P4: RISC-V Interpreter

11/7/2023

100 Points Possible

Add Comment

10/25/2023

Details

Reminder: This is an individual project. Do not show your work to anyone or look at the work of anyone (in the class or online). Everything you submit must be implemented by you personally.

This is also the last project for which you can use your slip days! No slip days can be used for P5.

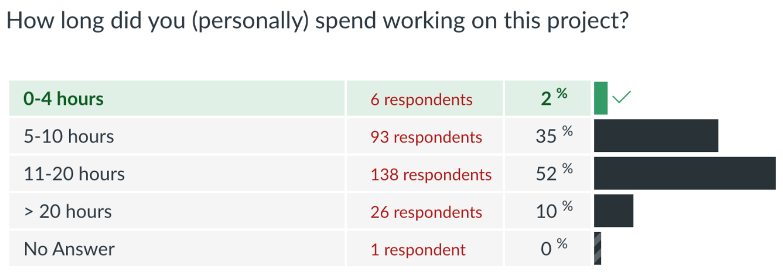
Overview

In this project, you will be implementing a (partial) RISCV interpreter in C. Your interpreter will support a subset of the instructions that you implemented in Logisim in P2.

We provide several starter files, which you should also familiarize yourself with before beginning the project. Your job is to implement everything marked as `// TODO:` in `linkedlist.c, hashtable.c, riscv.c.`

While it is certainly possible to use the native C compiler on your machine, we highly suggest that you make sure that your implementation works in either the course-provided VM, or SSH on the UGCLinux machines. Our autograder will be grading your assignment using the same environment as the VM/SSH.

Heads up! Last semester, students who filled out the survey reported the following:



How long did you (personally) spend working on this project?

0-4 hours	6 respondents	2 %	✓
5-10 hours	93 respondents	35 %	
11-20 hours	138 respondents	52 %	
> 20 hours	26 respondents	10 %	
No Answer	1 respondent	0 %	

What to Submit

- A complete implementation of `linkedlist.c`
- A complete implementation of `hashtable.c`
- A complete implementation of `riscv.c`

Compiling

In the shared Github repository, you should find starter files. We recommend taking a look at the header files (`*.h`) and the source files (`*.c`).

A `Makefile` is provided to help with compiling your code. As you progress through each section of this project, there will be a `make` target that you can use to compile the main function. This will generate an executable that you can run to test your code. If you are interested in reading more about Makefiles, Section F.6 of **this chapter (http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf)** is a good read.

The available options for `make` are as follows:

1. `make linkedlist`
2. `make hashtable`
3. `make riscv_interpreter`

By default, `make` will create the `riscv_interpreter` executable to run.

Simulating Memory

Since you will be supporting memory instructions, you need to have a way to represent memory.

A naive attempt would allocate a large array with 2^31 elements, and directly index into the array for each memory request. However, this poses certain limitations:

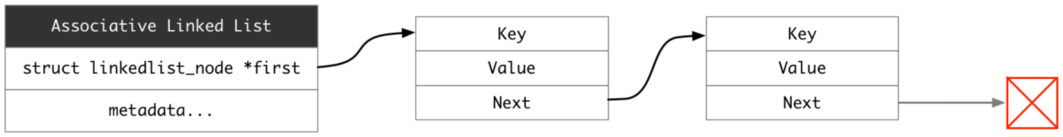
- the majority of the array will be unused, leading to waste
- it may not always be possible to allocate 2^31 bytes of memory for a single process (OS limitation)

Thus, we require you to simulate memory in a specific way, as specified below, which addresses both of the above limitations.

Associative Linked List

You may already be familiar with the notion of an associative linked list. It is one way to represent a dictionary, or a set of key-value mappings. Each node contains a key, a value, and a pointer to the next node. In this case, we are working with mappings from a 32-bit integer (the key, representing a memory address) to another 32-bit integer (the value at the given address).

The associative linked list can be visualized as follows:



For this part of the project, it will be valuable to walk through the **ArrayList Lab (https://canvas.cornell.edu/courses/54972/assignments/522694)** and familiarize yourself with pointers and structs.

Begin by implementing an associative linked list in `linkedlist.c`. The specification of each function can be found in `linkedlist.h`.

- `linkedlist_t *ll_init()`
- `void ll_free(linkedlist_t* list)`
- `void ll_add(linkedlist_t *list, int key, int value)`
- `int ll_get(linkedlist_t *list, int key)`
- `int ll_size(linkedlist_t *list)`

