RISC-V Simulator

Ramy Shehata-900222862

Omar Ganna-900222646

Mohamed Khaled-900225303

Department of Computer Science and Engineering

The American University in Cairo

Spring 2024

Table of Contents

1- Introduction

This report will discuss the implementation of a RISC-V simulator on C++ and aims to emulate the execution of RISC-V assembly code. From the registers used in the program to the data in the memory, our simulator tries to emulate how a RISC-V ISA executes the assembly code. The report will also talk about the design choices we took while designing such an interesting concept such as how we stop the code and how everything is allocated in terms of the registers and the memory, and how we handle the stack pointer. It will also include step by step guidelines on how to run the simulator, what is the needed code and its syntax for the simulator to run properly. Lastly, we will include around 6 sample programs we used to showcase how our simulator works and what is the output.

2- Simulator Implementation and Design

We will first talk about the implementations we used for the simulator and then dive into the design choices we have for our simulator.

a- The Implementation and Code

Firstly, we used C++ as our main programming language due to our experience in using it throughout most of our coding project. We are also most familiar with this language and so we opted for that option. However, we used Python to implement a small GUI program in order to run our C++ file. So, it is not entirely based on C++, but the main logic and heart of our simulator is built on C++.

Secondly, in order to classify everything, we used a class called RISCV_Instructions, which contains all the data needed to run the simulator. It uses a variety of data structures, mainly maps for storing labels, accessing memory, and dealing with registers. For example, we used a map<unsigned int, int> memory for the memory where unsigned int is the value of the address, and the int is the value that will be stored inside that memory location. We also used a struct called instructions to put the data we parse from the input assembly code into rs1, rs2, immediate and its program counter so we can know where that instruction was and the registers it has.

Thirdly, for our program flow, when the program is executed, it asks the user for the location of the data, the code, and the initial program counter. After that, it parses the instructions, extracting essential information such as the instruction type and its operands that are stored in the vector of the struct instructions. Additionally, the program keeps track of the address of each instruction. Upon parsing the assembly code, the program proceeds to open the data file and initializes memory using the contents of this file. Memory initialization is managed using map. Subsequently, the C++ code begins executing the instructions. The first instruction executed is the initial instruction in the main block. The program counter value is adjusted after each instruction execution. This process continues until all instructions have been executed. It's important to note that after executing each instruction, the simulator outputs the contents of the registers and memory in decimal, binary, and hexadecimal formats. Before the code runs, we ensure that the registers and memory are initialized with zero and it only outputs the registers and memory locations that either have a value or has undergone a change in the code. Otherwise, the non-output registers/memory is set to 0.

Lastly, we implemented 3 bonus features for our program. We did the GUI as a parsing tool and outputting the simulation. We did the output for both the registers and memory using not only decimal, but only hexadecimal and binary code. And the last bonus feature we implemented was the use of 6 sample test programs to ensure the robustness of our simulator.

b-Design of Our Simulator

Throughout the process of doing the simulator, we employed many different designs that may differ from other simulators out there. The first design choice is that we ask the user for the initial address of the instruction to ensure that the code knows the necessary information to do jumping instructions.

The second design choice is the use of ECALL, EBREAK and FENCE as halting instructions to ensure the code doesn't go through infinite loops and that the code actually stops since we are not jumping very far in the code unlike a real RISC-V ISA.

The third design choice is the use of data file, where the user must specify what the memory location of the data he is adding onto the memory and the value in decimal.

The fourth design choice is how the user must do specific things in the code for it to run properly. For example, when writing the code, the user must ensure that each instruction is written as specified here, with the spacing and commas as seen. Otherwise, the code will not be parsed into the simulator correctly.

- R-Format→ add rd, rs1, rs2
- I-Format→ addi rd, rs1, imm
- Load instruction → lw rd, offset(rs1)
- SB-format→ beq rs1, rs2, label
- S-Format→ sw rs2, offset(rs1)
- JAL instruction → jal rd, label
- JALR instruction → jalr rd, offset(rs1)
- U-Format → lui rd, imm

The fifth design choice is that our code can take the normal x base registers and named registers such as t0, s0, a3, sp, ra and so on. It also ignores capitalization.

