



香港公開大學 科技學院
THE OPEN UNIVERSITY OF HONG KONG
SCHOOL OF SCIENCE AND TECHNOLOGY

ELEC S411F (2020/21)

Electronic and Computer Engineering Project

Final Report

RISC-V Simulator

Project number:	A03
Student name	Chan Man Hei
Supervisor:	Angus Wong
Submission Date:	13/6/2021

Declaration of Originality

I, [Chan Man Hei](#), declare that this report and the work reported herein was composed by and originated entirely from me. This report has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the reference section.

[\[8/6/2021\]](#)

Abstract

RISC-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. It was designed to support computer architecture research and education, but which they now hope will also become a standard free and open architecture for industry implementations. So, I am going to develop the simulator to simulate the instruction of RISC-V and provide a good performance and good visualization of the processor.

One of the main goals of this simulator was to be easily extensible and modifiable. To achieve this objective, the original design was very simple and clear, with the use of naive techniques and a source code designed for simplicity.

In this paper I am going to show how the simulator work and what the simulator can do. By adding all the instruction of RISC-V architecture and coding the function, when the simulator read the instruction it will execute the function and print the information.

TABLE OF CONTENTS

1.	Introduction.....	5
1.1	Project Objectives	6
1.2	Organization of the Report.....	7
2.	Background.....	8
2.1	RISC-V Architecture.....	8
2.2	Executable and Linkable Format (ELF)	10
3.	Design	12
3.1	Disassembler	13
3.2	Simulator.....	16
4.	Implementation	17
4.1	Read Elf	18
4.2	Get Data Type	18
4.3	Decode Type	20
4.4	Get Type Instruction.....	20
4.5	Handle Type	21
4.6	Dump To Json	22
4.7	The New Instruction	23
5.	Results and Evaluation.....	24
5.1	Execute the instruction.....	24
5.2	Dump to Json File	25
5.3	Edit Elf File.....	26
6.	Conclusion and Further Work	28
7.	References.....	29

1. Introduction

RISC-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, RISC-V ISA is available under an open source license, and there are no royalties to pay. RISC-V hardware is available from a number of firms, open source operating systems that support the instruction set are available, and certain prominent software toolchains support the instruction set.

RISC-V ISA include architecture-neutral design, load-store architecture, simpler bit patterns for CPU multiplexers, IEEE 754 floating point, and storing the most significant bit in the set position to speed up sign extension. The instruction set has a wide range of applications. The basic instruction set is a fixed-length 32-bit natural alignment instruction, however the ISA provides for variable-length extensions, allowing for any number of 16-bit packet lengths for each instruction. A subset supports small embedded systems, personal computers, supercomputers with vector processors, and warehouse-scale 19 inch rack-mounted parallel computers.

The aims of this project are developing a RISC-V simulator and add some new instruction. When the simulator is executing user can type the RISC-V instruction. Then, the program will start to simulate the instruction and print the program counter. Then, the simulator can be read the instruction of ELF file and run the instruction of RISC-V architecture and read the instruction of ELF file. Finally, the simulator can execute the newly developed instruction for special using. User can change the content of function easily.

1.1 Project Objectives

- 1) Develop a simulator that can encode and execute the RISC-V instruction:

To begin, gather all of the instructions in the RISC-V architecture and determine their kind. Also, be familiar with the function and properties of the instruction. Then, for each type of instruction, build a function. Finally, finish each instruction's substance.

It is critical for the simulator; if the simulator requires additional function, this section should be implemented first.

- 2) Read an ELF file and translating the binary code to instruction and execute it:

First, the simulator should read each line of the elf file format. Second, the function will capture each instruction's binary number. Third, it will use the final seven digits of the instruction to determine the type and retrieve the instruction. Finally, call the execute function to execute it.

The amount of instructions is usually rather considerable. The user always saves the instructions in a text file. As a result, a scanner for text files is required in the simulation.

3) Create a new instruction to let user have a flexible command set.

We'll need a new instruction to make the simulator extendable and modifiable. At the original type, the new instruction should be different. As a result, a new type should be created to handle the new instruction.

It should be simple to extend or alter the new instruction. As a result, it should be brief. I'm going to follow the original instructions and employ the procedure. It has the potential to improve code consistency.

1.2 Organization of the Report

The simulator's objectives are to encode and execute RISC-V instructions, read an ELF file, and generate new instructions. Learning the RISC-V architecture requires a long time in order to fulfill the goals.

The first challenge is to plan the simulator's architecture. The simulator should be able to receive and execute the four different types of instructions. It should provide a method to handle the type and retrieve the get type function instruction. Also, have a method to decode the instruction execution.

The second challenge is that there are numerous instructions, many of which I am unfamiliar with. It should do several tests to ensure that the instruction function is correct.

The third challenge is that I am not very familiar with the ELF file format. The ELF

file format is a text-based file format. An ELF file is divided into two sections: the ELF header and the file data. I should be able to convert the data to ELF format.

The paper is structured in the following sections: Section 2 depicts background of the simulator, Section 3 show the design of the simulator, Section 4 shows the implementation of simulator, Section 5 show the result and evaluation, Section 6 show the conclusion and further work.

2. Background

In this part, I am going to explain the background information of the simulator. I am going to show the concept of the RISC-V architecture and ELF file format.

2.1 RISC-V Architecture

A RISC-V ISA is made up of a basic integer ISA that must be included in any implementation, as well as optional extensions to the base ISA. Except for the lack of branch delay slots and support for optional variable-length instruction encodings, the base integer ISAs are very similar to those of early RISC processors.

A base is carefully limited to a minimum set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional privileged operations), and thus serves as a useful ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

RISC-V is a family of related ISAs, of which there are now four basic ISAs, despite the fact that it is more practical to refer to it as the RISC-V ISA. The width of the integer registers and the matching size of the address space, as well as the number of

integer registers, define each base integer instruction set.

The RISC-V architecture is modular, with optional basic components and optional extension pieces. Industry, the scientific community, and educational institutions collaborated to create the ISA foundation and its expansions. Instructions, control flow, registers, memory and addressing, logical operations, and auxiliary equipment are all specified in the base address. A simplified general-purpose computer with comprehensive software support, including a general-purpose compiler, may be built from a single basis. Standard extensions are intended to function with all standard bases and are not incompatible with one another.

There are four core instruction formats (R/I/S/U), as shown in Figure 2.1. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned.

To make decoding easier, the RISC-V ISA preserves the source (rs1 and rs2) and destination (rd) registers in the same place in all formats. Except for the 5-bit instant used in CSR instructions, immediate are usually sign-extended and crammed into the instruction's leftmost available bits to save hardware complexity. To accelerate sign-extension circuitry, the sign bit for all immediate is always in bit 31 of the instruction.

