Lab 7 Report

Mohammed Abdur Rehman-AI23BTECH11015
Kodavatikanti Bhuvan Chandra-AI23BTCEH11013

November 12, 2024

Contents

1 Introduction

This code implements a RISC-V simulator, designed to parse and execute assembly instructions from an input file. Building on previous assignments, this simulator integrates a cache to mimic real-world memory handling. The goal is to enhance testing and validation of RISC-V assembly programs, particularly with the impact of cache on load and store instructions

2 Approach

In this lab assignment, the approach involved adapting prior work from "lab4" and adding cache simulation functionality, focusing on load and store instructions which interact with cache. We carefully managed cache data to ensure accurate simulation of cache operations. This includes ensuring cache attributes like size, associativity, and replacement policy are accurately reflected in the simulation to capture the effects on data storage and retrieval.

3 Implementation

The code implementation is modularized using several functions, which are classified into the following types:

3.1 Parse Instruction Input

This function reads an instruction from `input.s` as a string and splits it into smaller strings using commas, spaces, and braces. These smaller strings, which represent segments of the instruction, are then used in subsequent functions to process the instruction.

3.2 Validation Functions

- **validate R instruction**
- **validate I instruction**
- **validate I S instruction**
- **validate U instruction**
- **validate B instruction**
- **validate J instruction**

These functions validate the first part of the split strings by comparing them with hardcoded arrays of their respective operands and aliases. They return indices that are used to validate the instructions.

Note: The function `I S` is for the load and store instructions as they hold a similar instruction format, and the function `I` is for the remaining `I` format operands except the load instructions.

3.3 Execution Functions

- **Execute R instruction**
- **Execute I instruction**
- **Execute IS instruction**
- **Execute U instruction**
- **Execute B instruction**
- **Execute J instruction**

These functions process instructions based on their respective types and execute them according to the operand's behavior. Subsequently, the Program Counter (PC) value, memory, registers, and stack are updated.

3.4 Load File into Arrays

The `load_file_into_arrays` function reads data from a specified file and populates various memory arrays based on the data format.

Parameters

- `filename`: Name of the file to be read.
- `lines`: A 2D array to store each line of text from the file.
- `mem_values_dword`: Array for `.dword` values.
- `mem_values_word`: Array for `.word` values.
- `mem_values_half`: Array for `.half` values.
- `mem_values_byte`: Array for `.byte` values.

Functionality

- Opens the specified file for reading; returns -1 if the file cannot be opened.
- Initializes counters for different memory value types (`dword`, `word`, `half`, `byte`).
- Reads the file line by line, processing specific data types:
  - Ignores lines starting with `.data` or `.text`.
  - Parses and stores values for `.dword`, `.word`, `.half`, and `.byte` sections.
- Stores each non-data line into the `lines` array, respecting the `MAX_LINES` limit.
- Closes the file and returns the total number of lines read.

This function effectively organizes data from the file into designated arrays for further processing in a simulation or computation context.

3.5 Handle Instructions from Lines

The `handle_instructions_from_lines` function processes RISC-V assembly instructions stored in an array of lines. It executes these instructions based on their types and manages control flow using a stack.

Parameters:

- **lines:** A 2D array of strings containing assembly instructions.
- **total_lines:** Total number of lines in the `lines` array.
- **mem_values:** An array for memory values used in execution.
- **line_no:** The line number for reporting.
- **current_line_no:** Tracks the current line being executed.
- **stack1:** A pointer to a stack structure for managing control flow.

Functionality:

- Initializes variables and creates a label mapping for the instructions.
- Loops through each line of instructions:
  - Calculates the program counter (PC).

- Parses the current line to extract the instruction components.
- Validates the instruction type using several validation functions.
- Checks for breakpoints; if a breakpoint is hit, it stops execution.
- Executes the corresponding instruction based on its type:
  - **R-type:** Calls `execute_R_instruction`.
  - **I-type:** Calls `execute_I_instruction`.
  - **I/S-type:** Calls `execute_I_S_instruction` and handles `jalr`.
  - **U-type:** Calls `execute_U_instruction`.
  - **B-type:** Calls `execute_B_instruction` and manages control flow.
  - **J-type:** Calls `execute_J_instruction` and manages the stack for jumps.
- Logs execution details, including the instruction executed and the program counter.
- Increments the line number, handling jumps and branches as needed.
- Exits if the last line is reached.

This function facilitates the execution of a sequence of assembly instructions while managing control flow and breakpoints effectively, utilizing a stack for jumps and returns.

