CS61C Summer 2018 Project 2: RISC-V Instruction Set Emulator

TA: Sean Farhat

Part 1 due Friday, July 6th @ 23:59:59 PST
Part 2 due Friday, July 13th @ 23:59:59 PST

Updates:

Any updates will be listed here. To pull updates at any time, run the following:

```
$ git pull proj2-starter master
```

- 7/3 12:15 PM: There was a bug in how the immediate field was stored in the union, please pull the update
- 7/5 11:34 AM: A new test, unsigned.input/.solution, was added. This should cover more instructions, but remember, not all instructions are covered by the given tests, so please be sure to make your own. Instructions for making your own tests are on the project website. Also, make sure to include the name of any new tests, including this one, in the Makefile. If you don't, they won't be tested when you run "make part1".
- 7/5 3:45 PM: There was a bug in a comment in types.h that talked about "j-type" when it was supposed to say "u-type"
- 7/6: 9:54 AM: Part 2 files now released!
- 7/9: 3:35 PM: There was a bug in the Makefile that wouldn't allow you to test your part2 given certain conditions. Please pull the update. It may cause a merge conflict if you have modified the Makefile, so refer to Lab 0 on how to resolve these.
- 7/9: 9:33 PM: In utils.c, I added an exit(-1) statement that way there is consistency among how invalid instructions are treated. Please note that Error 255 means that you're hitting an exit(-1) statement. Again, might cause merge conflicts, so handle those appropriately.

Goals

We hope this project will enhance your C programming skills, deepen your understanding of RISC-V instructions and their formats, 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.

The RISC-V green card provides some information necessary for completing this project. You may also find this version helpful as well: Alternate green card.

Getting started

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

To obtain the proj2 files, pull from the skeleton git repo. As always, we recommend using a private BitBucket repo to track your changes. For the suggested code blow, we assume you have named your repository "proj2-xxx", where the xxx is your 3 letter login. You can set up your workflow by doing the following in a workspace directory of your choice:

```
$ git clone https://<mybitbucketusername>@bitbucket.org/<mybitbucketusername>/proj2-xxx.git
$ cd proj2-xxx
$ git remote add proj2-starter https://github.com/61c-teach/su18-proj2-starter.git
$ git fetch proj2-starter
$ git merge proj2-starter/master -m "merge proj2 skeleton code"
```

.

The files you will need to modify and submit are:

- `part1.c`: The main file which you will modify for part 1.
- `utils.c`: The helper file which will hold various helper functions for part 1.
- `part2.c`: The main file which you will modify for part 2. You will not be submitting this file for part 1.

You will NOT be submitting any files other than the ones listed above. This means that you will not be submitting header files. If you add helper functions, please place the function prototypes in the corresponding C files. If you do not follow this step, your code will likely not compile and you will get a zero on the project.

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.
- `utils.h`: File that contains the format for instructions to print for part 1.
- `test_utils.c`: File with some tests for the helper functions in utils.c.

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.

Your code will be tested (via our autograder) on the hive machines. BEFORE YOU SUBMIT, please make sure your code is functioning on a hive machine as opposed to just your local machine.

The RISC-V Emulator

The files provided in the start kit comprise a framework for a RISC-V emulator. In part 1, you will add code to `part1.c` and `utils.c` to print out the human-readable disassembly corresponding to the instruction's machine code. In part 2, you'll complete the program by adding code to `part2.c` to execute each instruction (including performing memory accesses). Your simulator must be able to handle the machine code versions of the RISC-V machine instructions below. 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. Note that symbols such as 12'b0 corresponds to <size>'<base format><number>. So 12'b0 is Verilog's way of representing a length 12 number, represented in binary, with the value 0.

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
sltu rd, rs1, rs2			0x3	0x00	R[rd] ← (R[rs1] < R[rs2]) ? 1 : 0 // unsigned comparison
mulhu rd, rs1, rs2			0x3	0x01	R[rd] ← (R[rs1] * R[rs2])[63:32] // unsigned operation
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]
divu rd, rs1, rs2			0x5	0x01	R[rd] ← R[rs1] / R[rs2] // unsigned operation
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]
remu rd, rs1, rs2			0x7	0x01	R[rd] ← R[rs1] % R[rs2] // unsigned operation
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] ← {24'b0, *Mem*(R[rs1] + offset, byte)}
lhu rd, offset(rs1)			0x5		R[rd] ← {16'b0, *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

