Project 1: RISC-V Instruction Set Emulator

Computer Architecture I ShanghaiTech University

Project 1

> **IMPORTANT INFO - PLEASE READ**
>
> The projects are part of your design project worth 2 credit points. As such they run in parallel to the actual course. So be aware that the due date for project and homework might be very close to each other! Start early and do not procrastinate.

Goal

We hope this project will enhance your C programming skills, familiarize you with some of the details of RISC-V, and prepare you for what's to come later in this course.

Background

In this project, you will create an emulator that is able to execute a subset of the RISC-V ISA. You'll provide the machinery to decode and execute a couple dozen RISC-V instructions. You're creating what is effectively a miniature version of VENUS!

The RISC-V green card provides some information necessary for completing this project.

Getting started

Make sure you read through the entire specification before starting the project.

The whole project is split into two parts. Project 1.1 (Part 1) and Project 1.2 (Part 2). Those will be autograded separately and have their own deadlines. You start with Project 1.1/ Part 1 now - but the instructions for Project 1.2 are also given alreay.

You will be using gitlab to collaborate with your group partner. Autolab will use the files from gitlab. Make sure that you have access to gitlab. In the group CS110_Projects you should have access to your project 1 project. Also, in the group CS110, you should have access to the p1_framework.

Obtain your files

1. Clone your p1 repository from gitlab. You may want to change `http` to `https` (currently still requires the use of the school VPN).
2. In the repository add a remote repo that contains the framework files:
   `git remote add framework https://autolab.sist.shanghaitech.edu.cn/gitlab/cs110/p1_framework.git` (or change to http)

3. Go and fetch the files:
   `git fetch framework`
4. Now merge those files with your master branch:
   `git merge framework/master`
5. The rest of the git commands work as usual.

How to Autolab

Autolab will be available soon.

1. Edit the text file `autolab.txt`. The first line has to be the name of your p1 project in gitlab. So `p1_email1_email2`.
2. The following lines have to contain a long, random secret. Commit and push to gitlab. We will test the length and randomness of this secret by running `tar -cjf size.tar.bz2 autolab.txt`.
3. When you want to run the autograder in autolab, you have to upload your autolab.txt. Autolab will clone, from gitlab, the master branch of the repo specified in the `autolab.txt` you uploaded and then continue grading only if all of these conditions are met:
   1. The `autolab.txt` you uploaded and the one in the clone repo are identical.
   2. The size of the generated `size.tar.bz2` is at least 1000B.
   3. Only the files from the framework are present in the cloned repo.
4. Autolab will replace all files except `part1.c`, `utils.c`, `utils.h` and `part2.c` with the framework versions before compiling your code.

Collaborative Coding and Frequent Pushing

You have to work at this project as a team. We invite you to use all of the features of gitlab for your project, for example branches, issues, wiki, milestones, etc.

We require you to push very frequently to gitlab. In your commits we want to see how the code evolved. We do NOT want to see the working code suddenly appear - this will make us suspicious.

We also require that all group members do substantial contributions to the project. This also means that one group member should not finish the project all by himself, but distribute the work among all group members!
Gitlab has excellent tools to track that (see "Repository : Contributors"). At the end of Project 1 we will interview all group members and discuss their contributions, to see if we need to modify the score for certain group members.

Files

The files you will need to modify:

- `part1.c` - The main file which you will modify for part 1.
- `part2.c` - The main file which you will modify for part 2.
- `utils.c` - The helper file which will hold various helper functions for both part 1 and 2.
- `utils.h` - File that contains the format for instructions to print for part 1. Available for new declaration of auxiliary functions.

You should definitely consult through the following, thoroughly:

- `types.h` - C header file for the data types you will be dealing with.
- `Makefile` - File which records all dependencies.
- `riscvcode/*` - Various files to run tests.

You should not need to look at these files, but here they are anyway:

- `riscv.h` - C header file for the functions you are implementing.
- `riscv.c` - C source file for the program loader and main function.

