CS 3432 Computer Architecture I
Lab Assignment 2 – RV32 Interpreter

Teaching Assistant: Steven Ibarra
Instructional Assistant: Salvador Robles Herrera
Assigned: February 16, 2022
Due: February 28, 2022 @ 11:59

Introduction

Interpreters are computer programs that directly run a program in the given language without needing it compiled into machine language. In this lab, you will build a 32-bit RISC-V (RV32) interpreter in C that will receive, parse, and execute RISC-V instructions as strings. Your interpreter will not be a simulator as it is not mimicking the hardware operation (e.g., utilizing clock cycles), but the resulting registers and memory should reflect what you would find after executing real RV32 instructions.

Labs must be completed alone. You may discuss ideas and concepts with your peers, but never share your code with another student. All submissions will be compared with another to check for plagiarism. Treat your code like your finances, you could discuss how much money you have with your peers, but you should never share your credit card information.

Objectives

By the end of this lab, you will:
1. Effectively parse strings in C
2. Understand memory data representation in the computer
3. Understand RISC-V at a higher level and execute its instructions

Your Task

You are to design a function

```
bool interpret(char* instr)
```

that will parse and execute a 32-bit RISC-V instruction. This function takes in user input as a string and once program has parsed and executed the instruction, will return **true** if the instruction properly executed and **false** otherwise (i.e., return **false** if you could not parse or the instruction parameters were bad). Your program will continue looping, asking the user for input, parsing the user input, and executing the instruction, until the user submits the **EOF** character (ctrl+D on windows) where the program should exit gracefully and print the value of the registers. The major data components will be represented as:

1) Registers -> An array of 32-bit integers (32 total ints)
2) Memory -> a large .txt file containing the memory address and 32-bit data (word)
3) Immediates -> These will translate to normal integers in C

Because this lab is about understanding RISC-V at a low level, you will be provided with functions that will read and write to the memory text file for you so you do not need to worry about reading or writing to files in C. Instead, we would rather you focus on the low-level data manipulation. Hopefully, this also prevents you from accidentally corrupting the **mem.txt** file.

You will be responsible for interpreting and implementing the following instructions:

	RISC-V Instruction	Example String
1) Loads:		
a. Load Word:	LW,RD,RS1,IMM	“LW X7 1000(X5)”
2) Stores:		
a. Store Word:	SW,RD,RS1,IMM	“SW X5 2000(X0)”
3) Arithmetic:		
a. Add:	ADD,RD,RS1,RS2	“ADD X6 X0 X29”
b. Add Immediate:	ADDI,RD,RS1,IMM	“ADDI X6 X6 329”
4) Logical (Extra Credit):		
a. AND:	AND,RD,RS1,RS2	“AND X5 X6 X7
b. OR:	OR,RD,RS1,RS2	“OR X6, X28 X0”
c. XOR:	XOR,RD,RS1,RS2	“XOR X7, X5, X6”

Note: the above examples use **space** as a delimiter, but you may use a **comma** if you wish. Just make it clear to the user.

You may assume the following:

1) User input will be in all upper-case
2) Delimiter will be your choice
3) You may assume base 10 or base 16 for immediate addressing (whichever is easiest for you, but be clear in your instructions to the user)
4) User input will be valid instructions and not trying to test your robustness
5) Registers will use their x# notation when given by the user (e.g., “x5” or “x31”)

Parsing Strings

A major part of this lab will be parsing C strings to understand which RISC-V instructions you need to perform, and on which registers and memory addresses. In the previous lab, the only restriction I made in terms of C libraries was that you could not use the built-in string tokenizer **strtok** (I hope for obvious reasons). However, this time around, I will be restricting the **string.h** library. The only functions you will be allowed to use from this library are **fgets** and **strtok.** This restriction includes functions like **strcmp**, **strcpy**, **strlen,** and more. If you are unsure if a function you are using falls under **string.h** try commenting out the **"#include string.h"** and see if the compiler complains with a warning. The exception for the string.h library ban is on your tokenizer and using **fgets** for gathering user input. So, if you wish to check if two strings are equal, you will need to implement the function yourself.

