RISC-V Simulator

Ramy Shehata-900222862

Omar Ganna-900222646

Mohamed Khaled-900225303

Spring 2024

Department of Computer Science and Engineering

The American University in Cairo

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 use of opcode file when parsing to ensure that when the code parses, it can parse the instructions easily depending on the opcode.

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.

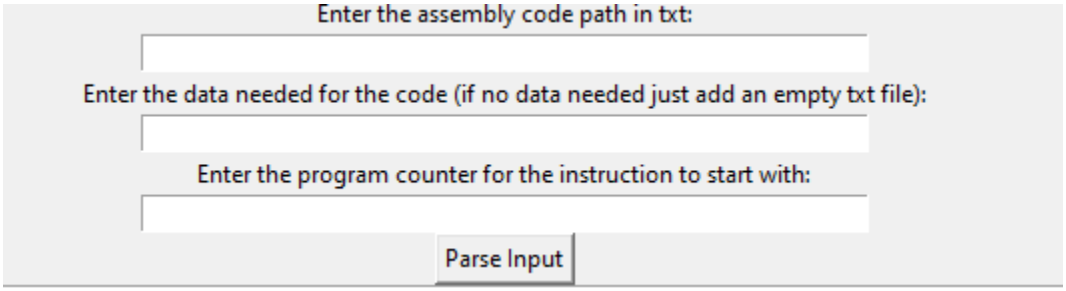
Assembly code:

```
1   addi s0, x0, 2000
2   lw a0, 0(s0)
3   lw a1, 36(s0)
4   jal ra, Sum
5   ecall
6   sum:
7   addi sp, sp, -8
8   sw ra, 0(sp)
9   sw a0, 4(sp)
10  blt a1, a0, exit
11  addi a0, a0, 1
12  jal ra, sum
13  lw t0, 4(sp)
14  lw ra, 0(sp)
15  addi sp, sp, 8
16  add a0, a0, t0
17  jalr x0, 0(ra)
18  exit:
19  addi a0, x0, 0
20  addi sp, sp, 8
21  jalr zero, 0(ra)
22
```

Data:

```
1   2000 0
2   2004 1
3   2008 2
4   2012 3
5   2016 4
6   2020 5
7   2024 6
8   2028 7
9   2032 8
10  2036 9
```

After that, make sure to run the RISCV_Instructions at least once in order to compile an exe file. 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.



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/sample.txt” and replace sample with the testcase of choice. Same for the data path as well → “../Testcases/sample/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.

4- Program testing

During our testing phase, we tested 6 different programs to ensure that our simulator can handle different codes. For some 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)
{
    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:

```
Instruction: add a0, zero, t1
Instruction Name: add Program Counter: 10052
Instruction: exit:
Instruction: jalr zero, 0(ra)
Instruction Name: jalr Program Counter: 10056
Register Decimal Hexadecimal Binary
------------------------------------------------------------------
x0          |    0        |0              | 00000000000000000000000000000000
x1          |    10008    |2718           | 00000000000000000010011100011000
x10         |    34       |22             | 00000000000000000000000000100010
x28         |    34       |22             | 00000000000000000000000000100010
x5          |    21       |15             | 00000000000000000000000000010101
x6          |    34       |22             | 00000000000000000000000000100010
x7          |    9        |9              | 00000000000000000000000000001001
```

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

Assembly Code:

```
1   addi s0, x0, 2000
2   lw a0, 0(s0)
3   lw a1, 36(s0)
4   jal ra, Sum
5   ecall
6   sum:
7   addi sp, sp, -8
8   sw ra, 0(sp)
9   sw a0, 4(sp)
10  blt a1, a0, exit
11  addi a0, a0, 1
12  jal ra, sum
13  lw t0, 4(sp)
14  lw ra, 0(sp)
15  addi sp, sp, 8
16  add a0, a0, t0
17  jalr x0, 0(ra)
18  exit:
19  addi a0, x0, 0
20  addi sp, sp, 8
21  jalr zero, 0(ra)
22
```

