

**Due date:** Thursday February 20th 11:59:59pm ET.

**Points:** This lab is worth 8 points (out of 200 points in 6.004).

**Getting started:** Log in to Athena. If you haven't done the initial setup, follow the Athena github setup instructions at [https://6004.mit.edu/web/\\_static/spring20/resources/references/athena-github-setup.pdf](https://6004.mit.edu/web/_static/spring20/resources/references/athena-github-setup.pdf). When connecting to Athena, use display forwarding (-X option) so you can use the graphical tools for this lab. If (-X option) doesn't work for you, then try (-Y option) instead.

```
ssh -X {YourMITUsername}@athena.dialup.mit.edu
```

To create your initial Lab 1 repository, please visit <https://6004.mit.edu/web/spring20/user/labs/lab1>. All files you need for the lab (except this handout) will be placed in your repository. Once it has been created, you can clone it anywhere in your Athena locker by running:

```
git clone git@github.mit.edu:6004-spring20/labs-lab1-{YourMITUsername}.git lab1
```

**Turning in the lab:** To turn in this lab, commit and push the changes you made to your git repository. **After pushing, check the course website (<https://6004.mit.edu>, Labs/Didit tab) to verify that your submission was correctly pushed and passes all mandatory tests.** If you finish the lab in time but forget to push, you will incur the standard late submission penalties.

**Check-off meeting:** After turning in this lab, you must go to the lab for a check-off meeting by Wednesday, February 26th. Checkoffs for Lab 1 will begin on Friday, February 14th. You are **strongly** encouraged to sign up for a scheduled checkoff, walk-ins will receive lowest priority in the lab. See the course website for lab hours.

## Introduction

In this lab you will learn how to write low-level assembly code. Specifically, instead of using high-level programming languages such as Python or C, you will write several programs in **RISC-V** assembly code. To evaluate its correctness, your code will be assembled and linked to testbenches using a RISC-V compiler, and the generated test programs will be executed by a simulated 32-bit RISC-V machine.

All discussion questions asked throughout this lab are fair game to be asked during your checkoff meeting. Place your answers in the provided **discussion\_questions.txt** file so that they will be available during your checkoff. You should also be prepared to walk-through and explain any of the code we ask you to write. When you have completed the lab, commit all your changes to the repository and push them.

**To be able to checkoff this lab you must do both exercises, answer the discussion questions and PASS all the tests in Didit.** Note: In this and future labs, required exercises are outlined with a solid line (like this paragraph) while optional exercises are outlined with a dashed line.

Most of this course is dedicated to designing general-purpose, programmable computers—machines that can stream cat videos, solve differential equations, or display the front page of reddit. But before we begin to design the programmable hardware, we will learn about the interface we will use to program it.

In 6.004, we'll use an architecture<sup>1</sup> called **RISC-V**. There are many variants of **RISC-V**, but we'll focus on the simplest one, called **RV32**. **RV32** has one main data type—a 32-bit number called a *word*. **RV32** programs can only store up to 32 words at a time in so-called *registers*. If more storage is needed, the program has to load words from main memory into registers and store them back in main memory after modifying them. Besides loading and storing words, programs can do some basic arithmetic operations (such as addition and subtraction) and branching (to implement loops and if-then statements), but that's it!

<sup>1</sup>An architecture, or more specifically an *instruction set architecture*, is the interface between software and hardware. The instruction set architecture precisely specifies the resources of the machine and the way programs can access and use them.

Let’s walk through an example program. Three equivalent representations are shown below; Python is on the left, **RISC-V** in human-readable form is in the middle, and **RISC-V** in binary is on the right.

Python	RISC-V ( <i>assembly</i> )	RISC-V ( <i>binary</i> )
<code>n = 9</code>	<code>li a0, 9</code>	00: 0000000000000001001010100110111
<code># check if n is even</code>	<code>// check if n is even</code>	
<code>if n % 2 == 0:</code>	<code>If:</code>	
<code>n = n / 2</code>	<code>andi a1, a0, 1</code>	04: 00000000000101010111010110010011
<code>else:</code>	<code>bnez a1, Else</code>	08: 0000000000001011001001101100011
<code>n = 3*n + 1</code>	Then:	
	<code>srai a0, a0, 1</code>	0C: 01000000000101010101010100010011
	<code>j End</code>	10: 0000000010000000000000001101111
	Else:	
	<code>slli a2, a0, 1</code>	14: 00000000000101010001011000010011
	<code>add a0, a2, a0</code>	18: 00000000110001010000010100110011
	<code>addi a0, a0, 1</code>	1C: 00000000000101010000010100010011
	<code>End:</code>	

A lot is going on here! Let’s go through it line by line.