A string tokenizer will be quite useful in this lab as you will have typically 3-4 tokens per instruction of various lengths and if you could split these up by delimiters, parsing the instruction becomes much easier. You may use the built-in String tokenizer **strtok** if you did not complete Lab 1 or are unsatisfied with the results, but if you use your own custom String tokenizer (you can add to your previous lab's tokenizer, but it should clearly be your own work), then you will get 10 points extra credit toward this lab.

Imagine the string: "SW X5 2000(X0)"
Throw this through our tokenizer and we have:
tokens[0]: "SW"
tokens[1]: "X5"
tokens[2]: "2000(X0)"

so if we had a way to figure out how to compare tokens[0] with a collection of strings, we could know how to operate on the rest of the tokens. Something akin to:
equals(tokens[0], "SW")
then we know whatever is at tokens[1] is the destination register for the store-word instruction. We just need to figure out how to go from "**X5**" to **r[5]** from the register array. The next difficult part is the relative address in tokens[2]. We can't just immediately check the 6th char in tokens[2], because the immediate at the beginning of the token may be any number of characters long. However, we could potentially make use of our tokenizer again. What if we used **tokenize(tokens[2], '(')**? We would end up with 2 tokens: "**2000**" and "**X0)**".

Reading and Writing to Memory

Memory plays are large role in how a computer functions. In this lab, we are representing memory as a large .txt file named **mem.txt**. You do not need to know how to read or write to C files, but you will need to understand what is going on in this **mem.txt** file every line represents a 32-bit address with its corresponding data. Because we are using byte-addressing and we are modelling a 32-bit architecture, every address is represented in hex (base-16) and the data is 32-bits represented by 1 column of 4 bytes, known as a word. We hex here because 4 bytes (32 bits) will always translate to 8 hex digits (2 hex digits per byte). However, this is not the case for decimal (base 10) and the data would be harder to read.

Address	Word
0x00000000	786F2EAB

Instead of reading or writing to the mem.txt file directly, you are provided 2 helper functions **read_address** and **write_address** as seen below:

```
int32_t read_address(int32_t address, char* file_name);
int32_t write_address(int32_t data, int32_t address, char* file_name);
```

read_address takes in a memory address matching what is found in **file_name** (which should almost always be **mem.txt** unless you wish to test other mem files) and returns the entire 32-bit integer data found at the address. **PLEASE NOTE** that we are using int32_t to ensure that we are using 32-bit integers, but really for most purposes they can be considered as a **long**. This is just to standardize it to 32-bit integers in case someone is operating on a 32-bit machine (where longs are 32-bit). The return value of **read_address** is just a 32-bit integer. Even though it is stored as a big ugly hex string it is functionally the exact same as its corresponding base-10 int value.

Much like:

```
	int a = 0xF;
	int b = 15;
	bool same a == b;
```

would have **same** be equal to **true**, because at the bit-level these two ints are the exact same number, regardless of their representation.

write_address receives a 32-bit **data** and writes it to specified memory address in file_name (typically **mem.txt**) and returns **data** if the write was successful or **NULL** if the write was unsuccessful.

```
    int32_t data = read_address(0, "mem.txt");
```

data will contain all 32-bits or (0x786F2EAB in hex and 86782063 in decimal)

Hints

To build your interpreter you will need to determine which instruction you are executing, which registers are being affected and which pieces of memory are being affected. There will be a couple major obstacles to overcome:

- Finding out which instruction you are executing
- How can you manipulate the 32-bit integers? Convert to a binary string and manipulate the chars? Manipulate the actual bits? Convert to a hex string and manipulate the chars?
- If writing your own atoi (string to integer conversion) function, think about how you would handle negative numbers like calling: atoi("-1600")
  hint: a negative symbol '-' is a lot like multiplying by negative 1, right?

