RISC-V Simulator Report

Bhumin Hirpara, CS23BTECH11009

3rd October, 2024

Contents

1 Introduction

This project aims to create a modular and extensible simulator capable of executing a variety of RISC-V instructions, including R-format, I-format, S-format, B-format, J-format, and U-format instructions. With robust error handling mechanisms in place, the simulator is designed to detect and report common errors, ensuring a smoother user experience. In addition to executing instructions, the simulator offers features for inspecting register values, memory contents, and the call stack.

2 Main Functions of 'main.cpp'

The 'main.cpp' file acts as the control hub for the simulator. It manages input parsing, instruction execution, memory handling, and debugging features like breakpoints and stepping through instructions. This section provides an overview of key functionalities in 'main.cpp'.

2.1 File Loading: `loadFile`

The 'loadFile' function is responsible for reading the input file and preparing the simulator to execute the instructions. Here's how it works:

- **Input File Parsing**: The function opens the specified file and reads it line by line. It identifies the '.data' and '.text' sections in the program.
- **Data Section Handling**: When the '.data' section is encountered, the function processes memory-related values such as '.dword', '.word', '.half', and '.byte', storing them in memory using appropriate functions.
- **Text Section Handling**: When the '.text' section starts, the program begins reading and storing the instructions. These instructions will later be executed during the simulation.

2.2 Running the Program: `run`

The 'run' function is used to execute all the instructions in the '.text' section in one go. It performs the following tasks:

- **Continuous Execution**: The 'run' function executes each instruction sequentially until the end of the program is reached or a breakpoint is encountered.
- **Breakpoint Check**: During execution, if the program hits a line marked as a breakpoint, it will pause, allowing the user to inspect memory or registers before continuing.
- **Jump Instructions**: If a jump or branch instruction is encountered, the simulator checks for labels and jumps to the correct line of code.

2.3 Step-by-Step Execution: `step`

The ‘step‘ function allows the user to execute the program one instruction at a time. Here’s how it works:

- **Single Instruction Execution**: Instead of running the entire program, ‘step‘ executes only the next instruction, then pauses.
- **Debugging**: This feature is useful for debugging, as the user can observe how each instruction affects memory and registers one step at a time.
- **Instruction Display**: After each step, the current instruction and its result are printed, allowing the user to verify correctness.

2.4 Memory Inspection: `mem`

The ‘mem‘ command allows the user to inspect memory values at any point in the program. Here’s what it does:

- **Specific Address Lookup**: The user can specify a memory address and count of memory sections.

2.5 Breakpoint Management: `break`

The ‘break‘ function allows the user to set breakpoints in the program. Breakpoints are specific lines where the program will pause during execution. Here’s how it works:

- **Setting a Breakpoint**: The user specifies a line number or label, and the program will stop at this point during execution.
- **Breakpoint Handling**: When the program reaches a breakpoint while running or stepping, it pauses execution, allowing the user to inspect memory, registers, or other state information.
- **Resuming Execution**: After a breakpoint is hit, the user can choose to resume the program using step through instructions with ‘step‘.
- **Deleting Breakpoint**: After analyzing the breakpoints, the user can delete them by using ‘del break‘ and specifying the line number.

2.6 Viewing the Stack: `show-stack`

The ‘show-stack‘ command is used to inspect the current state of the stack:

- **Stack Display**: When this command is executed, the current stack contents are displayed, showing the functions pushed onto the stack.
- **Stack Usage**: This feature is useful for observing the state of the stack during function calls or recursive operations.

2.7 Viewing Registers: `regs`

The ‘regs‘ command allows the user to inspect the values of the processor's registers:

- **Register Values**: When the ‘regs‘ command is issued, the contents of all general-purpose registers are displayed.
- **Debugging Register Changes**: This is useful for debugging, as the user can see how register values change after each instruction is executed.

