b

School of Sciences and Engineering

Spring 2022

CSCE2303-02 – Computer Organization and Assembly Language Programming

Assembly Simulator

Submitted to:

Dr. Nourhan Zayed

Dr. Noha Aboelfadl

Submitted by:

Ahmed Fathy - 900192723

Mohamed Ali - 900192994

Mona Kamel - 900191833

20/04/2022

We have implemented the required simulator using C++ code. We searched for the 37 different RISC-V instructions and their format and implementation. A modular programming approach was used to implement the project as we implemented a function for every command that gets called whenever this command is read while execution. Moreover, we used the suitable data structures, maps, to implement different functionalities in the simulator. For example, only used registers were initialized in a map that maps the name of a register to its content. A similar approach was used in the memory initialization where only the used data addresses where initialized and stored their corresponding values in them. Finally, the final map was a map for the commands of a program where each command was mapped to a specific order in the program to facilitate the process of branching and jumping. These three maps were declared as global variables to be able to use them freely instead of passing them each time they are called to decrease the coding complexity. Also a global Boolean variable was declared to identify whether a program is halted or not as well as the program counter to be able to monitor and edit it freely.

**Program Execution Steps:**

The simulator runs through a series of steps. First, the program reads the program counter and memory initialization file. It initializes both variables accordingly. Second, the simulator reads the commands file and store them sequentially in a map in which they are mapped to their ascending order according to the program counter. Third, the simulator iterates over the commands map. Note: the iteration over the commands map is not done sequentially, however, the iteration is done according to the specific commands that is mapped to the program counter that has the turn to be executed. Commands are not executed sequentially as the branching and jumping needs to skip some commands. Each command was written an if-else statement in which the its function is called and its parameters are parsed accordingly. We have divided the parsing functions into two formats where one is responsible for parsing of commands that take two source registers, like ADD and SLL, or commands that take one source register and one immediate value, like ADDI or SRAI. The other parsing function is responsible for parsing commands that take offset values, such as LW or SW. Upon the calling of each command, the program counter before and after the calling is displayed as well as the content of the destination register. Finally, the content of each memory address is edited and the memory file is edited to hold the final values after the execution of the whole program.

**Bonuses:**

We have opted to outputting the content of the registers in three forms, decimal, binary, and hexadecimal.