`li a0, 9` The opcode `li` stands for *Load Immediate*. This is actually a *pseudo-instruction*; it is shorthand for `addi a0, zero, 9`. The register `zero` always contains the value zero. “Immediate” is just another term for “constant”, in this case 9, and `a0` is the name of the eleventh register, typically used to store the arguments of a function.<sup>2</sup> In English, this pseudo-instruction means “load the constant 9 into register `a0`” — so we’ll be using `a0` the way we use `n` in Python.

`If:` This is a label. It doesn’t generate any instructions, but lets you name a specific place in the program.

`andi a1, a0, 1` The opcode `andi` stands for *bitwise-AND with an Immediate*. In English, this instruction means “take the bitwise-AND of [the contents of `a0`] and [the constant value 1], and place the result in register `a1`.” The immediate 1 is really being represented as a word (32 bits) inside the computer, so we are actually AND-ing with `0...01`. The result that we place in `a1` will be 1 if the number in `a0` is odd, and 0 if the number in `a0` is even.

**Discussion Question 1:** Why does AND-ing with `0x00000001` tell us if the value is even or odd?

To keep the words from getting tedious, from now on we’ll use a right arrow  $\Rightarrow$  when we’re describing the effect of an instruction, and the running prose will just be a commentary.

<sup>2</sup>The thirty-two registers are numbered 0–31, so we could have referred to `a0` by writing `x10` instead.

**bnez a1, Else** The opcode **bnez** stands for *Branch if Not Equal to Zero*. This is also a pseudo-instruction for **bne a1, zero, Else**. It can cause the computer to skip ahead (or back) to a different part of the program. Several instruction types do this; the term *branch* is used when the skip is conditional. The term *jump* is used if the skip is unconditional.

⇒ CONTINUE to Then if the number in **a0** is even (**a1** = 0)

⇒ GO TO Else if the number in **a0** is odd (**a1** = 1)

Then: This is just another label.

**srai a0, a0, 1** *Arithmetic right shift by immediate*. An arithmetic shift by 1 turns the binary number **abc...xyz** into **aab...wxy** by duplicating the most-significant bit (MSB) **a** and dropping the least-significant bit (LSB) **z**, whereas a *logical* shift would turn the same number into **0ab...wxy**.<sup>3</sup>

⇒ DIVIDE the number in **a0** by 2 and place the result in **a0**

**j End** ⇒ GO TO End

Else:

**slli a2, a0, 1** *Logical left shift by immediate*.

⇒ MULTIPLY the number in **a0** by 2 and place the result in **a2**

**add a0, a2, a0** ⇒ ADD the number in **a0** to the number in **a2** ( $= 2 \times \mathbf{a0}$ ) and place the result ( $= 3 \times \mathbf{a0}$ ) back in **a0**

**addi a0, a0, 1** ⇒ ADD 1 to the number in **a0** and place the result in **a0**

End: We're done! The register **a0** now contains  $\mathbf{a0} \div 2$  if **a0** was even and  $\mathbf{a0} \times 3 + 1$  if **a0** was odd.

The Collatz conjecture states that for any  $n_0$ , the sequence  $n_0, n_1, n_2, \dots$  eventually reaches 1 where each  $n_i$  after  $i = 0$  is defined as

$$n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{if } n_{i-1} \text{ is even} \\ 3n_{i-1} + 1 & \text{if } n_{i-1} \text{ is odd} \end{cases}$$

**Discussion Question 2:** How could you modify this program so that it tests the Collatz conjecture? In Python, the Collatz test might look something like this:

```
n = 9
while n != 1:
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3*n + 1
```

<sup>3</sup>When the contents of a register represent a signed binary number, arithmetic right shifts preserve the sign of the number, whereas logical right shifts do not. Since numbers are represented in binary, shifting the digits right by one has the effect of dividing by 2.

## Running and Viewing RISC-V Programs

In this section we’re going to introduce tools that help us run and debug our assembly.

