RISC-V Simulator

Krishna Teja Pulipati(CS23BTECH11028)
Harshith Peddineni(CS23BTECH11045)

September 2024

Contents

1 Introduction

The primary objective of this project is to design and implement a RISC-V simulator capable of running RISC-V assembly instructions using a C++ program. This project focuses on supporting a subset of the RISC-V instruction set, specifically the base integer instruction set , which is commonly used in educational and industrial applications. The simulator is designed to handle basic instruction formats, such as R-type,I-type,S-type,B-type,U-type and J-type instructions but doesn't support for pseudo-instructions that simplify assembly programming.

2 Overall Design

This section details the design and implementation of the RISC-V simulator, highlighting the structure and functionality of key components designed to run the RISC-V assembly code, supporting multiple instruction formats.The simulator processes the assembly code into three phases:

1. **Loading Phase:** Loads the input.s file and stores the instruction data in the form of a struct. Which contains all the required data.
2. **Parsing Phase:** Reads each line of assembly code, parses it into an Instruction object, and stores necessary label information.
3. **Executing Phase:** Executes the code as per the user commands in the terminal

The assembler consists of 10 headers and their corresponding source files.They are as follows :

- **CPU** : Contains the register data and functions related to initialisation and updating of registers
- **Data** : Holds the data of all instructions,Program counter and map of labels and their line numbers of the assembly code.
- **Encoder** : Translates Instruction objects into their corresponding hexadecimal machine code.
- **Executor** : Contains the functions required for execution of code line by line and data of breakpoints.

- **Instruction** : Instruction data type is defined here and it also manages different RISC-V instruction formats and their respective fields such as registers, opcode and immediate.
- **Linerunner** : Contains the function for execution of each instruction and an array of function pointer which in turn point towards the functions
- **Memory** : Contains the stack pointer and all the functions related to handling the memory of system which are used in store, load instructions and for print memory in the termial.
- **Parser** : Converts lines of assembly code into structured Instruction objects.
- **RISC-V** : Contains the information about all the register aliases and instruction's opcode, func3 and func7 in this ISA.
- **Utilities** : Handles the basic functions required frequently in the program

3 Implementation Details (File Breakdown)

3.1 Main Program (`main.cpp`)

Purpose: Serves as the entry point for the RISC-V simulator, facilitating user interactions and managing the flow of execution.

Key Functionalities:

- **File Loading**: Prompts the user to load an assembly code file. The program checks if the file is successfully opened and handles any errors appropriately.
- **Data and Instruction Parsing**: Reads the contents of the loaded file, differentiating between data and text sections. It populates the global data structures, such as `Lines`, `instructionMemory`, and calls `storeData()` to store data variables in memory.
- **User Commands**: Supports a command loop that interprets user commands such as:

- `load <filename>`: Loads the file containing RISC-V assembly code. (all registers and memory get initialized to the default value.)
- `run`: Executes the loaded instructions until a breakpoint or the end of the instructions.
- `regs`: Displays the current state of the CPU registers.
- `mem <addr> <count>`: Displays memory locations equal to count number starting from address addr in the data section. (assume Little Endian format).
- `step`: Executes a single instruction and updates the program state.
- `show-stack`: Displays the current state of the call stack.
- `break <line>`: Sets a breakpoint at a specified line number.
- `del break <line>`: Deletes a specified breakpoint.

- **Command Handling**: Manages command input from the user, providing feedback for invalid commands and ensuring proper command flow.

Role: This file is crucial for the simulator's operation, providing the main user interface and controlling the execution flow of the program. It initializes the CPU, loads instructions and data, and allows for user interaction to control the simulation.

3.2 Parser (`parser.cpp`)

Purpose: Responsible for parsing RISC-V assembly code into a machine-readable `Instruction` structure.
Key Functions:

- `parseInstruction()`: The main function that takes an assembly line as input, tokenizes it, and converts it into an `Instruction` object.

  Details:

  - The function begins by tokenizing the input line using `tokenize()`. If the line is empty, it returns an error message indicating an "empty line".

- It checks if the first token (the instruction mnemonic) exists in the `instructionData` map. If not, it returns an error indicating that the instruction is invalid.
- Depending on the instruction format (B-Type, R-Type, I-Type, etc.), the function assigns values to fields like `rd`, `rs1`, `rs2`, and `immediate`.
- Registers are validated using helper functions like `rigToInt()` to ensure they fall within valid ranges (0-32). If an invalid register or immediate value is encountered, appropriate error messages are set.