When your project is done, please submit all the code including the framework to your remote GitLab repo by running the following commands.

```
$ git commit -a
$ git push origin master:master
```

You will NOT be submitting extra files. If you add a public helper functions, please place the function prototypes in `utils.h`. Besides, we only grade code on the `master` branch. If you do not follow these requirements, your code will likely not compile and you will get a zero on the project.

The RISC-V Emulator

The files provided in the start kit comprise a framework for a RISC-V emulator. You'll first add code to `part1.c` and `utils.c` to print out the human-readable disassembly corresponding to the instruction's machine code. Next, you'll complete the program by adding code to `part2.c` to execute each instruction (including perform memory accesses). Your simulator must be able to handle the machine code versions of the following RISC-V machine instructions. We've already given you a framework for what cases of instruction types you should be handling.

It is critical that you read and understand the definitions in `types.h` before starting the project. If they look mysterious, consult chapter 6 of K&R, which covers structs, bitfields, and unions. Check yourself: why does `sizeof(Instruction) == 4`?

The instruction set that your emulator must handle is listed below. All of the information here is copied from the RISC-V green sheet for your convenience; you may still use the green card as a reference.

INSTRUCTION	TYPE	OPCODE	FUNCT3	FUNCT7/IMM	OPERATION
`add rd, rs1, rs2`	R	0x33	0x0	0x00	R[rd] ← R[rs1] + R[rs2]
`mul rd, rs1, rs2`			0x0	0x01	R[rd] ← (R[rs1] * R[rs2])[31:0]
`sub rd, rs1, rs2`			0x0	0x20	R[rd] ← R[rs1] - R[rs2]
`sll rd, rs1, rs2`			0x1	0x00	R[rd] ← R[rs1] << R[rs2]
`mulh rd, rs1, rs2`			0x1	0x01	R[rd] ← (R[rs1] * R[rs2])[63:32]
`slt rd, rs1, rs2`			0x2	0x00	R[rd] ← (R[rs1] < R[rs2]) ? 1 : 0

INSTRUCTION	TYPE	OPCODE	FUNCT3	FUNCT7/IMM	OPERATION
sltu rd, rs1, rs2			0x3	0x00	R[rd] ← (*U*(R[rs1]) < *U*(R[rs2])) ? 1 : 0
xor rd, rs1, rs2			0x4	0x00	R[rd] ← R[rs1] ^ R[rs2]
div rd, rs1, rs2			0x4	0x01	R[rd] ← R[rs1] / R[rs2]
srl rd, rs1, rs2			0x5	0x00	R[rd] ← R[rs1] >> R[rs2]
sra rd, rs1, rs2			0x5	0x20	R[rd] ← R[rs1] >> R[rs2]
or rd, rs1, rs2			0x6	0x00	R[rd] ← R[rs1] | R[rs2]
rem rd, rs1, rs2			0x6	0x01	R[rd] ← (R[rs1] % R[rs2]
and rd, rs1, rs2			0x7	0x00	R[rd] ← R[rs1] & R[rs2]
lb rd, offset(rs1)	I	0x03	0x0		R[rd] ← *SignExt*(*Mem*(R[rs1] + offset, byte))
lh rd, offset(rs1)			0x1		R[rd] ← *SignExt*(*Mem*(R[rs1] + offset, half))
lw rd, offset(rs1)			0x2		R[rd] ← *Mem*(R[rs1] + offset, word)
lbu rd, offset(rs1)			0x4		R[rd] ← *U*(*Mem*(R[rs1] + offset, byte))
lhu rd, offset(rs1)			0x5		R[rd] ← *U*(*Mem*(R[rs1] + offset, half))
addi rd, rs1, imm		0x13	0x0		R[rd] ← R[rs1] + imm
slli rd, rs1, imm			0x1	0x00	R[rd] ← R[rs1] << imm
slti rd, rs1, imm			0x2		R[rd] ← (R[rs1] < imm) ? 1 : 0
sltiu rd, rs1, imm			0x3		R[rd] ← (*U*(R[rs1]) < *U*(imm)) ? 1 : 0
xori rd, rs1, imm			0x4		R[rd] ← R[rs1] ^ imm
srli rd, rs1, imm			0x5	0x00	R[rd] ← R[rs1] >> imm
srai rd, rs1, imm			0x5	0x20	R[rd] ← R[rs1] >> imm
ori rd, rs1, imm			0x6		R[rd] ← R[rs1] | imm
andi rd, rs1, imm			0x7		R[rd] ← R[rs1] & imm
jalr rd, rs1, imm		0x67	0x0		R[rd] ← PC + 4 PC ← R[rs1] + imm
ecall		0x73	0x0	0x000	(Transfers control to operating system) a0 = 1 is print value of a1 as an integer. a0 = 4 is print the string at address a1. a0 = 10 is exit or end of code indicator. a0 = 11 is print value of a1 as a character.
sb rs2, offset(rs1)	S	0x23	0x0		*Mem*(R[rs1] + offset) ← R[rs2][7:0]
sh rs2, offset(rs1)			0x1		*Mem*(R[rs1] + offset) ← R[rs2][15:0]
sw rs2, offset(rs1)			0x2		*Mem*(R[rs1] + offset) ← R[rs2]
beq rs1, rs2, offset	SB	0x63	0x0		if(R[rs1] == R[rs2]) PC ← PC + {offset, 1b'0}
bne rs1, rs2, offset			0x1		if(R[rs1] != R[rs2]) PC ← PC + {offset, 1b'0}