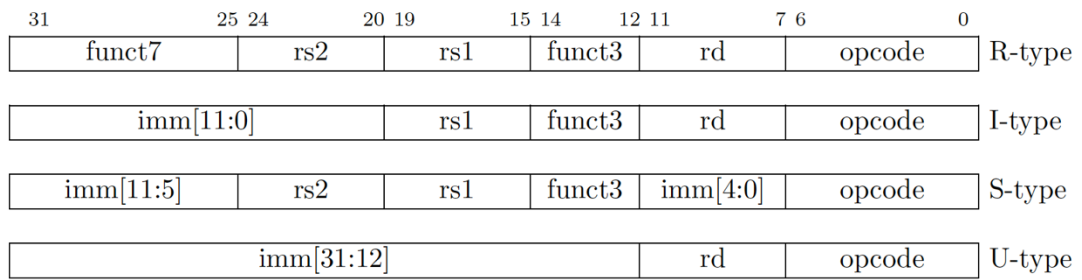


Figure 2.1

2.2 Executable and Linkable Format (ELF)

The ELF file is a common standard file format for executable files, object code, shared libraries, and core dumps. The design of ELF isn't restricted to a single processor, instruction set, or hardware architecture. It is flexible, extensible, and cross-platform. It supports different endiannesses and address sizes, so it does not exclude any particular or instruction set architecture.

An ELF file has two pieces, as you can see from the description above: an ELF header and file data. A program header table describing zero or more segments, a section header table describing zero or more sections, and data referred to by entries from the program header table and the section header table make up the file data section. Each segment includes information required for the file's run-time execution, whereas sections include critical data for linking and relocation. This is shown schematically in Figure 2.2.

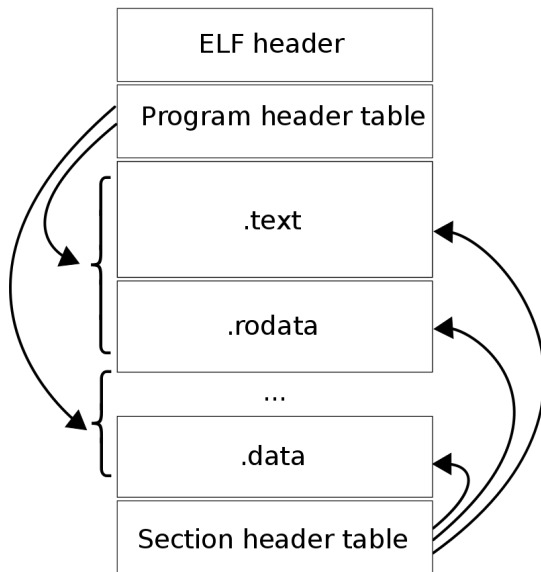


Figure 2.2

The ELF header is 32 bytes long and indicates the file's format. It begins with a four-byte sequence that begins with 0x7F and ends with 0x45, 0x4c, and 0x46, which correspond to the characters E, L, and F. The header also specifies if the file is in 32-bit or 64-bit format, whether it utilizes little or big endianness, the ELF version, and the operating system for which the file was produced so that it can communicate with the appropriate application binary interface (ABI) and CPU instruction set.

The program header shows the segments used at run-time and tell the system how to create a process image. The header from Listing 2 shows that the ELF file consists of 9 program headers that have a size of 56 bytes each, and the first header starts at byte 64.

The program header table of an executable or shared object file is an array of structures, each specifying a segment or other information needed by the system to prepare the program for execution. As described in 'Segment Contents,' an object file segment has one or more parts. Only executable and shared object files benefit from

program headers. The ELF header's `e_phentsize` and `e_phnum` members allow a file to choose its own program header size.

The section header table of an object file allows you to locate all of the file's sections. As stated below, the section header table is an array of `Elf32_Shdr` or `Elf64_Shdr` structures. The index of the section header table is a subscript into this array. The byte offset from the beginning of the file to the section header table is given by the `e_shoff` component of the ELF header. `e_shnum` usually indicates the number of items in the section header table. `e_hentsize` returns the size of each item in bytes.

3. Design

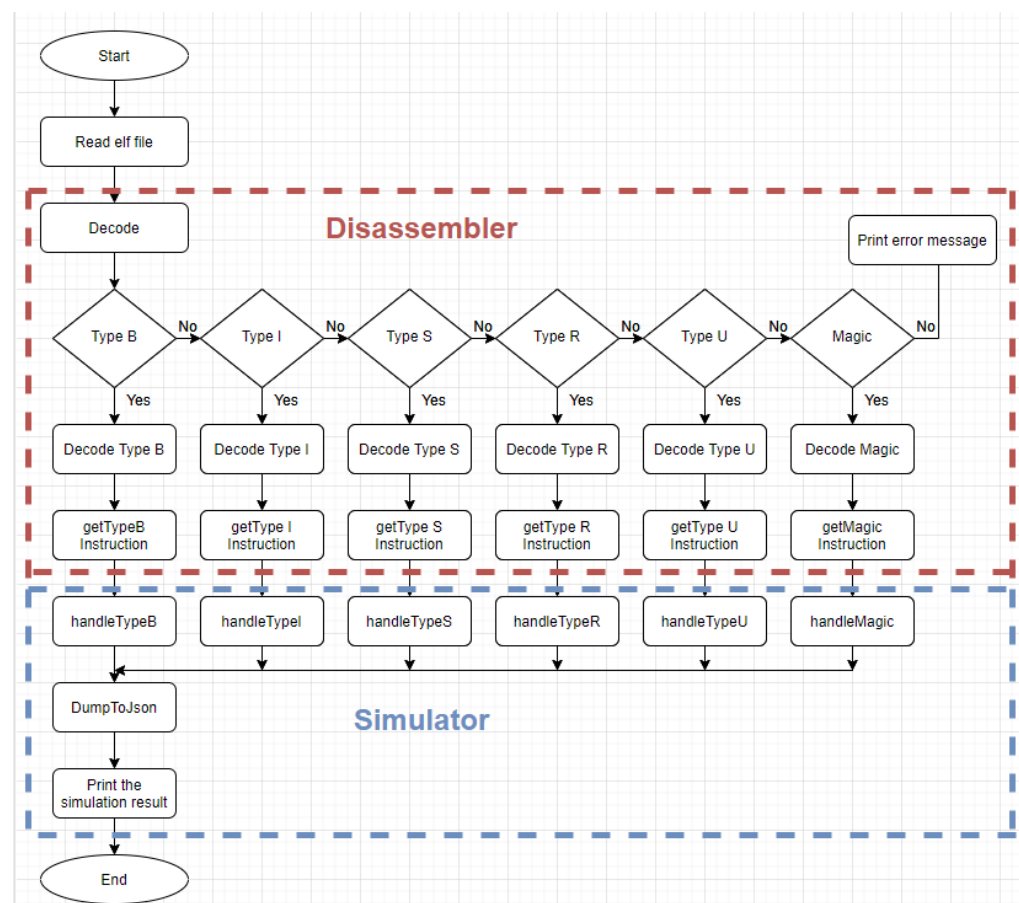


Figure 3.1

In this section, I am going to show the design of the simulator. The design of the simulator is divided by two part disassembler and simulator. The simulator is use to decode the instruction of disassembler. In Figure 3.1, it shows the design of simulator step by step. And I am going to show the detail design of them.

