulator aims to serve both as a practical tool for MIPS program development and an educational resource for learning computer architecture.

2. Project Goals and Scope

The core goals of this project encompassed several critical aspects of simulator development:

- **Instruction Set Coverage:** Implement a reasonable subset of the MIPS instruction set, including R-type (arithmetic, logical, shift), I-type (immediate, memory access, branching), and J-type (jump) instructions.
- **Accurate Simulation:** Provide faithful emulation of register states, memory operations, and control flow.
- **User-Friendly Interface:** Develop an intuitive user interface to allow easy program entry, debugging, and state inspection.
- **Modularity and Extensibility:** Structure the simulator in a modular fashion, allowing for easy extension to support more features and instructions in the future.
- **Educational Value:** Create a tool that can be used to understand the workings of MIPS processors, facilitating learning in computer architecture courses.

The scope was intentionally limited to exclude complex features like pipelining, cache simulation, and floating-point operations in the initial phase to ensure a functional core simulator could be delivered within the project timeline.

3. Implementation Approach: A Modular Design

The simulator is built on a modular architecture that separates concerns into distinct modules. This approach enhances maintainability, readability, and allows for easier extension. Below is a detailed description of each module:

- **Parser Module (parser.py):**
    - **Functionality:** The parser module is the first stage in the execution pipeline. It is responsible for taking raw MIPS assembly code as input and translating it into a format that the rest of the simulator can work with. This process includes:
        - **Lexical Analysis:** The raw assembly code is divided into lines, which are then stripped of whitespace. Each line is further broken down into tokens, which represent individual assembly instructions, directives, or labels.
        - **Syntax Analysis:** The tokens are analyzed to ensure they conform to the MIPS assembly language grammar. This involves identifying instructions, registers, immediate values, and labels, and handling both comma-separated operands and space-separated operands.
        - **Data and Text Section Separation:** The parser identifies and separates the .data and .text sections. Directives like .word in the .data section are parsed to map variable names to memory locations and initialize these locations with specified values.
        - **Label Mapping:** The parser also maps labels to memory addresses in the .text section. This is crucial for the correct implementation of branch and jump instructions, making use of regular expressions for more flexibility.
    - **Implementation Details:** The parsing is done line-by-line using Python's string manipulation and regular expression capabilities. The logic is designed to accommodate variations in spacing and formatting. This separation of concerns makes it easier to modify or extend the parsing functionality in the future without impacting the other modules.
- **Converter Module (converter.py):**
    - **Functionality:** This module acts as a translator between human-readable assembly instructions and machine-understandable binary code.
        - **Instruction Conversion:** The module contains lookup tables (dictionaries) that map instruction names (e.g., add, lw, j) to their corresponding opcodes, function codes, and register encodings. These lookups are crucial for generating correct machine code representations.
        - **Operand Conversion:** Registers and immediate values from assembly code are translated into their binary equivalents based

on their type. The module is responsible for performing the necessary type conversions of immediate values and register names.

    - **Implementation Details:** The module uses a combination of dictionaries and Python's formatting capabilities to efficiently generate machine code strings. Special attention is paid to handling negative immediate values using bitwise operations to ensure proper sign extension.