INSTRUCTION	TYPE	OPCODE	FUNCT3	FUNCT7/IMM	OPERATION
blt rs1, rs2, offset			0x4		if(R[rs1] < R[rs2]) PC ← PC + {offset, 1b'0}
bge rs1, rs2, offset			0x5		if(R[rs1] >= R[rs2]) PC ← PC + {offset, 1b'0}
bltu rs1, rs2, offset			0x6		if(*U*(R[rs1]) < *U*(R[rs2])) PC ← PC + {offset, 1b'0}
bgeu rs1, rs2, offset			0x7		if(*U*(R[rs1]) >= *U*(R[rs2])) PC ← PC + {offset, 1b'0}
auipc rd, offset	U	0x17			R[rd] ← PC + {offset, 12b'0}
lui rd, offset		0x37			R[rd] ← {offset, 12b'0}
jal rd, imm	UJ	0x6f			R[rd] ← PC + 4 PC ← PC + {imm, 1b'0}

For further reference, here are the bit lengths of the instruction components.

R-TYPE	funct7	rs2	rs1	funct3	rd	opcode
Bits	7	5	5	3	5	7

I-TYPE	imm[11:0]	rs1	funct3	rd	opcode
Bits	12	5	3	5	7

S-TYPE	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
Bits	7	5	5	3	5	7

SB-TYPE	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
Bits	1	6	5	5	3	4	1	7

U-TYPE	imm[31:12]	rd	opcode
Bits	20	5	7

UJ-TYPE	imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
Bits	1	10	1	8	5	7

Just like the regular RISC-V architecture, the RISC-V system you're implementing is little-endian. This means that when given a value comprised of multiple bytes, the least-significant byte is stored at the lowest address. If needed, review the lecture 4 slides 35-38.

The Framework Code

The framework code we've provided operates by doing the following.

1. It reads the program's machine code into the simulated memory (starting at address `0x01000`). The program to "execute" is passed as a command line parameter. Each program is given 1 MiB of memory and is byte-addressable.

2. It initializes all 32 RISC-V registers to `0x00000` and sets the program counter (PC) to `0x01000`. The only exceptions to the initial initializations are the stack pointer (set to `0xEFFFF`) and the global pointer (set to `0x03000`). In the context of our emulator, the global pointer will refer to the static portion of our memory. The registers and Program Counter are managed by the `Processor struct` defined in `types.h`.
3. It sets flags that govern how the program interacts with the user. Depending on the options specified on the command line, the simulator will either show a dissassembly dump (`-d`) of the program on the command line, or it will execute the program. More information on the command line options is below.