INSTRUCTION	TYPE	OPCODE	FUNCT3	FUNCT7/IMM	OPERATION
sltiu rd, rs1, imm			0x3		R[rd] ← (R[rs1] < imm) ? 1 : 0 // unsigned comparison
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 = 10 is exit or end of code indicator.
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}
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(R[rs1] < R[rs2]) // unsigned comparison
					PC ← PC + {offset, 1b'0}

bgeu rs1, rs2, offset			0x7		if(R[rs1] >= R[rs2]) // unsigned comparison
					PC ← PC + {offset, 1b'0}

auipc rd, imm	U	0x17			R[rd] ← PC + {imm, 12'b0}
lui rd, imm		0x37			R[rd] ← {imm, 12'b0}
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. Look at P&H (4th edition) page B-43 for further information on endianness (byte order).

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 0 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()`

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

Part 1: Disassembler (Due 7/6 at 11:59 PM)

In part 1, you will be writing a disassembler that translates RISC–V machine code into human–readable assembly code. You will also be laying the building blocks for the actual emulator that you will implement in part 2. You will be implementing the following functions. The files in which they are located are in parentheses.

1. `decode_instruction(instruction_bits)` (part1.c): Begins to break down instruction bits by looking at opcode. You need to filter out the relevant instruction bits that will tell you all the information needed to eventually use the appropriate `print_x` function, and `decode_instruction` is the "top level" of that filter.
2. `parse_instruction(instruction_bits)` (utils.c): Uses the given instruction (encoded as a 32–bit integer) and returns the corresponding instruction. You will have to determine the proper format of the given instruction and use the correspdonding member of the instruction struct. You will find the green sheet particularly helpful here.
3. `sign_extend_number(field, n)` (utils.c): This function interprets the number in `field` as an `n`–bit number and sign–extends it to a 32–bit integer.
4. `get_*_offset(instruction)` (utils.c): For the corresponding instruction type (either branch, jump, or store), this function unpacks the immediate value and returns the number of bytes to offset by. In this case of branches and jumps, these functions should return the number of bytes to add to the PC to get to the desired label. In the case of stores, the corresponding function will return the offset on the destination address.
5. `print_*(instruction)` (part1.c): Prints the instruction to standard output. You should use the constants defined in the file `utils.h` to properly format your instructions. Failure to do so will cause issues with the autograder. You should also refer to registers by their numbers and not their names.
6. `write_*(instruction)` (part1.c): Parses the instruction further to gather the information necessary to call the correct `print_x` function.

Testing

There are two types of tests for this project: unit tests and end–to–end tests. The unit tests can be found in the file `test_utils.c`. This suite tests the `sign_extend_number` and `parse_instruction` functions (we recommend making your own offset tests). You can run these tests using the command below.

```
$ make test-utils
```

To create your own unit tests, you should create a new function in the `test_utils.c` that contains your logic. You then must add your test function to the test suite located in the `main` function. You can look at the other tests in the file as examples. The unit tests use the CUnit framework. **It is more than likely that CUnit is not configured on your local machine, so you should only run these tests on the hive machines.**

To test part1 in its entirety, you can run the full disassembly end-to-end tests. 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!
unsigned_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 end-to-end tests, you first need to create the relevant machine code. This can either be done by hand or by using the Venus simulator built by Keyhan Vakil (currently a CS161 TA). 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 in `[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 7 of the Makefile, where it says `ASM_TESTS`, add `[test_name]` to the list with spaces in between file names.

In addition, a current tutor, Stephan Kaminsky, upgraded Venus and added some functionality to make creating your own tests easier. His simulator is similar to Venus, but with more options. You may find the Trace Generator particularly useful for this part, but make sure to have the "Proj2 Override" option enabled.

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` and `utils.c` will be considered by the autograder. To submit, enter in the command from within the hive.

If you want to get rid of any junk files that may result from compilation and testing, run `make clean`.

Debugging and GDB

The best way to debug your project will be to use GDB. To run GDB, you should use the following command.

```
$ cgdb riscv
```

Once inside GDB, you should use the following command to run your code. The possible test files are called `simple`, `multiply`, `random`, `unsigned`.

```
$ run -d riscvcode/code/[test-name].input
```

Submission

To submit your project, run the command on the hive machines. **You will only be submitting your `part1.c` and `utils.c` files.** Furthermore, before you submit, please ensure that your code works on the hive machines, as this is where we will be running the autograder.

```
$ submit proj2-1
```

In addition, don't forget to tag your submission commit on BitBucket. You can do this by running the following.

```
$  git tag -f "proj2-1-sub"
$  git push origin --tags
```

If you have already submitted and forgot to tag, don't fret! You can go back and tag your submission commit without loss of credit. Just be sure to do that ASAP. You can tag a previous commit by running "git log", getting the hash of your commit, which should look like a string of random letters and numbers, and running the following.

```
$  git tag -f "proj2-1-sub" <hash of commit here>
$  git push origin --tags
```

Part 2: Executor (Due on 7/13 at 11:59 PM)

For part 2 of this project, you will be implementing the actual emulator that can execute RISC-V machine code. In order to accomplish this task, you will need to complete the functions below.

