

# Project 3: RISC-V Instruction Set Emulator

## Goals

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 the [VENUS](#) simulator!

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. To obtain the project files, you need to download a zip file from the link below:

[https://drive.google.com/file/d/1xsm\\_vT2ovCOBL8wTNHc\\_HjRA\\_rmYR5VU/view?usp=sharing](https://drive.google.com/file/d/1xsm_vT2ovCOBL8wTNHc_HjRA_rmYR5VU/view?usp=sharing)

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.

In those files you will see comments such as `/* YOUR CODE HERE */` where you should place your modifications.

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** 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**.

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). BEFORE YOU SUBMIT, please make sure your code is functioning on your local machine, preferably on Ubuntu 64-bit Linux.

## 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 any C reference or Chapter 6 of K&R (C programming textbook), 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 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
<code>add rd, rs1, rs2</code>	R	0x33	0x0	0x00	$R[rd] \leftarrow R[rs1] + R[rs2]$
<code>mul rd, rs1, rs2</code>			0x0	0x01	$R[rd] \leftarrow (R[rs1] * R[rs2])[31:0]$
<code>sub rd, rs1, rs2</code>			0x0	0x20	$R[rd] \leftarrow R[rs1] - R[rs2]$
<code>sll rd, rs1, rs2</code>			0x1	0x00	$R[rd] \leftarrow R[rs1] \ll R[rs2]$
<code>mulh rd, rs1, rs2</code>			0x1	0x01	$R[rd] \leftarrow (R[rs1] * R[rs2])[63:32]$
<code>slt rd, rs1, rs2</code>			0x2	0x00	$R[rd] \leftarrow (R[rs1] < R[rs2]) ? 1 : 0$
<code>xor rd, rs1, rs2</code>			0x4	0x00	$R[rd] \leftarrow R[rs1] \wedge R[rs2]$
<code>div rd, rs1, rs2</code>			0x4	0x01	$R[rd] \leftarrow R[rs1] / R[rs2]$
<code>srl rd, rs1, rs2</code>			0x5	0x00	$R[rd] \leftarrow R[rs1] \gg R[rs2]$
<code>sra rd, rs1, rs2</code>			0x5	0x20	$R[rd] \leftarrow R[rs1] \ggg R[rs2]$
<code>or rd, rs1, rs2</code>			0x6	0x00	$R[rd] \leftarrow R[rs1] \vee R[rs2]$
<code>rem rd, rs1, rs2</code>			0x6	0x01	$R[rd] \leftarrow (R[rs1] \% R[rs2])$
<code>and rd, rs1, rs2</code>			0x7	0x00	$R[rd] \leftarrow R[rs1] \& R[rs2]$
<code>lb rd, offset(rs1)</code>	I	0x03	0x0		$R[rd] \leftarrow \text{SignExt}(\text{Mem}(R[rs1] + \text{offset}, \text{byte}))$

INSTRUCTION	TYPE	OPCODE	FUNCT3	FUNCT7 / IMM	OPERATION
<code>lh rd, offset(rs1)</code>			0x1		$R[rd] \leftarrow \text{SignExt}(\text{Mem}(R[rs1] + \text{offset}, \text{half}))$
<code>lw rd, offset(rs1)</code>			0x2		$R[rd] \leftarrow \text{Mem}(R[rs1] + \text{offset}, \text{word})$
<code>addi rd, rs1, imm</code>		0x13	0x0		$R[rd] \leftarrow R[rs1] + \text{imm}$
<code>slli rd, rs1, imm</code>			0x1	0x00	$R[rd] \leftarrow R[rs1] \ll \text{imm}$
<code>slti rd, rs1, imm</code>			0x2		$R[rd] \leftarrow (R[rs1] < \text{imm}) ? 1 : 0$
<code>xori rd, rs1, imm</code>			0x4		$R[rd] \leftarrow R[rs1] \wedge \text{imm}$
<code>srli rd, rs1, imm</code>			0x5	0x00	$R[rd] \leftarrow R[rs1] \gg \text{imm}$
<code>srai rd, rs1, imm</code>			0x5	0x20	$R[rd] \leftarrow R[rs1] \gg \text{imm}$
<code>ori rd, rs1, imm</code>			0x6		$R[rd] \leftarrow R[rs1] \mid \text{imm}$
<code>andi rd, rs1, imm</code>			0x7		$R[rd] \leftarrow R[rs1] \& \text{imm}$
<code>ecall</code>		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.
<code>sb rs2, offset(rs1)</code>	S	0x23	0x0		$\text{Mem}(R[rs1] + \text{offset}) \leftarrow R[rs2][7:0]$
<code>sh rs2, offset(rs1)</code>			0x1		$\text{Mem}(R[rs1] + \text{offset}) \leftarrow R[rs2][15:0]$
<code>sw rs2, offset(rs1)</code>			0x2		$\text{Mem}(R[rs1] + \text{offset}) \leftarrow R[rs2]$
<code>beq rs1, rs2, offset</code>	SB	0x63	0x0		if( $R[rs1] == R[rs2]$ ) $PC \leftarrow PC + \{\text{offset}, 1b'0\}$
<code>bne rs1, rs2, offset</code>			0x1		if( $R[rs1] != R[rs2]$ ) $PC \leftarrow PC + \{\text{offset}, 1b'0\}$
<code>lui rd, offset</code>	U	0x37			$R[rd] \leftarrow \{\text{offset}, 12b'0\}$
<code>jal rd, imm</code>	UJ	0x6f			$R[rd] \leftarrow PC + 4$ $PC \leftarrow PC + \{\text{imm}, 1b'0\}$

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. See lecture notes 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 disassembly 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, they should handle all 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 function `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-----
```

## Testing

The tests provided do not test every single possibility, 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`. In the `Makefile`, where it says `ASM_TESTS`, add `[test_name]` to the list with spaces in between file names.

To run your code through `gdb`, you can compile your code using `make riscv`. Then you can run the debugger on the `riscv` executable. You will need to 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 of the project, only changes to the files `part1.c` and `utils.c` will be considered by the autograder. To submit, upload **these two files** on Gradescope ([www.gradescope.com](http://www.gradescope.com)).

## 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 update 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 parameter 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()`.

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. Note that, for the final results, only changes to the files `part1.c`, `utils.c`, and `part2.c` will be considered by the autograder. To submit your final solution, upload **these three files** on Gradescope ([www.gradescope.com](http://www.gradescope.com)).

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 Makefile.

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

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