It then enters the main simulation loop, which simply executes a single instruction repeatedly until the simulation is complete. Executing an instruction performs the following tasks:

1. It fetches an instruction from memory, using the PC as the address.
2. It examines the `opcode/funct3` to determine what instruction was fetched.
3. It executes the instruction and updates the PC.

The framework supports a handful of command-line options:

- `-i` runs the simulator in interactive mode, in which the simulator executes an instruction each time the Enter key is pressed. The disassembly of each executed instruction is printed.
- `-t` runs the simulator in tracing mode, in which each instruction executed is printed.
- `-r` instructs the simulator to print the contents of all 32 registers after each instruction is executed. This option is most useful when combined with the `-i` flag.
- `-d` instructs the simulator to disassemble the entire program, then quit before executing.

In part 2, you will be implementing the following:

- The `execute_instruction()`
- The various `executes`
- The `store()`
- The `load()`

Many `UNUSED` modifiers, which you may find in the declaration of functions, are added to suppress unused-variable-warnings. You can remove them immediately you finish the code.

By the time you're finished, they should handle all of the instructions in the table above.

Part 1

Your first task is to implement the disassembler by completing the `decode_instruction()` method in `part1.c` alongside various other functions.

The goal of this part is, when given an instruction encoded as a 32-bit integer, to reproduce the original RISC-V instruction in human-readable format. For this part, you will not be referring to registers by name; instead, you should refer to registers by their

numbers (as defined on the RISC-V Green Card). Please look at the constants defined in `utils.h` when printing the instructions. More details about the requirements are below.

1. Print the instruction name. If the instruction has arguments, print a tab (`\t`).
2. Print all arguments, following the order and formatting given in the INSTRUCTION column of the table above.
    - Arguments are generally comma-separated (`lw/sw`, however, also use parentheses), but are not separated by spaces.
    - You may find looking at `utils.h` useful.
    - Register arguments are printed as an `x` followed by the register number, in decimal (e.g. `x0` or `x31`).
    - All immediates should be displayed as a signed decimal number.
    - Shift amounts (e.g. for `sll`) are printed as unsigned decimal numbers (e.g. 0 to 31).
3. Print a newline (`\n`) at the end of an instruction.
4. We will be using an autograder to grade this task. If your output differs from ours due to formatting errors, you will not receive credit.
5. We have provided some disassembly tests for you. However, since these tests only cover a subset of all possible scenarios, passing these tests do not mean that your code is bug free. You should identify the corner cases and test them yourself.

To implement this functionality, you will be completing the following:

- The `decode_instruction()` function in `part1.c`
- The various `writes` in `part1.c`
- The various `prints` in `part1.c`
- The various `gets` in `utils.c`
- The helper functions e.g. `bitSigner` in `utils.c`

You may run the disassembly test by typing in the following command. If you pass the tests, you will see the output listed here.

```
$ make part1
gcc -g -Wall -Werror -Wfatal-errors -O2 -o riscv utils.c part1.c part2.c riscv.c
simple_disasm TEST PASSED!
multiply_disasm TEST PASSED!
random_disasm TEST PASSED!
---------Disassembly Tests Complete---------
```

The tests provided do not test every single possiblity, so creating your own tests to check for edge cases is vital. If you would like to only run one specific test, you can run the following command:

```
$ make [test_name]_disasm
```

To create your own tests, you first need to create the relevant machine code. This can either be done by hand or by using the Venus simulator. You should put the machine instructions in a file named `[test_name].input` and place that file inside `riscvcode/code`. Then,

create what the output file will look like `[test_name].solution` and put this output file in `riscvcode/ref`. See the provided tests for examples on these files. To integrate your tests with the `make` command, you must modify the Makefile. On Line 4 of the `Makefile`, where it says `ASM_TESTS`, add `[test_name]` to the list with spaces in between file names.