- `execute_*()`: These functions handle the majority of the execution of the reduced RISC-V instruction set we are implementing in this project. To reduce the amount of busy work, the ladders of switch statements are already implemented for you. To complete the implementation, you will only need to fill in the switch cases. You should be updating the appropriate register values and interacting with memory when applicable.
- `store()`: This function takes an address, a size, and a value and stores the first –size– bytes of the given value at the given address.
- `load()`: This function accepts an address and a size and returns the next –size– bytes starting at the given address. **You will need to implement `load()` first in order to fetch instructions properly; without a functioning `load()` you will get errors due to invalid instructions.**

Note that a correct implementation of this part will depend on the functions in `utils.c`. Thus, you should ensure that these functions (which you wrote in part 1) are working correctly.

Fetching Updates

There are a few updates that have been made to the starter code for part 2. To integrate these changes, run the following command.

```
$ git pull [name-of-starter-repo] master
```

To find the name of the starter repo, you should run the command below and find the remote name that corresponds to the url `https://github.com/61c-teach/sp18-proj2-starter`

```
$ git remote -v
```

If there is no such remote repository, then you will need to add the starter code as a remote. To accomplish this, run the command below. This command gives the remote repository the name `proj2-starter`

```
$ git remote add proj2-starter https://github.com/61c-teach/sp18-proj2-starter.git
```

If you have been pulling updates without any problems, simply pull the most recent update to get the part 2 files.

Testing

You are provided with three test files that evaluate your emulator's correctness. The input files are the same as those in part 1. The output files are those with the `.trace` extension. The solutions are located in the folder `riscvcode/ref`, while your project's code will write it's output to the folder `riscvcode/out`. These trace files dump the contents of the registers after the execution of each instruction.

To run your code on all of the tests, you should run the command below.

```
$ make part2
```

If you would like to run a specific test, then run the following command. You should replace `[test_name]` with either `simple`, `random`, or `multiply`.

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

The testing suite is run through the python script `part2_tester.py`. This file will read in your output and compare it against the reference trace by looking at changes in the registers between instructions. If a test fails, you should see an output that lists which instruction and register was incorrect. You can then cross-reference this information with the tests' `.solution` file (the disassembled version from part 1) to find the erroneous error.

If you would like to add new tests, you should follow the same steps as those outlined when creating tests for part 1. The only difference is that you need to create a `.trace` reference file that contains the register values after every instruction

To test through GDB, run

```
$ make riscv
$ cgdb riscv
$ *set breakpoint*
$ run -r riscvcode/code/[test_name].input
```

The –r flag will print out the registers values at each step. To run the execution interactively (which will help when pinpointing error messages), run

```
$ make riscv
$ ./riscv -ri riscvcode/code/[test_name].input
```

Submission

For this part, you will only be submitting the files `part2.c` and `utils.c`. If you modify any other files (including header files), you will likely get a zero on the project. Any function prototypes you add should be placed at the top of the corresponding C file (and not the header file).

As always, your code will be tested (via our autograder) on the hive machines. BEFORE YOU SUBMIT, please make sure your code is functioning on a hive machine as opposed to just your local machine.

When you are ready, you can submit your project on the hive machines via glookup using the following command.

```
$ submit proj2-2
```

Again, don't forget to tag your submission commit on BitBucket. You can do this by running the following.

```
$  git tag -f "proj2-2-sub"
$  git push origin --tags
```

If you have already submitted and forgot to tag, don't fret! You can go back and tag your submission commit without loss of credit. Just be sure to do that ASAP. You can tag a previous commit by running "git log", getting the hash of your commit, which should look like a string of random letters and numbers, and running the following.

```
$  git tag -f "proj2-2-sub" <hash of commit here>
$  git push origin --tags
```

To check that your submission was successful, you should see a submission timestamp when you run the command below. Note that we will grade your most recent submission.

```
$ glookup -t
```