3 Simulator Header: `simulator.h`

The `simulator.h` file defines essential functions and data structures used across the simulator. It includes function declarations for handling instructions, memory, registers, and the stack, providing a common interface for different aspects of the simulator. The key components are listed below:

- **Instruction Handling**:
  - `void runInstruction(const string &instruction, int &lineNumber, const unordered_map<string, int> &labelAddresses);`
- **Register Operations**:
  - `void printRegisters();`
  - `void resetRegisters();`
- **Memory Management**:
  - `void printMemory(string address, int count);`
  - `void resetMemory();`
  - `void setDoubleword(vector<string> dataValues);`
  - `void setHalfword(vector<string> dataValues);`
  - `void setWord(vector<string> dataValues);`
  - `void setByte(vector<string> dataValues);`
- **Stack Handling**:
  - `void showStack(int extraLines);`
  - `void createStack(unordered_map<string, int> &labelAddresses);`
  - `void handleStack(unordered_map<string, int> &labelAddresses, int lineNumber);`
  - `void deleteStack();`
- **Utility Functions**:

- `string binaryToHex(string &binaryInstruction);`
- `string decimalToHex(ll number, int hexDigits);`

These functions allow the simulator to process instructions, manage memory, and interact with the stack and registers efficiently. The specific details of each function will be covered in later sections.

4 Overall Functionality of 'simulator.cpp'

The 'simulator.cpp' file performs the actual execution of instructions. Each function in this file is dedicated to run a specific instruction format. The execution process involves several steps:

- **Instruction Reading**: Instructions are read from the input, and their components are extracted.
- **Field Extraction**: Each instruction type has specific fields (e.g., registers, immediate values) that are extracted and processed.
- **Instruction Execution**: Different operations are performed on the registers and memory based on the instruction.

5 Instruction Executing Functions

5.1 R-Type Instructions

R-type instructions are used for arithmetic and logical operations involving two source registers and one destination register. These instructions perform computations based on the values stored in the source registers and place the result in the destination register. Examples include addition, subtraction, and bitwise operations.

Function: `runRFormat`

The `runRFormat` function handles the execution of R-type instructions.Below is an explanation of how the function works:

1. **Instruction Parsing**: The function begins by extracting the operation and registers from the given instruction string. It splits the instruction into four parts: the operation, the destination register (`rd`), and the two source registers (`rs1` and `rs2`).
2. **Register Index Conversion**: After parsing the registers, the function converts the register names (`rd`, `rs1`, `rs2`) into corresponding register indices using the `regToIndex` function.
3. **Operation Execution**: Based on the operation string, the appropriate arithmetic or logical operation is performed:

- For `add`, the function adds the values in `rs1` and `rs2`, and stores the result in `rd`.
- For `sub`, the function subtracts the value in `rs2` from `rs1`, and stores the result in `rd`.
- For bitwise operations such as `xor`, `or`, and `and`, the function performs the corresponding logical operations on `rs1` and `rs2`, and stores the result in `rd`.
- For shift operations like `sll` (shift left logical), `srl` (shift right logical), and `sra` (shift right arithmetic), the function shifts the value in `rs1` by the amount specified in `rs2` and stores the result in `rd`.

4. **Error Handling**: If the operation is unsupported or the register format is invalid, the function outputs an error message and returns without performing any action.

5.2 I-Type Instructions

I-type instructions are used for arithmetic, logical operations, and memory accesses where one of the operands is an immediate value. These instructions typically involve one source register, one destination register, and an immediate value. Examples include add immediate, logical operations with immediate values, and load instructions.

Function: `runIFormat`

The `runIFormat` function handles the execution of I-type instructions. Below is an explanation of how the function works:

1. **Instruction Parsing**: The function starts by identifying the operation from the given instruction string, which could be arithmetic, logical, or memory-related. The instruction is split into the operation, the destination register (`rd`), the source register (`rs1`), and the immediate value.

2. **Opcode Determination**: Based on the operation, the corresponding opcode is set:
   - For arithmetic and logical operations like `addi`, `xori`, `ori`, `andi`, and shift instructions like `slli`, `srli`, `srai`, the opcode is `0010011`.
   - For load instructions like `lb`, `lh`, `lw`, `ld`, `lbu`, `lhu`, `lwu`, the opcode is `0000011`.
   - For the `jalr` instruction, the opcode is `1100111`.

3. **Immediate and Register Parsing**: After identifying the operation, the function extracts the immediate value and registers from the instruction. It ensures that the immediate value is a valid decimal number and checks its range:
   - For general instructions, the immediate must be within the range -2048 to 2047.

- For shift operations (`slli`, `srli`, `srai`), the immediate must be between 0 and 63.

