Assembly Project Report

Adham Ali - 900223243
Ebram Thabet - 900214496
Khadeejah Iraky - 900222731

Submitted to Dr. Cherif Salama

CSCE2303, section 1
Computer Science and Engineering Department
The American University in Cairo

April 28, 2024

Design and Implementation

Our implementation consisted primarily of three main points: parsing the input, executing the instructions and producing the output.

(i) Printers and Initializers:

`printer_of_instructions`: Displays the sequence of assembly instructions that will be executed, aiding in tracing the execution flow step-by-step.

`printer_of_register_in_decimal`, `printer_of_register_in_hexa`, `printer_of_register_in_binary`: These functions print the current state of registers in various number formats (decimal, hexadecimal, and binary). They are useful for visualizing the effects of instructions on registers.

`removeLastNElements`: Manages vector data by removing a specified number of elements from the end. This function is useful for memory management operations within the simulated environment. .

(ii) Parsing the instructions:

The code defines two functions, `read_instructions` and `splitting`, aimed at parsing instructions from a text file into a two-dimensional vector of strings. The `read_instructions` function accepts a string `namefile` representing the file name containing instructions. It returns a 2D vector `sequential_instructions`, where each row represents a line of instructions from the file, and each column represents an individual instruction or label within that line. The `splitting` function splits a string representing a line of instructions into individual instructions or labels. It handles cases where multiple instructions or labels are separated by spaces or commas. Upon encountering a colon (':'), it identifies a label and appends the accumulated string followed by a colon to the `concatonate` vector. The `splitting` function handles spaces and commas between instructions, ensuring each instruction is correctly isolated. After processing each character, it returns the `concatonate` vector containing the individual instructions or labels. The `registers_handling` function is presumed to handle register aliases present in the instructions.

(iii) Handling Some Corner Cases:

`registers_handling`: Converts textual register identifiers into corresponding numeric identifiers, facilitating internal processing.

`getOffset`, `getContentInsideBrackets`: Extract specific parts from an instruction string, such as memory access offsets and register names within brackets, which are crucial for memory operations.

`Two_s_Complement`: Generates the two's complement of a binary string, a necessary operation for handling negative numbers in binary arithmetic.

`replace_ra_with_x1`, `replaceWords`: Adjust instruction strings to ensure consistency in register naming (e.g., replacing "ra" with "x1") or to correct specific keywords in the instruction set.

(iv) Functions:

"Functions" serves as the core execution engine for the simulated RISC-V assembly environment. It comprises a series of functions, each implementing a specific RISC-V instruction such as arithmetic operations, memory access, and control flow. These functions interact with an array of registers and a memory vector to simulate the effects of each instruction, modifying the program counter as necessary to dictate the sequence of execution. This section is critical for translating assembly language instructions into actionable operations that alter the state of the simulated machine, ensuring accurate emulation of the RISC-V processor's behaviour.

(v) Output Presentation:

Following the execution of each instruction, the program counter (PC) is printed, along with the names and contents of the 32 registers displayed in Decimal, Binary, and Hexadecimal formats.

Bonus Features

1. Outputting all values in decimal, binary, and hexadecimal (instead of just decimal which is assumed to be the default).
2. Add support for integer multiplication and division to effectively support the full RV32IM instruction set.
3. Including a larger set of test programs (at least 6 meaningful programs) and their equivalent C programs.
4. Including 5 psuedo-instructions in RISC-V.

Design Decisions and Assumptions

(i) Register Implementation

The simulation uses an array of 32 registers, all initialized to zero at the start. Values within these registers are altered according to executed instructions. During these operations, the x0 register is protected against any written actions to ensure it remains at zero, safeguarding its designated function and preventing unintended program behaviour changes.

(ii) Memory Architecture

In our design, the memory (stack) is designed as a vector¡int¿ Memory. How can you modify the memory? By the instruction of addi, we can modify the length of the memory. Ex: addi sp, sp, -16, this means that the memory size will be $16/4 = 4$. On the other hand, the instruction addi sp, sp, +ve, this instruction decreases the size of the memory.

(iii) Code Structure

Our code architecture benefits from a modular design to efficiently manage and organize the extensive number of code lines. By encapsulating each instruction into a distinct function, such as `void add(string rd, string rs_1, string rs_2, vector<int> &registers, int &program_counter)`, we significantly simplify the program's complexity. This method enhances the modularity and maintainability of the codebase.

(iv) Assumptions

1. The instructions must be written in lowercase, and this is the default of the RISC-V instructions. (the same for the labels)
2. Our program can not read comments.
3. We assume that the instruction includes space between every register / immediate or instruction name.

Known Bugs and Issues in the Simulator

1. **Handling of Label Instructions in Control Flow:** Branch instructions like `beq` and `bne` manipulate the instruction sequence based on labels without proper checks. This can cause infinite loops or undefined behavior if the label does not exist or the condition remains true indefinitely.
2. **No Bounds Checking on Memory Accesses:** The `lw` (load word) and `sw` (store word) functions to access the memory array using indices derived from immediate values without ensuring these indices are within the actual bounds of the memory vector, risking out-of-bounds access.
3. **Use of `stoi` Without Exception Handling:** The simulator uses `stoi` for string-to-integer conversions extensively without handling exceptions that `stoi` might throw. This can lead to the program terminating unexpectedly due to invalid numeric inputs.
4. **Potential Integer Overflow in Arithmetic Instructions:** Arithmetic operations such as `add` and `sub` are performed without checking for integer overflow or underflow, which could lead to incorrect results, especially when dealing with large or negative numbers.

User Guide

The project is written in C++, allowing you to compile it using any C++ compiler.

1. You must write the RISC-V instructions in the file, that the program will tell you to write in. You also have the option of linking a memory file with pre-determined memories for the program access.
2. You have only 40 instructions to use.
3. You can write the registers in the default form, or you can use these registers(sp, gp, fp, a0-a7, s0-s11, t0-t6)
4. You have 2 options (First: Representing the output by the 3 number systems implemented.) (Second: Representing the output only by 1 number system; decimal or hexadecimal or binary)

List of Programs

1. Fibonacci: This program is designed to calculate and find the nth element in the Fibonacci series, given the value of n.
2. Sum from 1 to n: This program is designed to calculate sum of all the numbers from 1 to the given number n.
3. Multiplying by 5: This program is designed to use the sll command in multiplying the given number n by 5.
4. Difference of two squares: This program is specifically developed to find the difference between the squares of two given numbers x and y.
5. Two to the power n: This program is given n, and calculates the value of two to the power n.
6. Factorial: This program calculates $n!$ given n.