Error Handling:

- The function handles errors related to invalid instruction mnemonics, registers, and immediate values. If any such issue arises, an error message is stored in the `Instruction` object.

Role: Converts a single line of RISC-V assembly into a structured `Instruction` format that can be used by the simulator.

3.3 Instruction (`instruction.cpp`)

Purpose: Responsible for converting various RISC-V instruction formats into hexadecimal representations.
Key Structures:

- `InstructionFormat`: An enum class that defines the format of an instruction. Possible values include `R_TYPE`, `I_TYPE`, `S_TYPE`, `B_TYPE`, `U_TYPE`, `J_TYPE`, and `UNKNOWN`.
- `InstructionInfo`: A struct that holds metadata about an instruction, including its format, `opcode`, `funct3`, and `funct7` fields.
- `Instruction`: A struct that represents a RISC-V instruction. It contains:
  - `instructionInfo`: An instance of `InstructionInfo` containing the instruction's format and other metadata.
  - `mnemonic`: The instruction's textual representation (e.g., "add").

- `rd`, `rs1`, `rs2`: Destination and source registers for the instruction.
- `label`: A string for storing branch or jump labels.
- `immediate`: The immediate value for the instruction, if applicable.
- `error`: A string used to store error messages encountered during parsing.

Key Functions:

- `R_type_to_hex(const Instruction R)`: Converts an R-type instruction into its hexadecimal representation.
- `I_type_to_hex(const Instruction I)`: Converts an I-type instruction into its hexadecimal representation.
- `S_type_to_hex(const Instruction S)`: Converts an S-type instruction into its hexadecimal representation.
- `B_type_to_hex(const Instruction B)`: Converts a B-type instruction into its hexadecimal representation.
- `U_type_to_hex(const Instruction U)`: Converts a U-type instruction into its hexadecimal representation.
- `J_type_to_hex(const Instruction J)`: Converts a J-type instruction into its hexadecimal representation.
  Details:
  - Each function computes the hexadecimal representation of the instruction by constructing the corresponding binary fields based on the instruction format.
  - Fields such as `opcode`, `funct3`, `funct7`, `rd`, `rs1`, `rs2`, and `immediate` are shifted and combined to form the final instruction in hexadecimal format.
  - For instructions where the immediate values are split (e.g., B-type and J-type), bit manipulation is performed to extract and combine the bits appropriately.
  - A `stringstream` is used to convert the final binary value into a hexadecimal string, ensuring it is padded to 8 characters for standard formatting.

Role: Encodes different instruction formats into their hexadecimal form, which is crucial for the binary representation of instructions in the simulator.

3.4 Encoder (`encoder.cpp`)

Purpose: Responsible for converting parsed instructions into their hexadecimal machine code equivalent.

Key Function:

- `InstructionToHex(Instruction instruct)`: Converts a parsed instruction into its hexadecimal representation.

Details:

- The function first checks for any errors within the instruction. If an error is found, it returns `"error!!"`.
- A `switch` statement is used to determine the instruction's format (R, I, S, U, J, or B type).
- For each instruction type:
  - `R_type_to_hex()` is called for R-type instructions.
  - I-type instructions, such as `slli`, `srli`, and `srai`, involve specific bit manipulations before passing to `I_type_to_hex()`.
  - S-type and U-type instructions are processed by their respective conversion functions.
  - For J-type instructions, the function checks for label validity in `labelData` and calculates the immediate value based on the label and program counter.
- For B-type instructions, similar label handling occurs, and the immediate value is calculated accordingly.
- If an unknown instruction format is detected, an error message is set, and the function returns `"error!!"`.

Role: This file plays a critical role in the simulator by transforming high-level assembly instructions into a machine-readable format, enabling execution in the RISC-V architecture.

3.5 Executor (`executor.cpp`)

Purpose: Responsible for executing the instructions in the simulator and managing breakpoints during execution.

Key Functions:

- `RunTillBreakPoint(int totalLines)`: Executes instructions until a breakpoint is encountered or all instructions have been executed.
- `executeInstruction()`: Executes a single instruction and manages the program counter.