This chapter will introduce the preliminary design of the whole system. It will be divided into 7 sections. Section 3.1.1 will show the read elf function. Section 3.1.2 will show get data function. Section 3.1.3 will show decode type function. Section 3.1.4 will show get type instruction function. Section 3.2.1 will show the handle type function. Section 3.2.2 will show the dump to json function. At the end, the new instruction will be presented in Section 3.2.3.

3.1 Disassembler

3.1.1 Read elf

The first function is read elf, the read elf function is use to receive the data. Each of the 32 bits will be assign to one group and pass it to the get data function.

The main purpose of the function is use to read elf file. It is important to the simulator because of when we doing simulation, there will have a lot of instruction. So, it should be have a file to store them all. Then, the read elf function should be implementing.

When the simulator gets the elf file, it will enter to the get elf function. The function will scan it line by line. Each 32 bits will pass it to get data function.

3.1.2 Get data

The second function is get data. It is use to get the type of instruction. And pass the type and the 32bits data to decode type function.

The purpose of this function is use to find the type of function and pass to the decode type function. Because of the decode type function should be know the type of instruction first. So, this function should be implementing.

When the 32 bits data passing to get data function, the last 7 bits of data is the objcode. So, in these 7 bits we can find out the type of instruction. And pass it to decode type function.

3.1.3 Decode type

The third function is decoding type function. This function is use to decode the data by their type. And there are 6 types of decode type function (decodeTypeB, decodeTypeI, decodeTypeS, decodeTypeR, decodeTypeU, decodeMagic).

The purpose of this function is use to get the information of the data, this data is use to initial the direction of instruction. Because of this function is use to get the information of instruction. So, this function should be implementing.

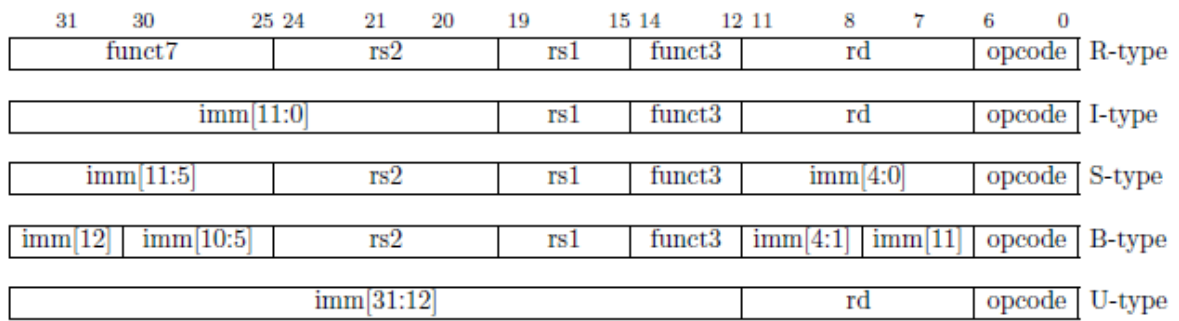


Figure 3.1.3

When the decode type function get the data, it will divide the data to different. In Figure3.1.3 we can see the distribution of the data. For type B, It will divide to funct3, r1, r2 and immediate value. For type I, It will divide to funct3, r1, rd and immediate value. For type S, It will divide to funct3, r1, r2 and immediate value. For type R, It will divide to funct7, r1, r2, funct3 and rd. For type U, It will divide to rd and immediate value. The funct3 or funct7 is meaning the function of the instruction and rd1, rd2, rd is meaning the register number.

3.1.4 Get type instruction

The fourth function is get type instruction function. This function is use to get the instruction by their code. And there are 6 types of get type instruction function (getTypeBInstruction, getTypeIInstruction, getTypeSInstruction, getTypeRInstruction, getTypeUInstruction, getMagicInstruction).

The purpose of get type instruction is use to get the instruction by their type. For the simulator, it is the important function. Because of get the instruction is the major function. So, this function should be implementing.

When the data information passes to this function, it will focus on the funct3. By checking funct3 value with their type, we can get the instruction. The `getTypeUInstruction` is special, It use the opcode to get the instruction.

3.2 Simulator

3.2.1 Handle Type

The fifth function is handle type function. This function is use to execute the instruction. And there are 6 types of handle type function (`handleTypeB`, `handleTypeI`, `handleTypeS`, `handleTypeR`, `handleTypeU`, `handleMagic`).

The purpose of get type instruction is use to execute the function of instruction. In this function it needs to specific the function of the instruction and presents it to the user. The reason of create 6 handle type function is it is easier to manage the instruction. Because of it is the most important function of the simulator. So, this function should be implementing.

When the instruction is passes to the handle type function, it will execute the function. For example the instruction `addi a0, a0, -4`, it will sum the register a0 value and -4 together and store back into register a0.

3.2.2 Dump to json

The sixth function is dump to json function. This function is used to store the value after execution and dump it to the json file. The value includes the operand, offset

and the offset after mapping.

The purpose of dump to json function is. If we don't have this function, the user can't get the information after execution. So, this function should be implementing.

When the handle type function is execute, the dump to json function will execute parallel. It will follow the handle type function when the execution of handle type function is done. Than it will call the dump to json function the write the result to the json file.

3.2.3 The New Instruction

Finally, the seventh is adding the new instruction. This function is use to create a new instruction for the RISC-V simulator. And it is dependence on the normal instruction.

The purpose of create the new instruction is let user have a more flexible command set. It can make the simulator more extendable and modifiable.

4. Implementation

In this part, I am going to show the detail of implement the simulator. I will show the method I use and the example of the simulator.