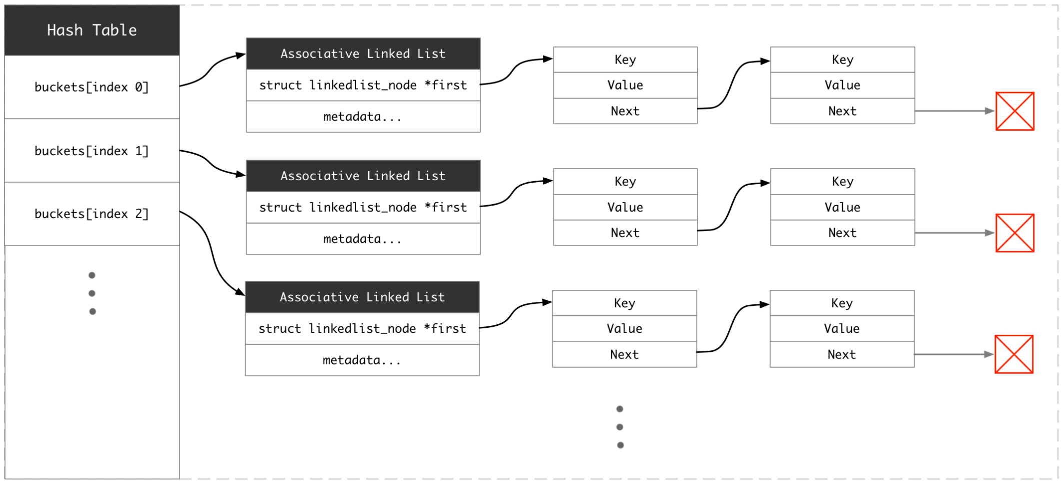
> Run `make linkedlist` to create a `linkedlist` executable to run. This will run the code found in `linkedlist_main.c`.

While an associative linked list can certainly simulate memory, it has performance limitations. You may have noticed that as you insert more key-value pairs into the linked list, the time it takes to find the matching key grows linearly. That is, the data structure provides an `O(n)` lookup. Let's see if we can use a different data structure to achieve a better performance.

Hash Table

You will now simulate memory in this project using a hash table, which provides much better performance than the associative linked list from the previous section. Once again, you are working with mappings from a 32-bit integer (the key, representing a memory address) to another 32-bit integer (the value at the given address).

To implement your hashtable, you **MUST** use your linked list implementation. That is, you should represent the buckets in your hash table using associative linked lists, which you previously implemented. It can be visualized as follows:



Implement the following functions in `hashtable.c`. The specification of each function can be found in `hashtable.h`.

- `hashtable_t *ht_init(int num_buckets)`
- `void ht_free(hashtable_t* table)`
- `void ht_add(hashtable_t *table, int key, int value)`
- `int ht_get(hashtable_t *table, int key)`
- `int ht_size(hashtable_t *table)`

You do not need to worry about the resizing case of a traditional hashtable. We intentionally use a fixed number of buckets, and will initialize your hashtable with a high number of buckets when grading the performance of your hashtable. Similarly, feel free to use a high number of buckets to simulate memory in your riscv interpreter, but do keep in mind memory constraints of the system (i.e. you don't need 2^31 buckets).

> Run `make hashtable` to create a `hashtable` executable to run. This will run the code found in `hashtable_main.c`.

Interpreter

For this part of the project, you should implement the `init(), end(), step(), evaluate_program()` functions in `riscv.c`. The specification for these functions can be found in `riscv.h`.

String Parsing

You will need to convert a string corresponding to a RISCV instruction to something that you can subsequently work with. Hence, you will have to do some string splitting. You may already be familiar with the tools available for string splitting; if not, it is worth reading up on them

You have probably noticed that RISCV instructions follow a certain format depending on the type of operation. Before you begin implementing anything, we strongly suggest that you take a look at the `get_op_type` function provided in `riscv.c`. We recommend that you use this, as it will likely make things simpler for you.

Spaces should not affect the correctness of your interpreter. For example:

- `addi x5,x0,12` should be handled the same way as `addi   x5,  x0 ,   12`
- `sw x12,4(x0)` should be handled the same way as `sw   x12 , 4  ( x0  )`

Your interpreter should support both decimal and hexadecimal immediate values. This means that the following are equivalent:

- `addi x4, x0, 17` should be equivalent to `addi x4, x0, 0x11`

- `sw x5, 16(x0)` should be equivalent to `sw x5, 0x10(x0)`

Operations

Your interpreter must support the following operations **in addition to empty strings (i.e. null terminator)**:

R-type	ADD, SUB, AND, SLT, SLL, SRA
I-type	ADDI, ANDI
Memory-type **	LW, LB, SW, SB
U-type	LUI
B-type	BEQ

** ***Note:*** *"Memory-type" is not a formal RISCV instruction type; we group the instructions this way to make parsing easier for you.*

Don't overthink these; most of them can be done quite simply. However, take note of edge cases and check that your interpreter behaves as specified in the **RISCV reference manual (https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf)** .

Memory is *little-endian*. We recommend implementing memory as **byte-addressed**, which will allow each memory address (`0x00000000`, `0x00000001`, ...) to exist as keys in your simulated memory, mapping to a value of a single byte. Other address modes are welcome (such as word-addressed), but discouraged.

Strings passed into your interpreter will either be empty strings or *valid* RISCV instructions. *Valid* means that the operation will be in the list of supported operations above, registers will be valid `x[0-31]`, immediates will be a maximum of 20 bits in either decimal or hexadecimal notation, at least one space follows the operation, and commas are used as the separator between tokens (there can still be any number of spaces).