3.6 Register Values Initialization

The variable `register_values` is an array of 32 elements, each of type `int64_t`, initialized to zero. This array is designed to represent the values of 32 general-purpose registers in a RISC-V architecture. Each register is initialized to zero, ensuring a clean starting state for any subsequent operations or computations involving these registers. This initialization is crucial for simulating accurate register behavior during instruction execution in the RISC-V simulator.

These arrays are hardcoded and contain the 32 register names and their aliases. They are used for instruction validation.

3.7 The Parse Input Function:

`void parse_instruction_input(char input[], char b[], char c[], char d[], char e[])`
This function parses a string input representing an instruction and extracts up to four components into separate strings: `b`, `c`, `d`, and `e`. It initializes each output string to be empty, skips any leading whitespace, and checks if the input contains a label (indicated by a colon `:`). If so, it skips the label and processes the remainder of the string. The function parses each component based on delimiters such as spaces, commas, or parentheses. The components are copied into `b`, `c`, `d`, and `e`.

3.8 Memory

The `mem <address> <count>` command prints the specified memory address and the values stored at that address, along with the total number of values stored.

3.9 Step

The `step` function allows for the execution of one line of code at a time. It is designed to run the instruction without utilizing the `handle_instruction` function. The `step` function integrates seamlessly with both the `run` and `breakpoint` functionalities, allowing for interactive command execution via the terminal.

3.10 Breakpoint

The breakpoint functionality utilizes an array to store up to five line numbers where execution should halt. The `break <line>` command scans the specified line and stores it in the breakpoints array. Additionally, the `breakpoint found` function checks whether a breakpoint exists at the specified line and is invoked during instruction handling to determine if execution should pause.

3.11 Deleting Breakpoint

The `del break <line>` command scans for the specified line number in the breakpoints array. If the line number is found, it is removed from the array, causing the subsequent values to shift. This ensures that execution will no longer pause at that line during program execution.

3.12 Stack Implementation

The stack structure implements a basic stack data structure to manage a collection of values and their associated line numbers. It includes the following components:

3.12.1 Fields

- `int top index`: Tracks the index of the top element in the stack, initialized to -1 when the stack is empty.
- `char* value[maxstacksize]`: An array to store pointers to the values pushed onto the stack, with a maximum size defined by `maxstacksize` (0x50000).
- `int line num[maxstacksize]`: An array to store the corresponding line numbers for each value, enabling the tracking of where each value originated.

3.12.2 Functions

- `createemptystack()`: Allocates memory for a new stack instance and initializes the top index to -1, indicating the stack is empty.
- `push_to_stack(stack *s, char* value)`: Adds a new value to the top of the stack. It returns false if the stack is full (i.e., when the top index reaches the maximum size).
- `pop_from_stack(stack* s)`: Removes and returns the value at the top of the stack. If the stack is empty, it returns NULL.
- `top(stack* s)`: Returns the value at the top of the stack without removing it. Returns NULL if the stack is empty.

This implementation provides essential stack operations, enabling Last In First Out (LIFO) access to the stored values, which can be utilized for various purposes, such as managing control flow in assembly-like programs.

3.13 Label Mapping Function

The `create label map` function processes an array of assembly instruction lines to extract labels and create a mapping between those labels and their corresponding line numbers.

3.13.1 Parameters

- `char lines[MAX_LINES][MAX_LINE_LEN]`: An array of strings containing the lines of assembly code.
- `int total_lines`: The total number of lines in the input array.
- `char label_map[MAX_LINES][MAX_LINE_LEN]`: An array to store the extracted labels.
- `int line_map[MAX_LINES]`: An array to store the line numbers corresponding to each extracted label.

3.13.2 Functionality

The function iterates through each line, checking for the presence of a label (indicated by a colon `:`). If a label is found, it extracts the label name, stores it in the label map, and records the line number in the line map. The function returns the total number of labels extracted.

This functionality facilitates efficient label management in assembly-like languages, enabling further processing of instructions based on their associated labels.

3.14 Cache

The implementation comprises key functions that handle instruction parsing, validation, execution, and cache management:

- **execute_I_s_instruction**: Handles the execution of load and store instructions with direct interaction with cache. The function adjusts cache data accordingly, simulating real-world memory access delays and cache hits/misses.
- **Cache Implementation**
  - **initialize_cache()**: Initializes cache memory, setting up each cache block with default values.
  - **init_cache_log()**, **close_cache_log()**, **log_access()**: These functions manage cache logging, recording each access to an output file to track cache performance, hits, and misses.
  - **replace_block()**: Handles cache replacement policies, specifically FIFO, LRU, and Random, to simulate different memory access patterns and measure their impact on execution.