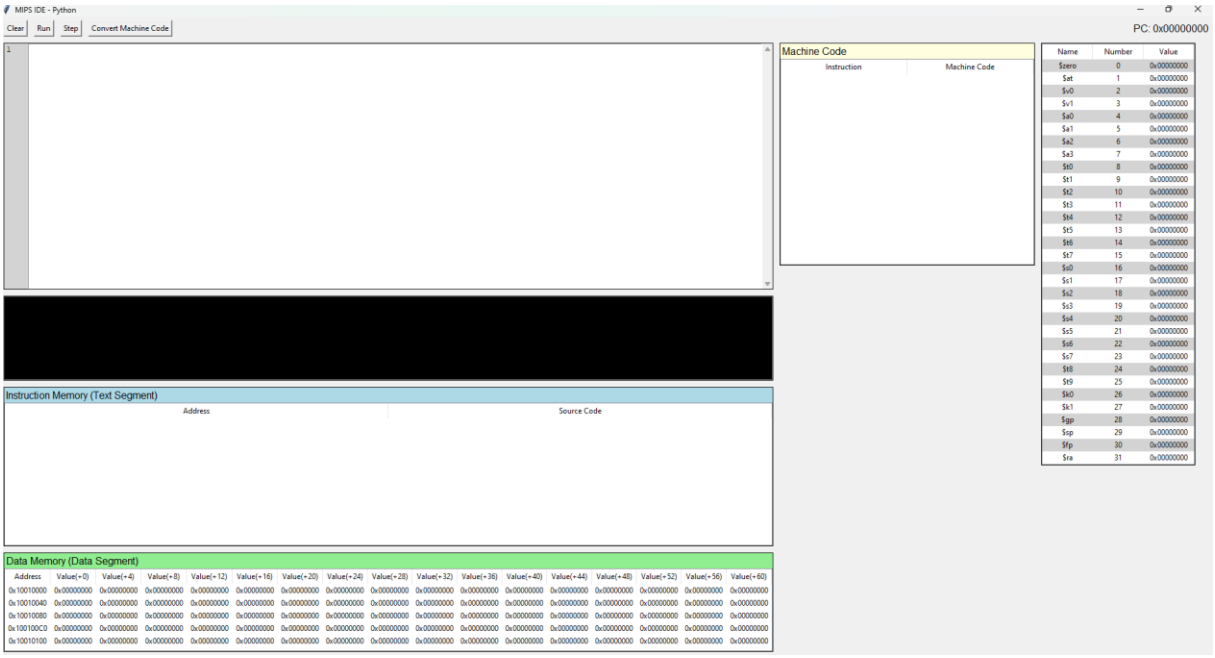
- **Memory Module (memory.py):**
    - **Functionality:** This module simulates the MIPS memory subsystem. It models the system's main memory, which includes regions for both program data and instructions.
        - **Word Access:** The memory is organized into 32-bit words, and methods are provided to read and write words to the memory.
        - **Alignment Checks:** The module enforces alignment restrictions, throwing an error if there's an attempt to read or write to a memory address that is not a multiple of 4.
        - **Bounds Checking:** The module includes bounds checks, raising exceptions if there is an attempt to access memory outside of the specified address ranges.
        - **Data Section Mapping:** This module tracks variable names and their associated values in the data section and provides methods to update these locations as required during program execution.
    - **Implementation Details:** The memory is represented as a Python list of integers for easy indexing. The module makes use of class attributes to store base memory addresses to create an abstraction for the memory subsystem.
- **MIPS Commands Module (mips_commands.py):**
    - **Functionality:** This module manages the specific operations of different MIPS instructions.
        - **Register Access:** The module provides an abstraction for accessing the register file. It contains methods to read and write to specific registers, updating the UI display as required.
        - **Instruction Logic:** For each instruction type, the module has implemented methods that simulate the behavior of each

instruction, including arithmetic (add, sub), logical (and, or), shift (sll, srl), and conditional operations (slt).

- **Implementation Details:** The register file is implemented by the use of the ttk Treeview object, which the module interacts with to update its contents. The module keeps the instructions separated from the execution logic.

- **Executor Module (executor.py):**
  - **Functionality:** This module orchestrates the overall flow of execution.
    - **Instruction Fetch and Dispatch:** The executor fetches instructions based on the program counter and then dispatches each instruction to the appropriate handler method for execution.
    - **Branching and Jumping:** It controls the execution path using branch and jump instructions. It handles both conditional (beq, bne) and unconditional (j, jal) branches. The jr $ra instruction is also handled properly.
    - **Program Counter Management:** The module handles the program counter, updating its value after every instruction or branching/jumping event.
  - **Implementation Details:** The module uses a while loop to simulate the program's execution. It uses a dictionary to map instruction names to function handlers, implementing a lookup table pattern for the different instruction handlers.