Formally, all valid instructions satisfy the ***regex format (https://regex101.com)*** *(though, the set of tokens may not match the format of the operation)*:
`[a-z]+ +x[0-9]+ *, *((x[0-9]+ *, *(x[0-9]+|(0x[0-9a-f]+|-?[0-9]+)))|(0x[0-9a-f]+|-?[0-9]+)( *\( *x[0-9]+ *\))?)`

The Main Program

> Run `make riscv_interpreter` to create a `riscv_interpreter` executable to run. This will run the code found in `riscv_interpreter.c`.

To simulate instructions, type them in after running `./riscv_interpreter`. When you are done, press **Ctrl+D** to show the final values in the registers after evaluation. To pass in a whole test file in, you can type `./riscv_interpreter < test.txt`. To automate this process, feel free to use redirection from a file as shown in this Tutorial 3 of **this Unix/Linux guide (http://www.ee.surrey.ac.uk/Teaching/Unix/)** .

Starting values. It is possible to initialize a register with a 32-bit value for testing. The special comment directive `## start[<r>] = <value>` can be placed anywhere in your test program to set the value in register `<r>` to `<value>`. For example, `## start[4] = 0x123` will set the value of register 4 to 0x123.

Number of instructions. Since, to support BEQ, the program must "memorize" the provided instructions by dynamically allocating heap space, you will need to specify how many instructions your program will contain up front by writing the special comment directive `## cycles = <value>`. If not specified, the program defaults to 50 instructions.

> Try writing some sample programs to verify the correctness of your RISCV Interpreter! :)

Prevent memory leaks

There are three functions you need to implement to free memory and prevent memory leaks: `end(), ll_free(), ht_free()`. `end()` will be called after all the instructions are executed by the interpreter. It then calls `ll_free(), ht_free()` to free the memory you allocate for simulating the memory. You can use valgrind to test whether you have unfree memory by running

```
$ valgrind --leak-check=full ./riscv_interpreter < test.txt
```

Your output message should contain "All heap blocks were freed -- no leaks are possible".

Coding styles

You need to follow the **C Programming Style Guide (https://canvas.cornell.edu/courses/54972/pages/c-programming-style-guide)**. Coding styles are part of the grading in this project.

Get a local copy

scp: secure copy (remote file copy program)

If you want to get a copy of a file of Project 4 from the undergrad linux machine locally onto your machine, **scp** to the rescue!

Say your file **game.c** is located at **/home/yourNetID/mygame/game.c** on the undergrad linux machine. You can secure copy it from that machine to your own machine from the command line of your local machine. Simply type:

$ scp yourNetID@ugclinux.cs.cornell.edu:mygame/game.c .

Testing Overview

Make sure to fully test your linked list and hashtable implementation. Your implementation of the data structures should work for all possible edge cases, even the ones that are outside the context of the interpreter. When grading your code, we will do a robust test on linked list and hashtable. In order to pass our tests, just submitting a code that does not crash when running is not enough!

Hint: we will test RISCV with 100000 unique addresses. This information might help you decide how big you want to initialize your hashtable.

Notes & Hints

Help. Ask the instructor, TAs and consultants for help. Also check Ed for more help.

Read the docs. Refer to the **RISCV manual (https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf)** often. Refer to the **C Programming Style Guide (https://canvas.cornell.edu/courses/54972/pages/c-programming-style-guide)**.

What to do first. We suggest that you implement all required functions in the same order as the sections appear in this writeup. We recommend attempting to simulate the memory using an associative linked list first, before attempting to simulate the memory with a hash table.

Modifications. You only need to modify `linkedlist.c`, `hashtable.c`, `riscv.c`. Modification of any other files is not permitted, as our autograder will grade using the default version of these files, without your changes. For example, you may not add another `struct` definition to `riscv.h`.

Advice from past students.

Familiarize yourself greatly with the mechanics of a RISCV processor. Additionally, it is key that you are able to understand pointers, structs, arrays, and various other vital aspects of C programming that are different from many of the languages we have used before.

Spend a lot of time testing the interpreter, especially with edge cases, and to pay very close attention to the RISCV specification for each instruction/instruction type.

The last part takes about 7x as long as the first two parts combined.

Make sure you understand the ArrayList lab completely and fully (also documentation is your friend).

Acknowledgments. Concept for this project derived from https://dannyqiu.me/mips-interpreter/ (https://dannyqiu.me/mips-interpreter/) . Makefile reading material is from OS textbook OSTEP <http://pages.cs.wisc.edu/~remzi/OSTEP/>. (http://pages.cs.wisc.edu/~remzi/OSTEP/) The project is converted from MIPS-based to RISC-V-based by a team of 5 course staff during Fall 2019.

P3 (https://canvas.cornell.edu/courses/54972/assignments/522717) P5 (https://canvas.cornell.edu/courses/54972/assignments/522719)

(https://canvas.cornell.edu/courses/54972/modules/items/2052985) (https://canvas.cornell.edu/courses/54972/modules/items