4 Error Handling

4.1 File Handling Errors

The function `load_file_into_arrays` checks if the file was successfully loaded and displays an error message if it fails:

```
if (line_count > 0) {
    ...
} else {
    printf("Error: Could not load file '%s'.\n", filename);
}
```

4.2 Breakpoint Handling Errors

When setting a breakpoint, the program checks if the user input is valid:

```
if (sscanf(command, "break %d", &line_no) == 1) {
    ...
} else {
    printf("Invalid command. Use: break <line>\n");
}
```

Similarly, when deleting a breakpoint, an error message is displayed if no breakpoint is found:

```
if (!found) {
    printf("Error: No breakpoint found at line %d\n", line);
}
```

4.3 End of Instructions Handling

In the `step` and `handle_instructions_from_lines` functions, there are checks to see if the program has reached the end of the instruction set:

```
if (line_number >= total_lines) {
    printf("No more instructions to execute.\n");
    return;
}
```

5 Assumptions and Limitations

1. It is assumed that there are no empty labels in the code.
2. Each instruction is expected to be on the same line as its corresponding label.
3. It is assumed that there are no blank lines present in the code.
4. The instructions are assumed to be syntactically correct and fall within the allowable bounds for sizes, immediates, and all related values. Hence, most of the error handling part from the assembler isn't continued into this segment.
5. If the code contains any of the directives `.dword`, `.word`, `.half`, or `.byte`, it is expected that all values for these directives will be present on the same line.
6. The function responsible for displaying the stack prints both the label name and the last executed line number.
7. The label `main:0` is pushed onto the stack before the first line following the `.text` directive.
8. The label `main` is popped from the stack after the last instruction has been executed.
9. During the stack implementation, line numbers are considered from the `.text` section. After the first line is executed, the command `show-stack` will display `main:1`, even if there are `.data`, `.text`, and `.dword` sections present.

5.1 Register Value Representation

The simulator prints register values in a 64-bit format, represented as hexadecimal numbers consisting of 16 digits. This format ensures that the full width of the register is visible, facilitating easy interpretation and debugging during the execution of RISC-V instructions.

5.2 Spacing and Formatting Rules

1. There is only one space between the instruction and the first operand.
2. There is only one space after a comma and before the next operand (e.g., `add x1, x2, x3`).
3. For labels, there is exactly one colon after the label name, followed by one space (e.g., `label:  add x1, x2, x3`).

5.3 Program Formatting

1. Each program line starts on the first character of that line (no leading spaces).
2. There is only one instruction per line.
3. There are no blank lines in the input file.

5.4 Instruction Set

The input file will contain only real RISC-V instructions (e.g., `add`, `lw`, `sw`, etc.) and will not contain any pseudo instructions (e.g., `li`, `mv`, etc.).

5.5 Immediate and Offset Values

1. Immediate or offset values provided with instructions will be in decimal format only. Negative values will be indicated with a `-` sign (e.g., `addi x1, x2, -10`).

6 Challenges Faced

During the development of the RISC-V + cache simulator, several challenges emerged:

- Simulating memory operations, particularly for load and store instructions, proved to be complex and time-consuming.
- An unknown error occurred while loading the file on Ubuntu systems (not on Windows), resulting in the overlapping of printed statements. This issue was traced back to the presence of invisible carriage return characters (`\r`) in the input file.
- Careful management of the program counter (PC) updates was required when executing function calls and jumps to ensure accurate simulation.
- The aforementioned non-visible carriage return character (`\r`) caused further complications during the execution of the `run` and `step` commands, leading to jumbled outputs. This issue was specific to Ubuntu systems, as Windows machines did not exhibit this behavior. The problem was eventually resolved.
- **Managing Multiple Output Files**: Structuring code to handle multiple output files efficiently was complex, especially as it required additional logic to maintain separate outputs for different simulation aspects.
- **Implementing Store Instructions**: The store instruction were tough to do in cache due to its interactions with both the simulator's memory management and cache implementation.

7 Conclusion

The simulator works out the cache memory accesses by keeping track of the trace for loading memory accesses, along with configuration cache and executing step by step. It supports the caches of types direct mapped, set associative, and fully associative with specific replacement policies like LRU or Random or FIFO. The simulator has already improved the process of cache analysis since it keeps a record of hits, misses, and related program counter values, thus it becomes a useful application of exploring cache behavior as well as optimizing cache performance. The cache simulator allows a user to specify cache parameters, such as cache size, block size, and associativity along with replacement policy and write policy. Then it processes memory accesses, simulates how the cache responds to each access, and provides statistics like hit rate, miss rate, Tag , Dirty/Clean states of blocks.The cache simulator is a valuable tool for improving cache performance. It shows cache behavior that is hard to see with manual testing, making it important for developing efficient computer systems.