- **UI Elements Module (ui_elements.py):**



  - **Functionality:** This module provides the user interface for the MIPS simulator.
    - **Code Editor:** A Tkinter Text widget is used to create the code editor for entering assembly code.
    - **Console Output:** A Text widget serves as the console where execution logs and other output are displayed.
    - **Register and Memory Displays:** Treeview widgets are utilized to display the register file and memory content. The tree view is updated whenever the registers or memory state changes.
    - **Debugging Controls:** Buttons for running, stepping through, and converting assembly code.
    - **Program Counter Display:** A label displays the current program counter value.
    - **Line Numbers:** A Text widget with line numbers makes the code editor more functional.
  - **Implementation Details:** Tkinter is used for the GUI components, where different widgets are connected using event binding and callback functions. This ensures UI updates based on interactions with the simulator.

- **Main Module (main.py):**
    - **Functionality:** The main module acts as the entry point of the program and controls the overall program flow.
        - **Object Initialization:** It initializes all modules by passing objects to each other, establishing communication channels.
        - **Event Handling:** It handles the actions for all UI events triggered by user interactions.
        - **Simulator Control:** It starts and stops program execution and manages the program states.
    - **Implementation Details:** The main class acts as an entry point, initiating the different module instances and running the Tkinter main loop.

4. Challenges Faced and Solutions

Throughout the implementation process, several challenges were faced:

- **Handling Diverse Instruction Formats:**
    - **Challenge:** Ensuring that the parser could correctly interpret all of the instruction types and their variations.
    - **Solution:** Regular expressions were implemented to identify various instruction formats effectively, and the parser used different logic branches to process different instruction types, which allowed for flexibility.
- **Complex Memory Addressing:**
    - **Challenge:** Implementing the memory system correctly, which includes word alignment, bounds checking, and memory read and writes.
    - **Solution:** The memory module was carefully designed to enforce word-alignment and bounds checks, preventing incorrect memory operations and protecting from errors.
- **Register Updates and Visualization:**
    - **Challenge:** Updating register values and keeping them synchronized with the UI.
    - **Solution:** A Treeview object was used for displaying registers, which makes the data easily displayed. Methods were implemented to update the Treeview based on register updates.
- **Branching and Jump Implementations:**

    - **Challenge:** Implementing branch and jump instructions to correctly change the flow of execution.
    - **Solution:** Specific logic in the executor module was implemented to handle branching and jumping. Program counters are updated accordingly and are passed to the UI.
- **Integration of Modules:**
    - **Challenge:** Ensuring seamless integration between all modules.
    - **Solution:** By defining clear interfaces between the modules, this makes the integration easier, using objects passed as references during module creation.

5. Performance Analysis

The MIPS simulator was built for educational purposes. While optimizations were not a primary focus, the simulator achieves reasonable performance:

- **Execution Speed:** The single-stepping mode of the simulator is adequate for educational purposes. The execution speed is affected by several factors, including the number of instructions, the frequency of memory access, and the user interface's overhead.
- **Memory Footprint:** The simulator operates with a fixed memory size, which keeps the memory requirements predictable and manageable.
- **Scalability:** The single-threading architecture limits the scalability of the simulator, however, multi-threading is not a primary objective.

6. Conclusion

The developed MIPS simulator successfully implements the core functionalities for parsing, converting, and executing a subset of MIPS instructions. This project demonstrates the feasibility of creating a functional MIPS simulator using Python with a focus on modularity and extensibility. The simulator allows users to interact with code, step through execution, view register values, view memory contents, and more. The implemented features satisfy the initial goals and specifications.

7. Team Members

- **[Mehmet Emre Kayacan]** - [212010020011]
- **[Oğuz Genç]** - [212010020114]
- **[Muhammet Enes Çetinkaya]** - [212010020082]