This chapter will introduce the Implementation of the whole system. It will be divided into 6 sections. Section 4.1 will show the read elf function. Section 4.2 will show get data function. Section 4.3 will show decode type function. Section 4.4 will show get

type instruction function. Section 4.5 will show the Handle Type function. Section 4.6 will show Dump to json function. At the end, the new instruction will be presented in Section 4.7.

4.1 Read Elf

The read elf function is designed to get the data of elf file. To handle the elf file format the company I work with created a library call “Quantr Dwarf Library”. The simulator uses this library to read the elf.

When the Simulator is executing, user can put their path to the simulator. Then the simulator will use the Quantr Dwarf Library to translate to the binary no.

```
File file = new File(TestFile1.class.getResource("riscv_simulator/data.elf").getPath());
```

```
Simulator.main(new String[]{"--elf", file.getAbsolutePath(), "-m", "1GB", "-s", "0x0000000",  
"-d", "dump.json", "--map=0x0-0x2ffff=0x8000000", "--init", "init.txt"});
```

Those code are telling the simulator that the path of the simulator and use what information to initialize the simulator. After initialize the simulator, each 32 bits will pass it to get data function to decode.

4.2 Get Data Type

The get data function is designed to decode the data of read elf function get. By the RISC-V architecture, we can know that the last 7 bits of 32 bits instruction data is their opcode. And this opcode can be translated to the type of instruction.

For getting the type of the instruction, I put the opcode case to the hashmap.

```
op32map2.put(0b1100011, "decodeTypeB"); // type B
op32map2.put(0b0000011, "decodeTypeI"); // type I
op32map2.put(0b0010011, "decodeTypeI"); // type I
op32map2.put(0b1110011, "decodeTypeI"); // ecall & ebreak
op32map2.put(0b0100011, "decodeTypeS"); // type S
op32map2.put(0b0110011, "decodeTypeR"); // type R
op32map2.put(0b0110111, "decodeTypeU"); // type U
op32map2.put(0b0010111, "decodeTypeU"); // type U
op32map2.put(0b1111111, "magic");
```

And use the code `String instructionType = getType(b & 0x7f);` can get the last 7 bits of the data. Because if the 32 bits long data using and gate with 1111111_2 ($7f_{16}$), we can confirm we only get the last 7 bits.

```
public static String getType(int b) throws Exception {
    if (op32map2.containsKey(b)) {
        return op32map2.get(b);
    } else {
        return "WrongType";
    }
}
```

And compare them to the hashmap to get the type of instruction. For example using 11101011 00110011 11101011 00110011 and 11111111 can get 0110011. Then put it to the get type function, it will return string decodeTypeR.

4.3 Decode Type

The decode type function is designed to get the information of the data, this data is use to initial the direction of instruction. Each of the type have different decode method. So, I created 6 Decode Type function. (decodeTypeB, decodeTypeI, decodeTypeS, decodeTypeR, decodeTypeU, decodeMagic).

To get the right data for the instruction, I will use shifting to get the right position.

For example, the type R can be divided to:

```
public Line decodeTypeR(int arr[]) {  
  
    int rd = (arr[1] & 0x0f) << 1 | (arr[0] >> 7);  
  
    int rs1 = (arr[2] & 0x0f) << 1 | (arr[1] >> 7);  
  
    int rs2 = (arr[3] & 0x01) << 4 | (arr[2] & 0xf0) >> 4;  
  
    int funct3 = (arr[1] & 0x70) >> 4;  
  
    int funct7 = (arr[3] >> 1);  
}
```

After those coding, we can get the funct7, rs2, rs1, funct3 and rd value. It means we already get all of the instruction information. addi a0, a0, -4

For example we get the data 1111111110001010000010100010011 (addi a0, a0, -4).

In this case, the funct7 is 1111111, r2 is 11100, r1 is 01010, funct3 is 000, rd is 01010 and opcode is 0010011. Then the other can be using those data to decode the instruction.

4.4 Get Type Instruction

The get type instruction function is designed to get the instruction by their code. And there are 6 types of get type instruction function (getTypeBInstruction,

getTypeIInstruction, getTypeSInstruction, getTypeRInstruction, getTypeUInstruction, getMagicInstruction).

In this moment, we already get the type of instruction and we will use their type of get type instruction function. The example on 4.3 shows that the opcode is 0010011 and funct3 is 000.

```
if (opcode == 0b0000011) {  
    if (funct3 == 0) {  
        instruction = "lb";  
        ...  
    }  
    if (opcode == 0b0010011) {  
        if (funct3 == 0) {  
            instruction = "addi";
```

In this case, we can see that the instruction is addi. And after checking the instruction, then it will return the variable instruction.

4.5 Handle Type

The handle type instruction function is designed to execute the instruction. And there are 6 types of handle type function (handleTypeB, handleTypeI, handleTypeS, handleTypeR, handleTypeU, handleMagic).

This function is use to specific function of instruction. So, each of the instruction have their own coding.

```
if (line.instruction.equals("addi")) {
```

```

    if (rs1.equals("zero")) {

        registers.get(rd).setValue(line.imm);

    } else {

        long_val = registers.get(rs1).value + line.imm;

        registers.get(rd).setValue(long_val);

    }

```

It is the addi function example. It shows that the value of rs1 + the immediate value. Continue with the example on 4.3, the data 11111111110001010000010100010011 (addi a0, a0, -4) is means adding value of register a0 and -4 together and store it to register a0.

4.6 Dump To Json

The dump to json is used to let user check the information after execution. The value includes the operand, offset and the offset after mapping.

```

HashMap<String, Object> map = new HashMap();

map.put("bytes", ins);

map.put("registers", registers);

map.put("memoryOperands", memory.lastOperands);

JsonElement jsonElement = new Gson().toJsonTree(map);

arr.add(jsonElement);

```

Those code is use to put the data to the hashmap, and put the data in hashmap to the dump.json file.

4.7 The New Instruction

The new instruction is design to have a special function the RISC-V ISA doesn't have.