Finding out which instruction you are dealing with might be the first step. Because you need to implement so many instructions, so you need a way to compare strings. Also, a switch statement sounds appropriate, but unlike Java, you cannot switch a **char***. You can only switch an **int**, but perhaps you could write a function that checks which instruction and returns an int to be used by a switch statement?

There will be many solutions to manipulating the data to and from the mem file. Perhaps you could convert the **int32_t** data into an array of ints to manipulate easily? Perhaps convert it to an array of chars? But if you do manipulate to an array, you would also need a way to convert it back to **int32_t**. You could also manipulate the bit logic.

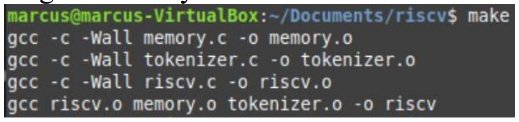
Files you are given

You are given multiple resources to help you complete the lab. Inside the Lab2 folder found in Teams, you should find 5 documents:

1) This PDF handout
2) **riscv.c**: This is the C program you will be implementing
3) **populate.c**: This a pre-written C program I used to initialize the mem file.
4) **mem.txt**: This is the memory file that represents the computer's memory for your RV32 interpreter
5) **memory.c**: This is a pre-written program containing **read_address** and **write_address**, 2 functions to help you write to the mem file.
6) **memory.h**: header filder for memory.c containing 2 function signatures.
7) **Makefile**: Make files help build C programs. This one is prewritten for you, but you might need to change the tokenizer file. Open the Makefile and name the **CUSTOM_TOKENIZER** value to the name of your tokenizer without a file extension. The default is "tokenizer" but if your tokenizer.c file is called, "Lab1Tokenizer.c" then you will need to change this value to:

CUSTOM_TOKENIZER=Lab1Tokenizer
If not using a custom tokenizer (i.e., if you want to use **strtok**), then set:
USE_TOKENIZER=false

To build and run this program, you just need all the above files (minus this PDF) in the same folder, including your tokenizer and tokenizer.h file (look into **memory.h** to see how to build header files) then just type:
make
This should compile everything to be ready to run like below:



```
marcus@marcus-VirtualBox:~/Documents/riscv$ make
gcc -c -Wall memory.c -o memory.o
gcc -c -Wall tokenizer.c -o tokenizer.o
gcc -c -Wall riscv.c -o riscv.o
gcc riscv.o memory.o tokenizer.o -o riscv
```

You can also run
make clean
to clean up all the leftover **.o** files.

Grading (50 total points| 11 points extra credit)

1. Implement **interpret** function that can run RV32 instructions (5 pts)
2. Implement required instruction (5 pts per instruction | 20 pts max)
3. Correctly update registers per instruction (5 pts)
4. Correctly read and write to memory (5 pts)
5. Design code to be easily interfaceable with user to show **interpret** function (5 pts)
6. Submit clean code that is well documented/commented (5 pts)
7. Demo code to TA or IA's within 1 weeks of your submission (5 pts)
8. Utilize your own custom String Tokenizer (5 points extra credit)
9. Implement logical instructions (2 points extra credit per instruction | 6 points max)

Late Penalty: Code is due on Monday, **February 28** @ 11:59pm. Once late, .5 point will be taken off every hour up to the first 10 hours on the FIRST DAY ONLY. After this, 5 points will be taken off per day late.

Hand-In Procedures

This lab will be pushed onto your GitHub repository and submitted to teams with your repo link by the due date of **February 28** @ 11:59pm**. Late Penalty**: Project is due on Monday, **February 28** @ 11:59pm. Once late, 0.5 point will be taken off every hour up to the first 10 hours on the FIRST DAY ONLY (Tuesday, March 01). After this, 5 points will be taken off per day late.

Demos

Your code will need to be demoed within 1 weeks of your submission. In the demo, you will walk through your solution and may need to answer questions from Steven, Alan, or Esteban about aspects of your code. Please mention any troubles you encountered during the lab and resources you used (online tutorials, the book, etc.). We anticipate most demos will take roughly 10 -15 minutes but may take longer if there are numerous questions about the assignment or your solution.

.