The sixth design choice we took was the use of labels, for the code to run properly, the code cannot be written beside a label name. For example, the simulator cannot take the instruction written like this → label: add x10, x11, x12. It must take the instruction for the label as

Label:

add x10, x11, x12

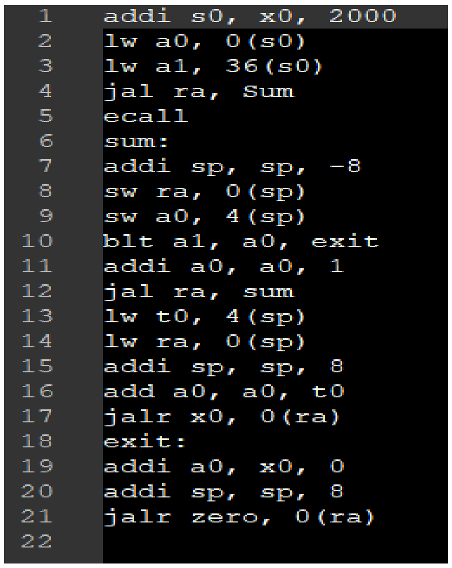
This is to ensure that our code can detect labels correctly and parse it correctly. If the instruction was beside the label, it would have caused many problems in implementation.

The last design choice is the regarding the stack pointer where the stack is managed within the same memory structure used for other data. To ensure ample space and prevent overwriting other memory contents, the stack pointer is assigned a significantly large address.

3- Simulator Usage Guide

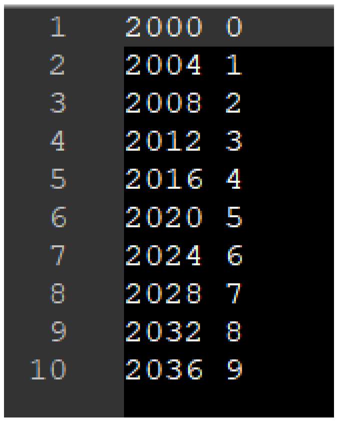
First, to run the simulator, you must ensure the code follows the design choices for the code and data file explained in part 2.b. Attached is a sample code of one of our testcases that showcases how the format is written.

Assembly code:



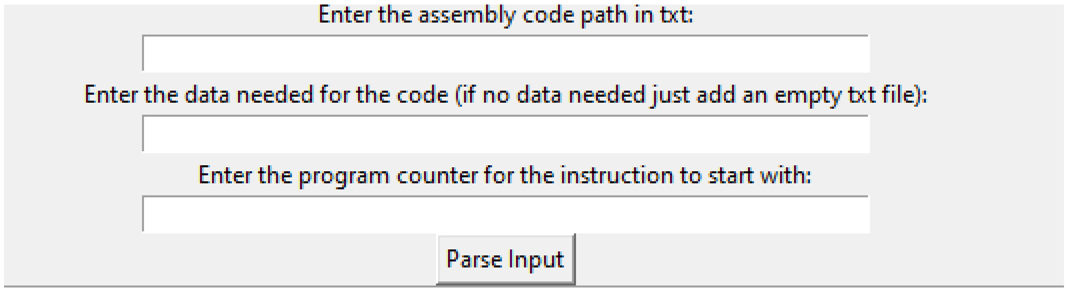
```
addi s0, x0, 2000
lw a0, 0(s0)
lw a1, 36(s0)
jal ra, Sum
ecall
sum:
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
blt a1, a0, exit
addi a0, a0, 1
jal ra, sum
lw t0, 4(sp)
lw ra, 0(sp)
addi sp, sp, 8
add a0, a0, t0
jalr x0, 0(ra)
exit:
addi a0, x0, 0
addi sp, sp, 8
jalr zero, 0(ra)
```

Data:



```
2000 0
2004 1
2008 2
2012 3
2016 4
2020 5
2024 6
2028 7
2032 8
2036 9
```

After that, you run the program "gui.py", where it will ask you for the assembly code in txt, the data in txt and the program counter as seen in the screenshot below.



Enter the assembly code path in txt:

Enter the data needed for the code (if no data needed just add an empty txt file):

Enter the program counter for the instruction to start with:

After that, the user will put the path for the assembly code or the data, if you want to test the testcases just add "Testcases/sample.txt" and replace sample with the testcase of choice. Same for the data path as well → "Testcases/data.txt". However, if you don't need to add any data into the memory, just simply add an empty text. After that you can specify the program counter, our testcases were tested for program counters 1000, 10000, 100 and any number divisible by 4.

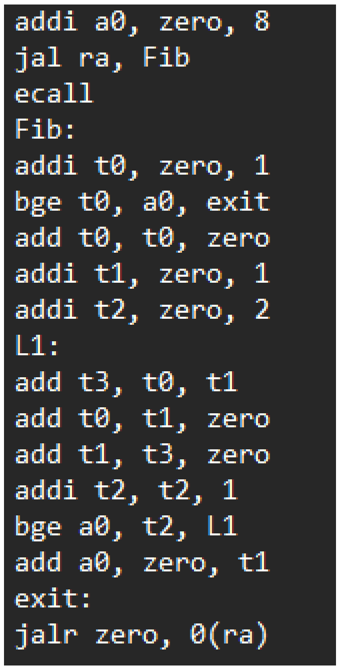
After adding the needed data, press the parse button and you should see on your screen an output. This output includes the program counter and each instruction of the user. After that, you can scroll down for a bit where you will find the changed registers of the entire program and the relevant memory location for the data if specified. Attached is a screenshot which specifies the following output.