And the method is based on the normal instruction.

```
private void handleMagic(int arr[], Line line) throws Exception {  
  
    String rd = Registers.getRegXNum32(line.rd);  
  
    String rs1 = Registers.getRegXNum32(line.rs1);  
  
    long long_val;  
  
    System.out.println("hello "+line.instruction);  
  
    if (line.instruction.equals("magic")) {  
  
        long_val = registers.get(rs1).value + 2;  
  
        System.out.println("a : " + registers.get(rs1).value);  
  
        System.out.println("a + 2 : " + long_val);  
  
        registers.get(rd).setValue(long_val);  
  
    }  
  
    modifyPC();  
}
```

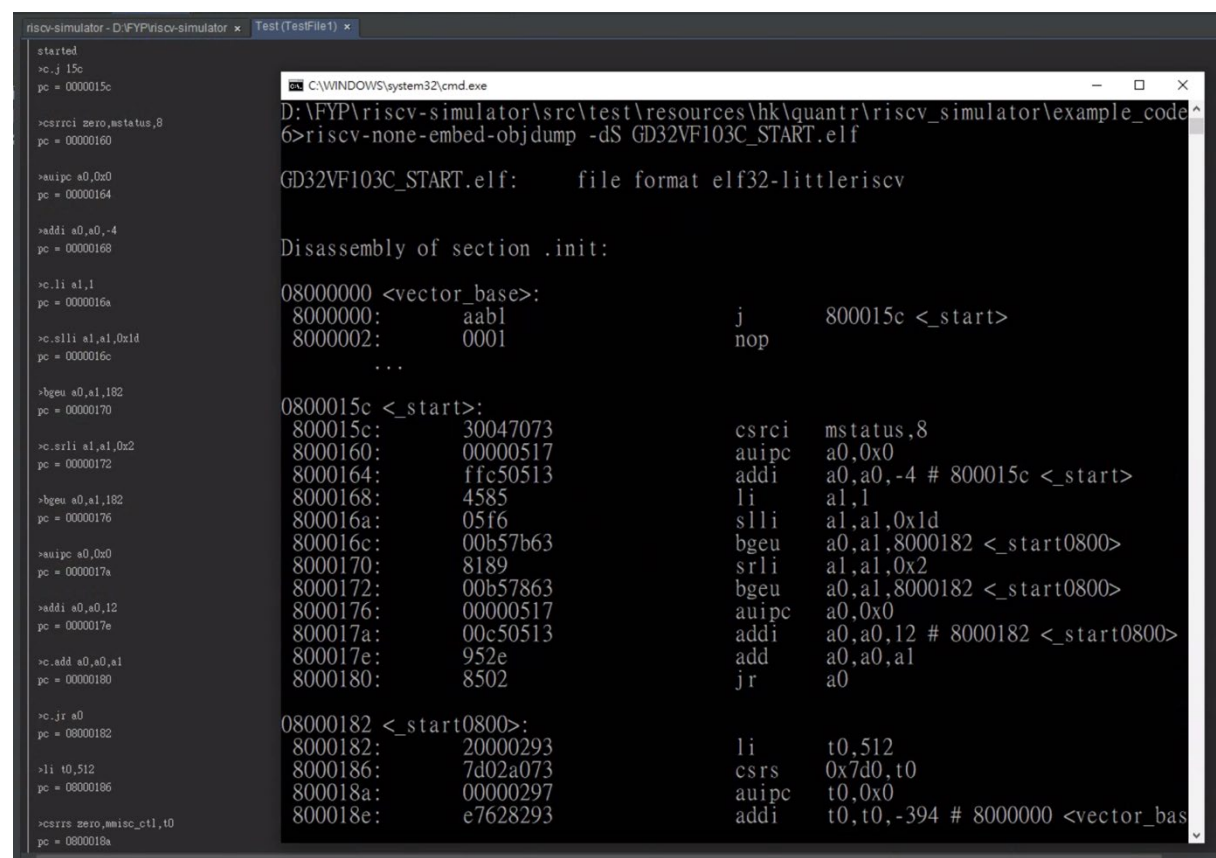
This is the source code of decoding the new instruction. This instruction is going to add two in to their original register. Because when we use the simulator there are some operate is the value added by two.

5. Results and Evaluation

This chapter will show the result and evaluation of the simulator. It will have 4 sections. Section 5.1 will show the execution of the instructions. Section 5.2 will show the function dump to json file. Section 5.3 will show how to edit the elf file. At the end, the new instruction will be presented in Section 5.4.

5.1 Execute the instruction

To show the executed instruction, I use the elf file to store the instruction and use the simulator to execute it. And compare the executed instruction and their register to the original elf file, as shown in Figure 5.1.



The image shows a screenshot of a computer screen with two windows. The left window is titled 'riscv-simulator - D:\FYP\riscv-simulator x' and contains a list of assembly instructions being executed, such as 'start', 'c.j 15c', 'csrrci zero, mstatus, 8', 'auipc a0, 0x0', 'addi a0, a0, -4', 'li a1, 1', 'slli a1, a1, 0x1d', 'bgeu a0, a1, 182', 'srli a1, a1, 0x2', 'bgeu a0, a1, 182', 'auipc a0, 0x0', 'addi a0, a0, 12', 'c.add a0, a0, a1', 'c.jr a0', 'li t0, 512', and 'csrrs zero, mstatus, t0, t0'. The right window is a command prompt titled 'C:\WINDOWS\system32\cmd.exe' showing the command 'D:\FYP\riscv-simulator\src\test\resources\hk\quantr\riscv_simulator\example_code^6>riscv-none-embed-objdump -dS GD32VF103C_START.elf' and its output. The output shows the disassembly of the 'GD32VF103C_START.elf' file, indicating it is in 'elf32-littleriscv' format. It displays the disassembly of the '.init' section, starting at address 08000000, and the '_start' section, starting at address 0800015c. The disassembly includes instructions like 'csrrci mstatus, 8', 'auipc a0, 0x0', 'addi a0, a0, -4 # 800015c <_start>', 'li a1, 1', 'slli a1, a1, 0x1d', 'bgeu a0, a1, 8000182 <_start0800>', 'srli a1, a1, 0x2', 'bgeu a0, a1, 8000182 <_start0800>', 'auipc a0, 0x0', 'addi a0, a0, 12 # 8000182 <_start0800>', 'add a0, a0, a1', 'jr a0', 'li t0, 512', 'csrrs 0x7d0, t0', 'auipc t0, 0x0', and 'addi t0, t0, -394 # 8000000 <vector_base>'.