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:

```
Instruction: jalr zero, 0(ra)
Instruction Name: jalr Program Counter: 10072
Register Decimal Hexadecimal Binary
------------------------------------------------------------------
x0          |    0        |0              | 00000000000000000000000000000000
x1          |    10016    |2720           | 00000000000000000010011100100000
x10         |    45       |2d             | 00000000000000000000000000101101
x11         |    9        |9              | 00000000000000000000000000001001
x2          |    0        |0              | 00000000000000000000000000000000
x5          |    0        |0              | 00000000000000000000000000000000
x8          |    2000     |7d0            | 00000000000000000000011111010000
------------------------------------------------------------------
Memory Address Decimal Hexadecimal Binary
------------------------------------------------------------------
2000        |    0        |0              | 00000000000000000000000000000000
2004        |    1        |1              | 00000000000000000000000000000001
2008        |    2        |2              | 00000000000000000000000000000010
2012        |    3        |3              | 00000000000000000000000000000011
2016        |    4        |4              | 00000000000000000000000000000100
2020        |    5        |5              | 00000000000000000000000000000101
2024        |    6        |6              | 00000000000000000000000000000110
2028        |    7        |7              | 00000000000000000000000000000111
2032        |    8        |8              | 00000000000000000000000000001000
2036        |    9        |9              | 00000000000000000000000000001001
```

3- Array Sum iteratively: This tests the iteration and jumping.

Assembly Code:

```
addi s0, zero, 100
addi s1, zero, 5
Sum:
addi s2, zero, 0
addi t0, zero, 0
Loop:
bge t0, s1, break
lw t2, 0(s0)
addi s0, s0, 4
add s2, s2, t2
addi t0, t0, 1
jal ra, Loop
break:
add x1, zero, s2
ecall
```

C++ code:

```
#include <stdio.h>

int sum(int a[], int l) {
    int sum = 0;
    int i = 0;
    while (i < l)
    {
        sum += a[i];
        i = i + 1;
    }
    return sum;
}

int main()
{
    int a[5] = { 6,8,9,3,2 };
    sum(a, 5);

    return 0;
}
```

Data File:

```
100 6
104 8
108 9
112 3
116 2
```

Output:

```
Register Decimal Hexadecimal Binary
-----------------------------------------------------------------
x0       |    0       |0              | 00000000000000000000000000000000
x1       |    28      |1c             | 00000000000000000000000000011100
x18      |    28      |1c             | 00000000000000000000000000011100
x5       |    5       |5              | 00000000000000000000000000000101
x7       |    2       |2              | 00000000000000000000000000000010
x8       |    120     |78             | 00000000000000000000000001111000
x9       |    5       |5              | 00000000000000000000000000000101
-----------------------------------------------------------------
Memory Address Decimal Hexadecimal Binary
-----------------------------------------------------------------
100      |    6       |6              | 00000000000000000000000000000110
104      |    8       |8              | 00000000000000000000000000001000
108      |    9       |9              | 00000000000000000000000000001001
112      |    3       |3              | 00000000000000000000000000000011
116      |    2       |2              | 00000000000000000000000000000010
```

4- Bitwise Testing: this tests the majority of the bitwise operators and ensures they work

Assembly Code:

```
addi s0, zero, 100
addi s1, zero, 500
AND t0, s0, s1
OR t1, s0, s1
XOR t2, s0, s1
andi t3, s0, 2000
ori t4, s1, 1500
xori t5, s0, 365
```

Output File:

```
Instruction: addi s0, zero, 100
Instruction Name: addi Program Counter: 10000
Instruction: addi s1, zero, 500
Instruction Name: addi Program Counter: 10004
Instruction: and t0, s0, s1
Instruction Name: and Program Counter: 10008
Instruction: or t1, s0, s1
Instruction Name: or Program Counter: 10012
Instruction: xor t2, s0, s1
Instruction Name: xor Program Counter: 10016
Instruction: andi t3, s0, 2000
Instruction Name: andi Program Counter: 10020
Instruction: ori t4, s1, 1500
Instruction Name: ori Program Counter: 10024
Instruction: xori t5, s0, 365
Instruction Name: xori Program Counter: 10028
Register Decimal Hexadecimal Binary
-----------------------------------------------------------------------
x0          |    0       |0              | 00000000000000000000000000000000
x28         |    64      |40             | 00000000000000000000000001000000
x29         |    1532    |5fc            | 00000000000000000000010111111100
x30         |    265     |109            | 00000000000000000000000100001001
x5          |    100     |64             | 00000000000000000000000001100100
x6          |    500     |1f4            | 00000000000000000000000111110100
x7          |    400     |190            | 00000000000000000000000110010000
x8          |    100     |64             | 00000000000000000000000001100100
x9          |    500     |1f4            | 00000000000000000000000111110100
-----------------------------------------------------------------------
Memory Address Decimal Hexadecimal Binary
-----------------------------------------------------------------------
```

5- Shifting Test: This test code tests some of the shifitng instructions and slt:

Assembly Code:

```
addi t0, x0, 20
addi t1, x0, 10
addi t2, x0, 80
addi t3, t3, 2
slt t4, t0, t1
srl t5, t2, t3
slli t6, t2, 1
sra s0, t2, t1
```