4. **Register Index Conversion**: After parsing the registers, the function converts the register names (`rd`, `rs1`) into corresponding register indices using the `regToIndex` function.

5. **Operation Execution**: Based on the operation, the function performs the necessary computation or memory access:

   - For `addi`, the immediate value is added to the value in `rs1`, and the result is stored in `rd`.
   - For `xori`, `ori`, and `andi`, the function performs the corresponding bitwise operation between the value in `rs1` and the immediate, storing the result in `rd`.
   - For shift operations (`slli`, `srli`, `srai`), the value in `rs1` is shifted by the immediate value, and the result is stored in `rd`.
   - For load instructions (`lb`, `lh`, `lw`, `ld`, etc.), the function calculates the memory address by adding the immediate value to the value in `rs1`, and then loads the appropriate number of bytes from memory into `rd`.
   - For `jalr`, the program counter is updated based on the value in `rs1`, and the function stack is adjusted accordingly.

6. **Error Handling**: The function checks for invalid instructions, such as missing commas, invalid registers, or out-of-range immediate values, and reports errors accordingly without performing any action.

5.3 B-Type Instructions

B-type instructions are used for conditional branching based on comparisons between two registers. These instructions determine whether a branch should be taken based on the result of the comparison, and if so, they change the program's execution flow to a target address specified by a label. Common B-type instructions include `beq` (branch if equal), `bne` (branch if not equal), `blt` (branch if less than), and others.

Function: `runBFormat`

The `runBFormat` function handles the execution of B-type instructions. Below is an explanation of how the function works:

1. **Instruction Parsing**: The function extracts the operation, two source registers (`rs1`, `rs2`), and the branch label from the instruction string. The instruction is split into these four parts.

2. **Register Index Conversion**: The function converts the register names (`rs1`, `rs2`) into their corresponding register indices using the `regToIndex` function.

3. **Label Validation**: The function checks if the provided label exists in the `labelAddresses` map. If the label is not found, an error is displayed, and the function terminates.

4. **Branch Condition Execution**: The function checks the specific operation and compares the values of the two registers:

   - For `beq`, the function checks if `rs1` and `rs2` are equal.
   - For `bne`, the function checks if `rs1` and `rs2` are not equal.
   - For `blt` and `bge`, the function performs signed comparisons of `rs1` and `rs2`.
   - For `bltu` and `bgeu`, the function performs unsigned comparisons by converting the register values into their unsigned representations.

   If the condition is met, the program counter (`currentLine`) is updated to the line corresponding to the target label.

5. **Error Handling**: If an unsupported operation is encountered or if the registers or label are incorrectly formatted, the function outputs an error message and returns without performing any action.

5.4 S-Type Instructions

S-type instructions are used for memory operations involving a base register and an offset (immediate value). These instructions compute an effective memory address by adding the immediate value to the base register, and then store the value from the source register at the calculated address. Examples of S-type instructions include `sd` (store double word), `sw` (store word), `sh` (store half word), and `sb` (store byte).

Function: `runSFormat`

The `runSFormat` function handles the execution of S-type instructions. Below is an explanation of how the function works:

1. **Instruction Parsing**: The function begins by extracting the operation, the source register (`rs2`), and the immediate value along with the base register (`rs1`) from the given instruction string. It splits the instruction into three main parts: the operation, `rs2`, and the immediate with base register.

2. **Immediate and Register Extraction**: The immediate value and the base register (`rs1`) are extracted from the portion enclosed in parentheses. The function checks for syntax correctness, including validating the immediate value.

3. **Register Index Conversion**: After extracting `rs1` and `rs2`, the function converts the register names to corresponding register indices using the `regToIndex` function.

4. **Address Calculation**: The effective memory address is calculated by adding the immediate value to the content of the base register (`rs1`).

5. **Memory Store Operation**: Based on the operation string, the function performs the appropriate store operation:

    - For `sd` (store double word), the function stores 8 bytes from `rs2` to the calculated address.
    - For `sw` (store word), the function stores 4 bytes from `rs2` to the calculated address.
    - For `sh` (store half word), the function stores 2 bytes from `rs2` to the calculated address.
    - For `sb` (store byte), the function stores 1 byte from `rs2` to the calculated address.