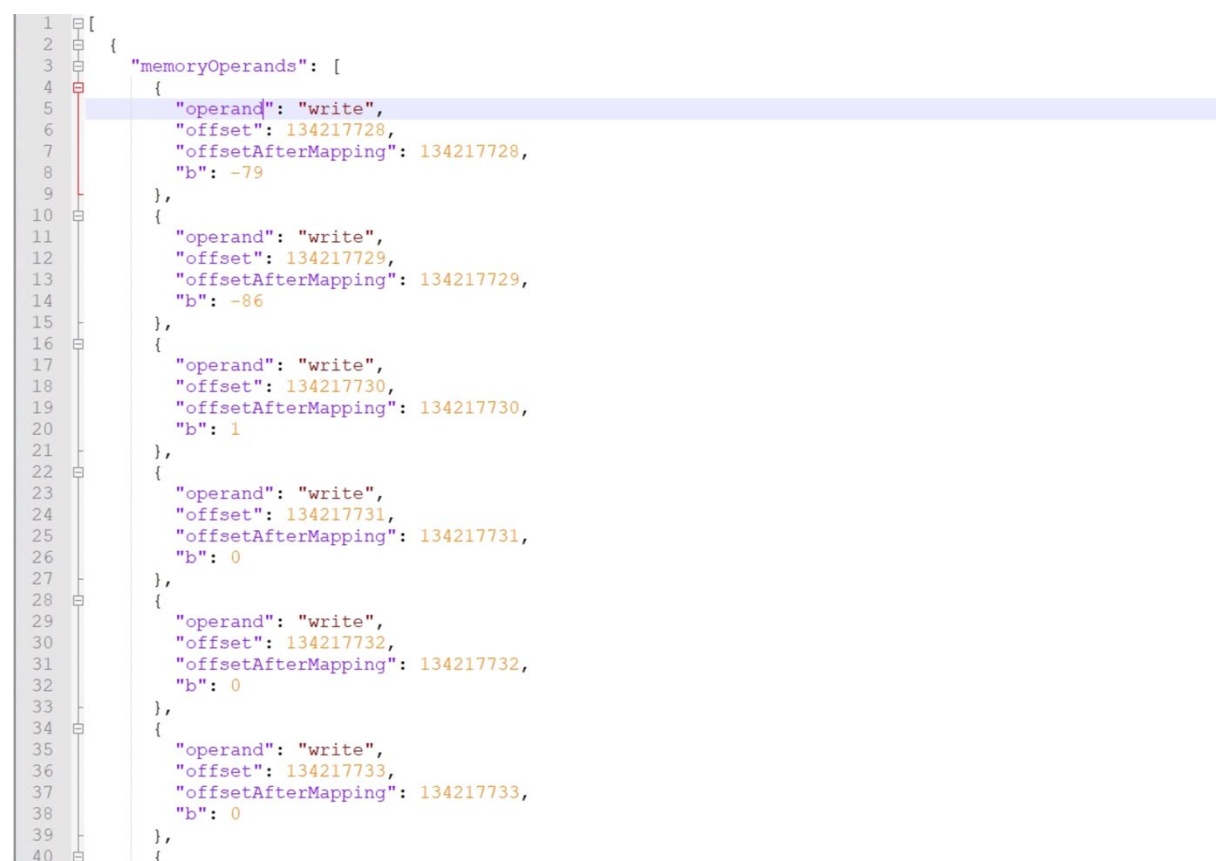
Figure 5.1

As the following instruction, the first instruction is j into register 8000c. Then the simulator will print the next register and run the binary code (30047073) in the next register and translate to an instruction (csrci mstatus,8).

And we can see that in Figure 5.1 the CMD is showing the instruction on elf file. It can be comparing with the simulator and confirm that there are no error occur.

5.2 Dump to Json File

Dump to json file function will generate the dump.json file to store the operand, offset and the offset after mapping, as shown in Figure 5.2

A screenshot of a code editor with a light gray background. On the left, there is a vertical scrollbar and a line number column ranging from 1 to 40. The main area displays a JSON array of memory operands. The first element is highlighted with a light blue background. The JSON structure is as follows:

```
1  [
2  {
3    "memoryOperands": [
4      {
5        "operand": "write",
6        "offset": 134217728,
7        "offsetAfterMapping": 134217728,
8        "b": -79
9      },
10     {
11       "operand": "write",
12       "offset": 134217729,
13       "offsetAfterMapping": 134217729,
14       "b": -86
15     },
16     {
17       "operand": "write",
18       "offset": 134217730,
19       "offsetAfterMapping": 134217730,
20       "b": 1
21     },
22     {
23       "operand": "write",
24       "offset": 134217731,
25       "offsetAfterMapping": 134217731,
26       "b": 0
27     },
28     {
29       "operand": "write",
30       "offset": 134217732,
31       "offsetAfterMapping": 134217732,
32       "b": 0
33     },
34     {
35       "operand": "write",
36       "offset": 134217733,
37       "offsetAfterMapping": 134217733,
38       "b": 0
39     },
40     {
```

Figure 5.2

When the simulator is executed, the dump to json function will execute parallelly. It

will start to write the result to the json file.

5.3 Edit Elf File

First, I use the `riscv-none-embed-readelf -S` command to show the section header, as shown in Figure 5.31.

```
D:\FYP\riscv-simulator\src\test\resources\hk\quantr\riscv_simulator\example_code6>riscv-none-embed-readelf -S GD32VF103C_START.elf
There are 22 section headers, starting at offset 0x18860:

Section Headers:
[Nr] Name                Type              Addr             Off              Size             ES Flg Lk  Inf Al
[ 0]                      NULL              00000000          000000           000000           00  0  0  0  0
[ 1] .init                  PROGBITS          08000000          001000           000236           00  AX  0  0  4
[ 2] .lalign                NOBITS            08000236          001236           000002           00  WA  0  0  1
[ 3] .text                  PROGBITS          08000240          001240           0018e4           00  AX  0  0 64
[ 4] .sdata2._glo[...]      PROGBITS          08001b24          002b24           000004           00  A   0  0  4
[ 5] .lalign                PROGBITS          08001b28          003068           000000           00  W   0  0  1
[ 6] .dalign                PROGBITS          20000000          003068           000000           00  W   0  0  1
[ 7] .data                  PROGBITS          20000000          003000           000068           00  WA  0  0  4
[ 8] .bss                   NOBITS            20000068          003068           000094           00  WA  0  0  4
[ 9] .stack                 NOBITS            20007800          003800           000800           00  WA  0  0  1
[10] .debug_info            PROGBITS          00000000          003068           0065ba           00  0   0  0  1
[11] .debug_abbrev          PROGBITS          00000000          009622           001788           00  0   0  0  1
[12] .debug_loc             PROGBITS          00000000          00adaa           002528           00  0   0  0  1
[13] .debug_aranges         PROGBITS          00000000          00d2d8           000630           00  0   0  0  8
[14] .debug_ranges          PROGBITS          00000000          00d908           000698           00  0   0  0  8
[15] .debug_line            PROGBITS          00000000          00dfa0           006448           00  0   0  0  1
[16] .debug_str             PROGBITS          00000000          0143e8           002772           01  MS  0  0  1
[17] .comment               PROGBITS          00000000          016b5a           000039           01  MS  0  0  1
[18] .debug_frame           PROGBITS          00000000          016b94           000fd4           00  0   0  0  4
[19] .symtab                SYMTAB            00000000          017b68           0007a0           10  20  63  4
[20] .strtab                STRTAB            00000000          018308           00047b           00  0   0  0  1
[21] .shstrtab              STRTAB            00000000          018783           0000dc           00  0   0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)
```

Figure 5.31

The instruction is store in `.text` and start at offset `0x1240`.