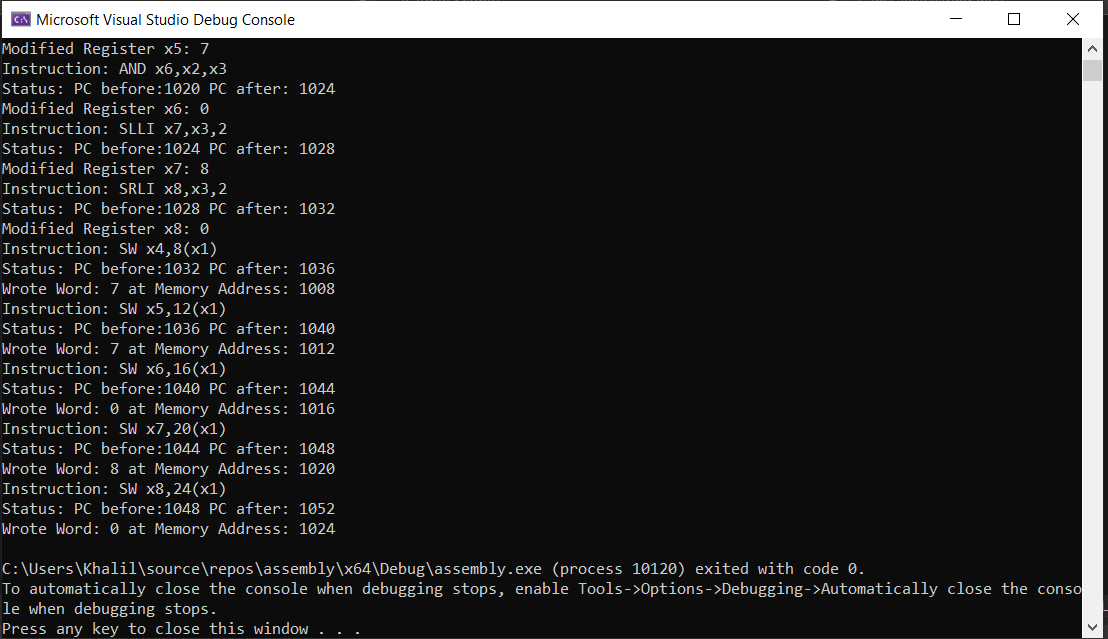
**Design Decision and Assumptions:**

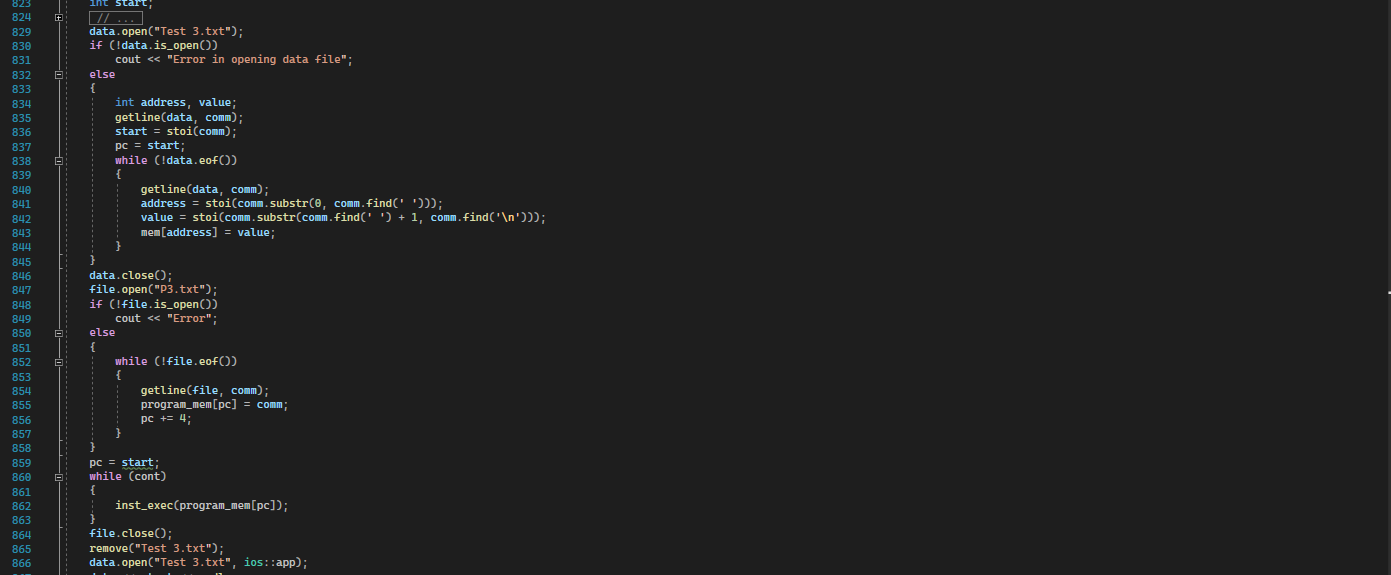
We decided to design the jump and branching commands to deal with the program counter instead of labels, which is accepted in RISC-V commands. Moreover, to further stick to the RISC 32 ISA, we decided to increment the program counter and the memory addresses by 4 each times as this means 4 bytes or 32 bits which the size specified by the ISA for the registers and memory locations.

**Bugs:**

The simulator was tested multiple times and was fixed to solve all the errors found and there are no remaining bugs.

**Guide to Use the simulator:**

In order to run the program on a specific test case the user must edit the name of the two text files entered to the simulator which are shown in the screenshots below:



The program then runs and shows the contents of registers are displayed on the console.

**Testing Programs:**

1. P1 and Test 1 are the first test case and they are used to mimic the behavior of a C-code While that iterates 10 times and increments the value of one of the values until it is equal to the second value. It is used to test loading, storing, branching, and adding commands.
2. P2 and Test 2 are the second test case and they are used to identify whether two numbers are divisible by each other or not. First it identifies which one of them is bigger and decrements it with the value of the smaller value and if the remainder is zero then they are divisible and a flag is set to 1. Otherwise, they are not divisible and the flag is set to 0. It is used to test loading, storing, branching, setting, and subtracting.
3. P3 and Test 3 are the third test case and they are used to test logical and Boolean functions like loading, storing, anding, oring, xoring, and shifting left and right.