Details:

- The `RunTillBreakPoint` function checks if the current program counter (PC) has a breakpoint. If so, it executes the instruction at that point and then enters a loop to continue execution until the PC reaches the total number of lines or another breakpoint is hit.
- During execution, if a breakpoint is hit, the function prints a message and stops further execution.
- The `executeInstruction` function retrieves the instruction at the current PC.
- For B-type and J-type instructions, it checks if the label exists in `labelData` and calculates the immediate value based on the label.
- The instruction's mnemonic is looked up in `functionMap` to determine the appropriate function to execute.
- Before executing the instruction, it formats and prints the current instruction and program counter in hexadecimal format.
- After execution, the program counter is incremented, unless a branching instruction alters its flow.

Role: This file is crucial for the operation of the simulator, as it handles the execution of instructions and allows for debugging through breakpoints.

3.6 Memory Management (`memory.cpp`)

Purpose: Handles memory operations, including data storage, retrieval, and management of the call stack in the simulator.

Key Functions:

- `ShowMemory(std::string address, int count)`: Displays memory contents starting from a specified address for a given count.
- `showCallStack(int numLines)`: Displays the current state of the call stack.
- `fechMemory(int address, int Numbytes)`: Fetches data from memory at the specified address over a specified number of bytes.
- `storeMemory(int address, int Numbytes, int value)`: Stores data in memory at the specified address for a specified number of bytes.
- `storeData(std::vector<std::string> StoreData)`: Stores data in memory based on specified directives (e.g., '.byte', '.half', '.word', '.dword').

Details:

- The `ShowMemory` function converts the provided address to an appropriate format and prints the memory content for each address in the specified range.
- The `showCallStack` function checks if the program counter exceeds the total number of lines; if not, it iterates through the stack pointer and prints labels and line numbers in the call stack.
- The `fechMemory` function reads the specified number of bytes from memory, assembling the data into an integer by shifting and combining byte values.
- The `storeMemory` function writes the specified integer value into memory, byte by byte, starting from the provided address.
- The `storeData` function handles storing data in memory based on the type of data directive provided. It calls `storeMemory` with the appropriate byte size based on the directive.

Role: This file is essential for managing memory operations in the simulator, enabling the storage and retrieval of data as well as facilitating function calls and returns through the call stack.

3.7 CPU Management (`cpu.cpp`)

Purpose: Manages CPU operations, including register initialization and manipulation within the simulator.

Key Functions:

- `InitCPU(CPU *cpu)`: Initializes all CPU registers to zero.
- `GetRegister(CPU *cpu, int regIndex)`: Retrieves the value of a specified CPU register.
- `SetRegister(CPU *cpu, int regIndex, int value)`: Sets the value of a specified CPU register, ensuring that register 0 remains zero.
- `ShowRegisters(CPU *cpu)`: Displays the values of all CPU registers in hexadecimal format.

Details:

- The `InitCPU` function iterates through all 32 registers, setting each to zero to prepare the CPU for execution.
- The `GetRegister` function retrieves the current value of the specified register by its index.
- The `SetRegister` function updates the value of the specified register, with a check to ensure that register 0 remains unchanged.
- The `ShowRegisters` function prints the current values of all registers in a formatted manner, converting each value to hexadecimal and ensuring proper alignment in the output.

Role: This file is critical for managing CPU state within the simulator, allowing for register manipulation and visualization of the CPU's current status during execution.

3.8 Utility Functions (`utilis.cpp`)

Purpose: Provides various utility functions for string manipulation and tokenization to assist with instruction parsing in the simulator.

Key Functions:

- `tokenize(std::string &line)`: Splits a given line of text into meaningful tokens based on delimiters and special characters.
- `strToInt(const std::string &num)`: Converts a string representation of a number (in decimal, hexadecimal, or binary) into an integer.
- `rigToInt(const std::string &num)`: Converts a register-like string (e.g., `x1`) into its corresponding integer index.
- `split(const std::string &str)`: Splits a string into words based on whitespace delimiters.

Details:

- The `tokenize` function removes leading spaces from the line, extracts tokens, and handles labels by storing them in the `labelData` map. It also processes special cases for parentheses.
- The `strToInt` function checks for negative signs and identifies the number base (decimal, hexadecimal, or binary) to convert the string to an integer correctly.
- The `rigToInt` function extracts the integer part of a register-like string, which is typically prefixed with an `x`.
- The `split` function utilizes a string stream to break the input string into individual words.

Role: This file plays a crucial role in processing input data, allowing the simulator to interpret assembly instructions accurately and manage data effectively.