Then, use the binary editor to change the data. The data of first instruction is `30047073`, and I change it to `FFFFFFFF`, as shown in Figure 5.32 and Figure 5.33.

无标题 - x	GD32VF103C_START...	
000010D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000010E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000010F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001150	00 00 00 00 00 00 00 00 00 00 00 00 00 73 70 04 30	
00001160	17 05 00 00 13 05 C5 FF 85 45 F6 05 63 7B B5 00	
00001170	89 81 63 78 B5 00 17 05 00 00 13 05 C5 00 2E 95	
00001180	02 85 93 02 00 20 73 A0 02 7D 97 02 00 00 93 82	
00001190	62 E7 73 90 72 30 97 02 00 00 93 82 62 4A 73 90	
000011A0	C2 7E 73 E0 C0 7E 97 02 00 00 93 82 A2 41 73 90	
000011B0	52 30 97 01 00 18 93 81 E1 6A 17 81 00 18 13 01	
000011C0	61 E4 17 25 00 00 13 05 65 96 97 05 00 18 93 85	
000011D0	65 E3 17 06 00 18 13 06 66 E9 63 FA C5 00 83 22	
000011E0	05 00 23 A0 55 00 11 05 91 05 E3 EA C5 FE 17 05	
000011F0	00 18 13 05 A5 E7 93 85 01 8A 63 77 B5 00 23 20	
00001200	05 00 11 05 E3 6D B5 FE 73 F0 02 32 17 15 00 00	
00001210	13 05 A5 62 EF 10 C0 5E EF 10 80 65 01 45 81 45	
00001220	C5 22 6F 10 80 5E 01 A0 73 E0 02 32 82 80 73 F0	
00001230	02 32 82 80 00 00 00 00 00 00 00 00 00 00 00 00	
00001240	0A 50 72 6F 67 72 61 6D 20 68 61 73 20 65 78 69	
00001250	74 65 64 20 77 69 74 68 20 63 6F 64 65 3A 00 00	
00001260	30 78 00 00 6E 6D 69 0A 00 00 00 00 74 72 61 70	
00001270	0A 00 00 00 00 00 00 00 00 00 00 00 40 AF 40	
00001280	44 0A 00 00 C8 08 00 00 D8 09 00 00 E0 09 00 00	
00001290	D8 09 00 00 F8 08 00 00 D8 09 00 00 E0 09 00 00	
000012A0	C8 08 00 00 C8 08 00 00 F8 08 00 00 E0 09 00 00	
000012B0	3A 0A 00 00 3A 0A 00 00 3A 0A 00 00 F8 08 00 00	
000012C0	38 10 00 00 2E 0F 00 00 2E 0F 00 00 2C 0F 00 00	
000012D0	34 0F 00 00 34 0F 00 00 FA 0E 00 00 2C 0F 00 00	
000012E0	34 0F 00 00 FA 0E 00 00 34 0F 00 00 2C 0F 00 00	
000012F0	24 10 00 00 24 10 00 00 24 10 00 00 FA 0E 00 00	
00001300	00 01 02 02 03 03 03 03 04 04 04 04 04 04 04 04	
00001310	05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05	
00001320	06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06	
00001330	06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06	
00001340	07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07	
00001350	07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07	

Figure 5.32

无标题 - x	test.elf x	
000010D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000010E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000010F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00001150	00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF	
00001160	17 05 00 00 13 05 C5 FF 85 45 F6 05 63 7B B5 00	
00001170	89 81 63 78 B5 00 17 05 00 00 13 05 C5 00 2E 95	
00001180	02 85 93 02 00 20 73 A0 02 7D 97 02 00 00 93 82	
00001190	62 E7 73 90 72 30 97 02 00 00 93 82 62 4A 73 90	
000011A0	C2 7E 73 E0 C0 7E 97 02 00 00 93 82 A2 41 73 90	
000011B0	52 30 97 01 00 18 93 81 E1 6A 17 81 00 18 13 01	
000011C0	61 E4 17 25 00 00 13 05 65 96 97 05 00 18 93 85	
000011D0	65 E3 17 06 00 18 13 06 66 E9 63 FA C5 00 83 22	
000011E0	05 00 23 A0 55 00 11 05 91 05 E3 EA C5 FE 17 05	
000011F0	00 18 13 05 A5 E7 93 85 01 8A 63 77 B5 00 23 20	
00001200	05 00 11 05 E3 6D B5 FE 73 F0 02 32 17 15 00 00	
00001210	13 05 A5 62 EF 10 C0 5E EF 10 80 65 01 45 81 45	
00001220	C5 22 6F 10 80 5E 01 A0 73 E0 02 32 82 80 73 F0	
00001230	02 32 82 80 00 00 00 00 00 00 00 00 00 00 00 00	
00001240	FF FF 72 6F 67 72 61 6D 20 68 61 73 20 65 78 69	
00001250	74 65 64 20 77 69 74 68 20 63 6F 64 65 3A 00 00	
00001260	30 78 00 00 6E 6D 69 0A 00 00 00 00 74 72 61 70	
00001270	0A 00 00 00 00 00 00 00 00 00 00 00 40 AF 40	
00001280	44 0A 00 00 C8 08 00 00 D8 09 00 00 E0 09 00 00	
00001290	D8 09 00 00 F8 08 00 00 D8 09 00 00 E0 09 00 00	
000012A0	C8 08 00 00 C8 08 00 00 F8 08 00 00 E0 09 00 00	
000012B0	3A 0A 00 00 3A 0A 00 00 3A 0A 00 00 F8 08 00 00	
000012C0	38 10 00 00 2E 0F 00 00 2E 0F 00 00 2C 0F 00 00	
000012D0	34 0F 00 00 34 0F 00 00 FA 0E 00 00 2C 0F 00 00	
000012E0	34 0F 00 00 FA 0E 00 00 34 0F 00 00 2C 0F 00 00	
000012F0	24 10 00 00 24 10 00 00 24 10 00 00 FA 0E 00 00	
00001300	FF 01 FF 02 03 03 03 03 04 04 04 04 04 04 04 04	
00001310	05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05	
00001320	06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06	
00001330	06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06	
00001340	07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07	
00001350	07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07	