Output:

```
Instruction: addi t0, x0, 20
Instruction Name: addi Program Counter: 10000
Instruction: addi t1, x0, 10
Instruction Name: addi Program Counter: 10004
Instruction: addi t2, x0, 80
Instruction Name: addi Program Counter: 10008
Instruction: addi t3, t3, 2
Instruction Name: addi Program Counter: 10012
Instruction: slt t4, t0, t1
Instruction Name: slt Program Counter: 10016
Instruction: srl t5, t2, t3
Instruction Name: srl Program Counter: 10020
Instruction: slli t6, t2, 1
Instruction Name: slli Program Counter: 10024
Instruction: sra s0, t2, t1
Instruction Name: sra Program Counter: 10028
Register Decimal Hexadecimal Binary
-----------------------------------------------------------------
x0          |    0        |0              | 00000000000000000000000000000000
x28         |    2        |2              | 00000000000000000000000000000010
x29         |    0        |0              | 00000000000000000000000000000000
x30         |    20       |14             | 00000000000000000000000000010100
x31         |    160      |a0             | 00000000000000000000000010100000
x5          |    20       |14             | 00000000000000000000000000010100
x6          |    10       |a              | 00000000000000000000000000001010
x7          |    80       |50             | 00000000000000000000000001010000
x8          |    81920    |14000          | 00000000000000010100000000000000
-----------------------------------------------------------------
Memory Address Decimal Hexadecimal Binary
-----------------------------------------------------------------
```

6- Max in Array: This tests plethora of functions and tests arrays and memory

Assembly Code:

```
addi a0, x0, 32
addi a1, x0, 5
max:
addi t0, zero, 0
lw t1, 0(a0)
ForLoop:
slli t2, t0, 2
add t2, t2, a0
lw t2, 0(t2)
bge t1, t2, cont
addi t1, t2, 0
cont:
addi t0, t0, 1
blt t0, a1, ForLoop
addi x1, t1, 0
```

C++ code

```
#include <stdio.h>

int maximum(int a[], int l)
{
    int max = a[0];
    for (int i = 0; i < l; i++)
    {
        if (a[i] > max)
        {
            max = a[i];
        }
    }
    return max;
}
```

Data File

```
32 5
36 3
40 -1
44 8
48 2
```

Output File:

```
Instruction: addi t0, zero, 0
Instruction Name: addi Program Counter: 10008
Instruction: lw t1, 0(a0)
Instruction Name: lw Program Counter: 10012
Instruction: forloop:
Instruction: slli t2, t0, 2
Instruction Name: slli Program Counter: 10016
Instruction: add t2, t2, a0
Instruction Name: add Program Counter: 10020
Instruction: lw t2, 0(t2)
Instruction Name: lw Program Counter: 10024
Instruction: bge t1, t2, cont
Instruction Name: bge Program Counter: 10028
Instruction: addi t1, t2, 0
Instruction Name: addi Program Counter: 10032
Instruction: cont:
Instruction: addi t0, t0, 1
Instruction Name: addi Program Counter: 10036
Instruction: blt t0, a1, forloop
Instruction Name: blt Program Counter: 10040
Instruction: addi x1, t1, 0
Instruction Name: addi Program Counter: 10044
Register Decimal Hexadecimal Binary
-----------------------------------------------------------------
x0          |    0        |0              | 00000000000000000000000000000000
x1          |    8        |8              | 00000000000000000000000000001000
x10         |    32       |20             | 00000000000000000000000000100000
x11         |    5        |5              | 00000000000000000000000000000101
x5          |    5        |5              | 00000000000000000000000000000101
x6          |    8        |8              | 00000000000000000000000000001000
x7          |    2        |2              | 00000000000000000000000000000010
-----------------------------------------------------------------
Memory Address Decimal Hexadecimal Binary
-----------------------------------------------------------------
32          |    5        |5              | 00000000000000000000000000000101
36          |    3        |3              | 00000000000000000000000000000011
40          |    -1       |ffffffff       | 11111111111111111111111111111111
44          |    8        |8              | 00000000000000000000000000001000
48          |    2        |2              | 00000000000000000000000000000010
```

5- Known Bugs and Issues:

With the current testing we have done, we have tested almost all instructions but due to time constraints we have failed to test the lb and lhu instructions. They may have some issues with them. Other than that, there seems to be no known issues or bugs that we have encountered thus far. However, there is always room for improvement. We could have added more to the simulator such as support for RV32IM, the mul and div instructions.

Thank you for reading the report. If you have any questions feel free to ask us