3.9 Data Structures (`data.cpp`)

Purpose: Defines global variables and data structures used throughout the simulator.

Key Data Structures:

- `Instruction Lines[4096]`: An array to store instructions loaded from the assembly code, allowing for a maximum of 4096 instructions.
- `std::unordered_map<std::string, int> labelData`: A hash map that associates labels in the assembly code with their corresponding line numbers, facilitating quick lookups during instruction execution.
- `int ProgramCounter`: A variable to track the current position in the instruction set being executed, allowing for the sequential execution of instructions.
- `int EmptyLines[4096]`: An array that may be used to track the status of lines in the instruction array, indicating whether a line is empty or has valid instruction data.

Role: This file serves as a central repository for key global variables that are essential for the operation of the RISC-V simulator. By maintaining instructions, labels, and the program counter, it provides the necessary context for executing assembly code.

3.10 Instruction Execution Functions (`linerunner.cpp`)

Purpose: Contains functions that execute various RISC-V instructions by manipulating CPU registers and memory.

Role: This file implements the core instruction execution logic for the RISC-V simulator, allowing the simulator to execute various types of instructions effectively by manipulating CPU registers and memory.

4 Challenges and Solutions

During the implementation, several challenges were encountered:

- **Handling Complex Instruction Formats:** The variety of RISC-V instruction formats required careful design of parsing and encoding functions. This was solved by creating dedicated functions for each format.
- **Label Resolution:** Managing forward references to labels was tricky. The solution involved a two-pass process: one for identifying labels and another for resolving them during encoding.
- **Error Handling:** Handling the errors at each step of parsing like
  (a) Undefined instructions
  (b) Incomplete instructions that are having missing registers or more number of registers.
  (c) Improperly given instruction that is giving register in place of immediate,giving non existent register etc,.
  (d) Handling the comments.
  (e) In the lui if the value is greater than 20 bits it automatically truncates the value upto 20 bits.

5 Makefile Overview

The following is an overview of the Makefile used for the RISC-V simulator project.

5.1 Targets

- `all`: This is the default target that builds the final executable `riscv_sim` and also calls the `cleano` target to remove object files.
- `final`: This target links all object files together to create the executable `riscv_sim`. It depends on the object files generated from the source files.
- `cleano`: This target removes all object files from the `src` directory.
- `clean`: This target calls `cleano` and additionally removes the final executable `riscv_sim.exe`.

5.2 Rules for Compiling

Each source file in the `src` directory is compiled into an object file using the following pattern rules:

- `data.o`: Compiles `data.cpp`.
- `encorder.o`: Compiles `encorder.cpp`.
- `instruction.o`: Compiles `instruction.cpp`.
- `main.o`: Compiles `main.cpp`.
- `parser.o`: Compiles `parser.cpp`.
- `utils.o`: Compiles `utils.cpp`.
- `cpu.o`: Compiles `cpu.cpp`.
- `memory.o`: Compiles `memory.cpp`.
- `executor.o`: Compiles `executor.cpp`.
- `lineRunner.o`: Compiles `lineRunner.cpp`.

5.3 Cleanup Commands

The `clean` and `cleano` targets ensure that the workspace is clean by removing unnecessary object files and the executable. The command `make clean` removes all object files and the executable, while `make cleano` only removes the object files.

5.4 Example Usage

To build the simulator, run:

```
make
```

To clean up object files, run:

```
make cleano
```

To remove all generated files, including the executable, run:

```
make clean
```

6 Conclusion

The development of this RISC-V assembler represents a significant achievement in understanding and implementing low-level machine code.The project successfully translated the RISC-V assembly code into machine code,supporting translation of wide range of instruction formats like R,I,S,U,B and J types.Through this process, the project met its primary objectives, including the correct parsing of assembly code, the accurate encoding of instructions into hexadecimal format, and the management of labels and addresses within the code.Also popss out errors very effectively in each line.

This project also provided valuable insights into the RISC-V architecture and assembly language programming. It reinforced the importance of precise bitwise operations, careful management of instruction formats, and the complexities involved in translating human-readable code into machine-executable instructions. Moreover, the project underscored the significance of rigorous testing and validation in the development of assembly-level tools

In conclusion, the RISC-V assembler has achieved its goals of accurately translating assembly instructions into machine code, offering a solid foundation for further enhancements.