4- Program testing

During our testing phase, we tested 6 different programs to ensure that our simulator can handle different codes. For each assembly code done, there will be its respective C++ file to show how we implemented it.

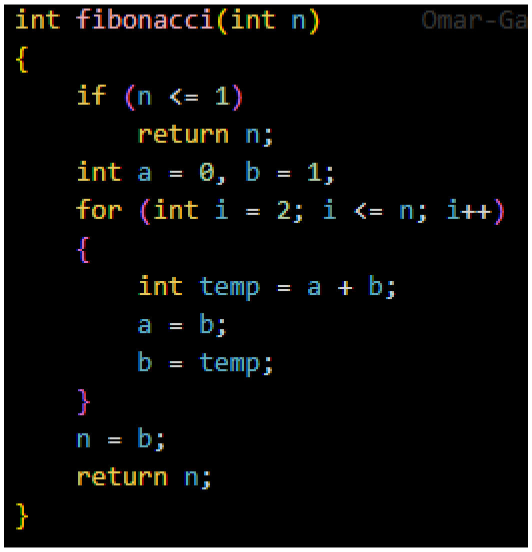
1- Iterative Fibonacci Sequence: This code tests loops and branching instructions

Assembly Code:



```
addi a0, zero, 8
jal ra, Fib
ecall
Fib:
addi t0, zero, 1
bge t0, a0, exit
add t0, t0, zero
addi t1, zero, 1
addi t2, zero, 2
L1:
add t3, t0, t1
add t0, t1, zero
add t1, t3, zero
addi t2, t2, 1
bge a0, t2, L1
add a0, zero, t1
exit:
jalr zero, 0(ra)
```

C++ Code:

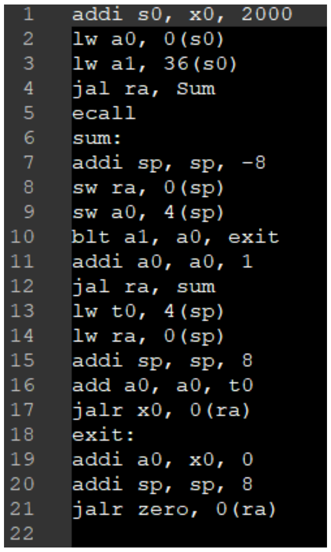


```
int fibonacci(int n)        Omar-Ga
{
    if (n <= 1)
        return n;
    int a = 0, b = 1;
    for (int i = 2; i <= n; i++)
    {
        int temp = a + b;
        a = b;
        b = temp;
    }
    n = b;
    return n;
}
```

Output File:

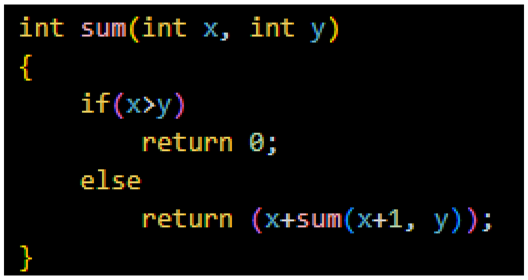
2- Recursive Array Sum: This tests recursion and jumping instructions.

Assembly Code:



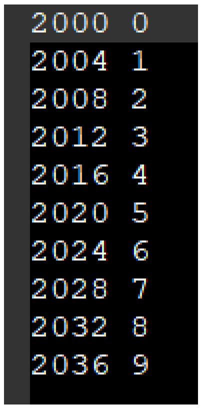
```
addi s0, x0, 2000
lw a0, 0(s0)
lw a1, 36(s0)
jal ra, Sum
ecall
sum:
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
blt a1, a0, exit
addi a0, a0, 1
jal ra, sum
lw t0, 4(sp)
lw ra, 0(sp)
addi sp, sp, 8
add a0, a0, t0
jalr x0, 0(ra)
exit:
addi a0, x0, 0
addi sp, sp, 8
jalr zero, 0(ra)
```

C++ Code:



```
int sum(int x, int y)
{
    if(x>y)
        return 0;
    else
        return (x+sum(x+1, y));
}
```

Data File:



```
2000 0
2004 1
2008 2
2012 3
2016 4
2020 5
2024 6
2028 7
2032 8
2036 9
```

Output File:

3-