Figure 5.33

无标题 - x	test.elf x	
000010D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001150	00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF
00001160	17 05 00 00 13 05 C5 FF 85 45 F6 05 63 7B B5 00+ aE+c{d
00001170	89 81 63 78 B5 00 17 05 00 00 13 05 C5 00 2E 95	@ucx{.....+.o
00001180	02 85 93 02 00 20 73 A0 02 7D 97 02 00 00 93 82	.a0.. sa.}u...o6
00001190	62 E7 73 90 72 30 97 02 00 00 93 82 62 4A 73 90	btsEr0u...o6bJsE
000011A0	C2 7E 73 E0 C0 7E 97 02 00 00 93 82 A2 41 73 90	+sa-l-u...o6bAsE
000011B0	52 30 97 01 00 18 93 81 E1 6A 17 81 00 18 13 01	R0u...o6bJ.u....
000011C0	61 E4 17 25 00 00 13 05 65 96 97 05 00 18 93 85	az.9...e0u...o6
000011D0	65 E3 17 06 00 18 13 06 66 E9 63 FA C5 00 83 22	en.....fec+.a"
000011E0	05 00 23 A0 55 00 11 05 91 05 E3 EA C5 FE 17 05	...#AU...a.m0+...
000011F0	00 18 13 05 A5 E7 93 85 01 8A 63 77 B5 00 23 20	...R:o6a.ecw}.#
00001200	05 00 11 05 E3 6D B5 FE 73 F0 02 32 17 15 00 00	...mmj+sm.2....
00001210	13 05 A5 62 EF 10 C0 5E EF 10 80 65 01 45 81 45	...lbn.l+n.c.e.EUE
00001220	C5 22 6F 10 80 5E 01 A0 73 E0 02 32 82 80 73 F0	+p.c+.8so.26csw
00001230	02 32 82 80 00 00 00 00 00 00 00 00 00 00 00 00	.26c.....
00001240	FF FF 72 6F 67 72 61 6D 20 68 61 73 20 65 78 69	Program has exit
00001250	74 65 64 20 77 69 74 68 20 63 6F 64 65 3A 00 00	ted with code:..
00001260	30 78 00 00 6E 6D 69 0A 00 00 00 00 74 72 61 70	@x.nm{.....trap
00001270	0A 00 00 00 00 00 00 00 00 00 00 00 40 AF 40m@
00001280	44 0A 00 00 C8 08 00 00 D8 09 00 00 E0 09 00 00	D...k.....+
00001290	D8 09 00 00 F8 08 00 00 D8 09 00 00 E0 09 00 00	+...+.....
000012A0	C8 08 00 00 C8 08 00 00 F8 08 00 00 E0 09 00 00	k...k...+.....
000012B0	3A 0A 00 00 3A 0A 00 00 3A 0A 00 00 F8 08 00 00	t...t...t...t...
000012C0	38 10 00 00 2E 0F 00 00 2E 0F 00 00 2C 0F 00 00	8.....
000012D0	34 0F 00 00 34 0F 00 00 FA 0E 00 00 2C 0F 00 00	4...4.....
000012E0	34 0F 00 00 FA 0E 00 00 34 0F 00 00 2C 0F 00 00	4...4.....
000012F0	24 10 00 00 24 10 00 00 24 10 00 00 FA 0E 00 00	\$...\$...\$...\$...
00001300	FF 01 FF 02 03 03 03 03 04 04 04 04 04 04 04 04
00001310	05 05 05 05 05 05 05 05 05 05 05 05 05 05 05 05
00001320	06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06
00001330	06 06 06 06 06 06 06 06 06 06 06 06 06 06 06 06
00001340	07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
00001350	07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07

Figure 5.34

The original data is 30047073; register is 0x15c and instruction csrcl mstatus,8. Then the data change to ffff and register divided to two register 0x15c and 0x15e and they

can't be translating to the instruction. It is because the ffff is not the type of RISC-V architecture.

5.4 New instruction

The screenshot shows a RISC-V simulator window titled "C:\WINDOWS\system32\cmd.exe". The window is split into two panes. The left pane shows the assembly code being executed, and the right pane shows the register values and the current instruction.

```

started
>c.j 15c
pc = 0000015c

>This is type Magic
magic
hello magic
a : 0
a + 2 : 2
pc = 00000160

>auipc a0,0x0
pc = 00000164

>addi a0,a0,-4
pc = 00000168

>c.li a1,1
pc = 0000016a

>c.slli a1,a1,0x1d
pc = 0000016c

>bgeu a0,a1,182
pc = 00000170
  
```

The right pane shows the memory layout and the current instruction:

```

08000000 <vector_base>:
80000000: aabl          j      800015c <_start>
80000002: 0001          nop
...
0800015c <_start>:
800015c: ffff          0xffff
800015e: ffff          0xffff
8000160: 00000517      auipc   a0,0x0
8000164: ffc50513      addi    a0,a0,-4 # 800015c <_start>
8000168: 4585          li      a1,1
800016a: 05f6          slli   a1,a1,0x1d
800016c: 00b57b63      bgeu   a0,a1,8000182 <_start0800>
8000170: 8189          srli   a1,a1,0x2
8000172: 00b57863      bgeu   a0,a1,8000182 <_start0800>
8000176: 00000517      auipc   a0,0x0
800017a: 00c50513      addi    a0,a0,12 # 8000182 <_start0800>
800017e: 952e          add     a0,a0,a1
8000180: 8502          jr     a0
  
```

Figure 5.4

In Section 5.3, it shows how to change the value of the elf file. It is use for the new instruction. In Figure 5.4, It shows the new instruction is success to execute. When we execute that instruction the simulator will print the value of the register and the value of register after it added by two.

6. Conclusion and Further Work

The simulator can decode the elf file and provide a simulation result. And the new instruction is already created. That means the simulator is meet the targets of project objectives. For now, the simulator is a functionable simulator. And it can easily extensible and modifiable.

In creating the simulator, it has a lot of knowledge I should learn. It takes a lot of time

to fulfill the basic knowledge should use in the simulator. And the most challenging is the Handle Type function, it has a lot of try and fail when I modified the function of instruction.

In further, I hope the new instruction can have more function. For now, I created a type for new instruction. But in this type, it only has one instruction. So, it can be creating more instruction of this type. And make the simulator have more function.

7. References

https://linuxhint.com/understanding_elf_file_format/

<https://riscv.org/>