6. **Error Handling**: If the immediate value is out of range or if any registers are invalid, the function outputs an error message and returns without performing any action.

5.5 J-Type Instructions

J-type instructions are used for jump operations, typically involving a destination register and a label or immediate value. These instructions are responsible for altering the program counter (PC) to jump to a specific memory address, usually represented by a label, while storing the return address in the destination register. A common example is the `jal` (jump and link) instruction.

Function: `runJFormat`

The `runJFormat` function handles the execution of SJ-type instructions. Below is an explanation of how the function works:

1. **Instruction Parsing**: The function begins by extracting the operation, the destination register (`rd`), and the label from the given instruction string. It splits the instruction into three parts: the operation, `rd`, and the label.

2. **Register Index Conversion**: After parsing the destination register (`rd`), the function converts the register name to its corresponding index using the `regToIndex` function.

3. **Label Handling**: The function then checks if the label corresponds to a valid address in the `labelAddresses` map, which stores the addresses of labels defined in the program.

4. **Operation Execution**:

    - If the label is found, the function stores the return address (the address of the next instruction) in the destination register (`rd`).

- It then alters the program counter to jump to the line number associated with the label.

5. **Error Handling**: If any part of the instruction is invalid (e.g., a missing destination register, an invalid label, or an incorrect syntax), the function outputs an error message and returns without performing any action.

5.6 U-Type Instructions

U-type instructions are used for operations that involve a destination register and a large immediate value. These instructions typically load a 20-bit immediate value into the upper portion of a register. Examples include `lui` (load upper immediate) and `auipc` (add upper immediate to PC).

Function: `runUFormat`

The `runUFormat` function handles the execution of U-type instructions. Below is an explanation of how the function works:

1. **Instruction Parsing**: The function begins by extracting the operation, the destination register (`rd`), and the immediate value from the instruction string. It splits the instruction into three parts: the operation, `rd`, and the immediate value.

2. **Register Index Conversion**: After parsing the destination register, the function converts the register name (`rd`) into its corresponding register index using the `regToIndex` function.

3. **Immediate Handling**:

   - If the immediate value is in hexadecimal format (prefixed with `0x`), the function converts it to a decimal value.
   - If the immediate is a decimal value, it checks whether the value is valid and within the allowable range (0 to 1048575).

4. **Operation Execution**:

   - The immediate value is multiplied by 4096 (shifting it left by 12 bits) and then stored in the destination register (`rd`).

5. **Error Handling**: If any part of the instruction is invalid (e.g., missing or malformed register, invalid immediate value), the function outputs an error message and returns without performing any action.

5.7 Register and Memory Management Functions

The following functions are responsible for managing registers and memory in the simulator:

Function: `printRegisters`

The `printRegisters` function outputs the current values of all 32 registers. It

converts the decimal register values to hexadecimal format for display and ensures leading zeros are removed. The output format aligns the register numbers for clarity.

Function: `printMemory`

The `printMemory` function prints the values stored in memory from a specified address for a given count. It checks if the address falls within the valid memory range and outputs the corresponding memory values in hexadecimal format. An error message is displayed if memory access is attempted beyond the allowed sections.

Function: `resetRegisters`

The `resetRegisters` function resets all registers to zero. This is useful for initializing the state of the simulator before a new execution.

Function: `resetMemory`

The `resetMemory` function clears all entries in memory, effectively resetting the memory state for fresh data input.

Function: `setDoubleword`

The `setDoubleword` function stores 64-bit values (double words) from the data section into memory. It handles both decimal and hexadecimal formats, converting decimal values to hexadecimal and ensuring the values are padded to the appropriate length before storing them in memory in 8-byte increments.

Function: `setHalfword`

The `setHalfword` function stores 16-bit values (half words) in memory. Similar to `setDoubleword`, it processes both decimal and hexadecimal input, padding the values as needed and storing them in 2-byte increments.

Function: `setWord`

The `setWord` function stores 32-bit values (words) into memory. It manages both input formats and ensures that the values are properly formatted and stored in 4-byte increments.

Function: `setByte`

The `setByte` function stores 8-bit values (bytes) from the data section into memory. It processes both decimal and hexadecimal formats, padding values to ensure they are two characters long before storing them.

Function: `showStack`

The `showStack` function displays the current call stack, showing the functions that have been called during execution. If the stack is empty, it indicates that the execution is complete.