To run your code through `cgdb`, you can compile your code using `make riscv`. Then you can run the debugger on the riscv executable. You will need to the supply input file as a command-line argument within the debugger.

If your disassembly does not match the output, you will get the difference between the reference output and your output. Make sure you at least pass this test before submitting `part1.c`.

For this part, only changes to the files `part1.c`, `utils.c`, and `utils.h` will be considered by the autograder. The test environment on autolab is Ubuntu 16.04 with gcc 5.x.

Part 2

Your second task is to complete the emulator by implementing the `execute_instruction()`, `execute()'s`, `store()`, and `load()` methods in `part2.c`

This part will consist of implementing the functionality of each instruction. Please implement the functions outlined below (all in `part2.c`).

- `execute_instruction()` - executes the instruction provided as a parameter. This should modify the appropriate registers, make any necessary calls to memory, and updatge the program counter to refer to the next instruction to execute.
- `execute()'s` - various helper files to be called in certain conditions for certain instructions. Whether you use these functions is up to you, but they will greatly help you organize your code.
- `store()` - takes an address, a size, and a value and stores the first -size- bytes of the given value at the given address. The `check_align` parameter will enforce alignment constraints when the parameter is 1. We include this parameter to enforce that instructions are word-aligned. When implementing `store` and `load` instructions, this paramter should be 0 since RISC-V does not enforce alignment constraints.
- `load()` - takes an address and a size and returns the next -size- bytes starting at the given address. The `check_align` is the same as that of `store()`.

Here is an implementation guideline for you. To save your time spent on understanding the whole framework, please consider the following tips.

1. The `main` function ends up with a dead loop, which means simulation will not quit even if it reaches the last instruction. Every input should contain an exit `ecall` at the ending. When you're creating .input files, please pay attention to this.
2. The emulator invokes function `load` to read instructions from a simulated memory. To enable instruction fetching, you need to implement `load` first.
3. However, the emulator will still trap into the dead loop before exit ecall can work. Hence, we suggest you implement `execute_ecall` before instructions of other types. **You are required to print the message "exiting the simulator" followed by a newline "\n" before exiting simulation. The exit code should always be 0.**

4. You might expect instructions will be automatically fed into `execute_instruction` line by line. However, the emulator will not fetch the next instruction unless you set the PC register correctly. You can simply insert a line for increasing PC register before you implement jump instructions.

We have provided a simple self-checking assembly test that tests several of the instructions. However, the test is not exhaustive and does not exercise every instruction. Here's how to run the test (the output is from a working processor).

```
$ make part2
gcc -Wall -Werror -Wfatal-errors -O2 -o riscv utils.c part1.c part2.c riscv.c
simple_execute TEST PASSED!
multiply_execute TEST PASSED!
random_execute TEST PASSED!
-----------Execute Tests Complete-----------
```

Most likely you will have bugs, so try the tracing mode or other debugging modes described in the Framework section above.

We have provided a few more tests and the ability to write your own. Just like part1, you will have to create .input files and put them in the relevant folders. However, for part 2, you will want to name your solution file with a `.trace` instead.

1. Create the new assembly file in the riscvcode directory (use `riscvcode/simple.input` as a template)
2. Add the base name of the test to the list of ASM_TESTS in the Makefile. To do this, just add [test_name] to the end of line 4.

Now build your assembly test, and then run it by typing in the following commands:

```
$ make [test_name]_execute
```

You can, and indeed should, write your own assembly tests to test specific instructions and their corner cases. Furthermore, you should be compiling and testing your code after each group of instructions you implement. It will be very hard to debug your project if you wait until the end to test.

For the final results, only changes to the files `part1.c`, `part2.c`, `utils.c`, and `utils.h` will be considered by the autograder. The test environment on autolab is again Ubuntu 16.04 with gcc 5.x.

TAs: *Peihao Wang, Anqi Pang, Tianyuan Wu, Hang Su*
Last Modified: Apr. 6th, 2020