Function: `createStack`

The `createStack` function initializes the call stack by pushing the main function onto it, provided its address is known.

Function: `handleStack`

The `handleStack` function updates the current function's position in the stack after each line of execution, ensuring that the call stack reflects the latest function calls.

Function: `deleteStack`

The `deleteStack` function empties the call stack after execution is complete, cleaning up any stored function calls for future runs.

6 Makefile Overview

The ‘Makefile‘ automates the build process, specifying rules for compiling and linking the necessary files. It includes:

- **Compilation Rules**: Defines how to compile each ‘.cpp‘ file into an object file.
- **Linking Rules**: Specifies how to link object files to create the final executable.
- **Dependencies**: Manages file dependencies to ensure that files are recompiled only when necessary.
- **Clean Up**: Includes a rule to remove intermediate files and clean up the build directory.

7 Error Handling in Detail

- **Argument Count Mismatch**: Each instruction type expects a specific number of arguments. For example, R-type instructions need three registers, while I-type instructions require two registers and one immediate value. The error handling ensures that any deviation from this count is reported.
- **Missing Commas**: Proper argument separation is crucial. If commas are missing between arguments, the parser will throw an error, specifying the line and nature of the format issue.
- **Invalid Register Names**: Register names must adhere to RISC-V conventions. For instance, ‘x0‘ to ‘x31‘ and there aliasesare valid but anything else is considered invalid. An error message will indicate incorrect register names.
- **Empty Lines**: Empty lines or lines with only whitespace are skipped during parsing. This prevents errors from blank lines and keeps the parsing process clean.
- **Immediate Value Out of Range**: Immediate values are constrained by instruction formats (e.g., 12-bit for some I-type instructions). Values exceeding these constraints trigger an error, ensuring that only valid values are processed.
- **Invalid Labels**: Labels must be correctly defined and referenced. Errors are raised if labels are missing, duplicated, or incorrectly used, particularly in branch or jump instructions.
- **Incorrect Instruction name**: If the instruction name is incorrect than an error will be thrown pointing the line number and asking to give correct instruction.

8 Limitations of the Code

While the implementation functions correctly, there are a few limitations:

1. **No Support for Pseudo Instructions**: The code does not handle pseudo instructions like 'li', 'mv', etc. These would require additional logic to translate.

2. **Number Conflicts with Labels**: If a label and a number share the same name, the code may misinterpret the number as a label. This issue can lead to incorrect encoding, and a more sophisticated naming scheme might be needed.

3. **Hexadecimal Not Allowed for Immediate Values in Certain Instructions**: Certain instructions require immediate values in decimal format. Using hexadecimal immediate values can lead to encoding errors, as the code expects decimal values.

4. **Only Labels are Allowed in B-Type and J-Type Instructions**: B-Type and J-Type Instructions constaining decimal jump offset are not encoded in this implementation. Only labels are allowed as offsets.

5. **Limited Error Reporting**: The error reporting mechanism could be expanded to provide more detailed diagnostics and suggestions for correcting issues. Currently, it focuses on basic error detection without providing in-depth guidance.

9 Testing and Verification

To ensure the accuracy and functionality of the RISC-V simulator, various test cases were executed. For verification, I have attached a folder containing input test cases and their corresponding output results. These test cases cover a range of instructions and scenarios to validate the correctness of the assembler.

It is important to note that the attached folder does not include all the test cases that were run. Many additional test cases were also executed to thoroughly check the robustness of the assembler. These extra test cases were designed to cover edge cases and ensure the assembler handles various instructions and formats as expected.

10 Conclusion

The RISC-V assembly language simulator offers a comprehensive environment for executing and analyzing assembly code. It effectively manages registers, memory, and the call stack, allowing users to observe the internal state of the simulator during execution. By implementing functions for resetting registers and memory, as well as storing various data formats, the simulator demonstrates flexibility and robustness in handling different input scenarios.

While the current implementation covers essential functionalities, further enhancements could include support for more complex instructions and improved error handling. Additionally, integrating features like breakpoint management and advanced debugging tools would significantly enrich the user experience. Overall, the simulator serves as a valuable educational resource for understanding RISC-V architecture and assembly programming, with opportunities for continued development and refinement.