The lab repository contains a copy of the code from page 2 (in a file called **collatz.S**). Also in the repo is **assemble-collatz.sh**, which will first assemble the program into binary and then generate a human-readable version. Technically, **assemble-collatz.sh** doesn’t generate an executable binary file, but something similar. One more step would turn it into something you could actually run on a **RISC-V** processor. To run it, open up the terminal and navigate to the lab repo; then enter the three following commands (only enter the black text after the **\$** prompt, the text before it should be similar to what you see):

```
student@some-dialup.mit.edu:~$ cd lab1
student@some-dialup.mit.edu:~/lab1$ ./assemble-collatz.sh
student@some-dialup.mit.edu:~/lab1$ cat collatz.dump
```

At the top of the output (just beneath “Contents of section .text.init”) is the hex representation of the **RISC-V** instructions in binary format. Under that (beneath “Disassembly of section .text.init”) is an instruction-by-instruction listing of the program, created by translating the binary above back into human-readable format.

```
00000000 <collatz>:
0:  00900513          li      a0,9

00000004 <If>:
4:  00157593          andi    a1,a0,1
8:  00059663          bnez    a1,14 <Else>
```

On the left (0:) is the address of each instruction. Beside that (00900513) is the hex representation of the instruction itself. On the right (li a0, 9) is the assembly. Notice that the addresses are numbered by fours (0, 4, 8, C, 10, 14, ...). **RISC-V** uses byte addresses and since each instruction is 32 bits, it takes four bytes (8 bits each) to store it. Therefore, successive instructions are four bytes apart.

Fundamentally, all programs that your computer will ever run look like this. They may start at an address other than 0, they might get moved around if they’re included inside another program, and they may even be interrupted and later resumed. But your computer really does work by only doing many (many, many) small operations like these!

For the rest of 6.004, we’ll be using test suites written by the staff, for two purposes. First, tests help us verify that our programs match the specification—in other words, that they actually do what we think they do. Second, the tests are a method for the staff to prove that you completed the lab (to complement the less exhaustive but more conceptual checkoffs).

The test framework will wrap each of your programs with extra code. If you take a look at a **.dump** file after assembling your programs, you’ll see the extra instructions (with your work sandwiched in the middle). To generate the files necessary for testing your programs, simply run **make**, like so:

```
student@some-dialup.mit.edu:~/lab1$ make
```

Anytime you change your code, be sure to re-run **make** before testing.

We provide you with a **RISC-V** simulator for testing and debugging your code.

To get **collatz** to work with the simulator, you'll need to make some small tweaks. Fire up your editor of choice and delete the line containing **li a0,9** (but not the line containing the label **collatz**!) Then, add the instruction **ret** after the label **End**. Be sure to run **make**!

To start the simulator, run **rv\_sim\_gui**, like so:

```
student@some-dialup.mit.edu:~/lab1$ rv_sim_gui &
```

**Note:** The **&** at the end is optional but causes the simulator to be “detached” from your terminal so that you get a prompt back and can run additional commands.

**Note:** If you get an error saying something like `_tkinter.TclError: no display name and no $DISPLAY environment variable` when running this, that means your graphical tools aren't setup correctly. Check that you followed the setup instructions linked at the start of this handout and supplied the **-X** flag to **ssh**.

On the left under Program select *collatz*, and under Test Case select test 1. Then, at the top, press Load Program.

**Note:** To select a different program or a different test, you must first press Stop Program. If you modify a program, you must re-run **make** and then re-load the program in the simulator.

If you press Run, you should see:

```
$ Load program collatz with test 1
$ Run:
10:Passed
Lab1.collatz1: PASSED
```

This is great news! Except it's not very informative. If our program encountered an error or had a bug in it, all we would see is **FAILED**. We want something that gives us a better idea of what's going on.

Hit Exit Program and then Load Program to reset the simulator. This time, press Step a few times. As you press Step, you should see *PC = 0x...* change and *Instrs executed = ...* begin to increment. “PC” stands for *Program Counter*; it is the address of the instruction that the simulator is about to execute. The first four dozen or so steps are setup for the test framework, but if you pressed Step four dozen times you would reach the code for **collatz**. That would be very tedious, so there's an additional feature that lets you run the simulation until PC reaches particular addresses. These are called *break points*.

*You should Exit and Load again before adding break points.*

Open up **collatz.dump** and locate the first instruction from **collatz.S** (try searching for the symbol `<collatz>`). In the simulator, enter the address of this instruction under Break Points (at the bottom left) as a hexadecimal number with a **0x-** prefix, and then press Apply.

Now when you hit Run, the simulator will automatically step until PC equals the address you just entered; from this point you can step manually. The default tab (Console Output) is not particularly useful since our programs don't print anything yet. Instead, select Registers and note the current values of **a0**, **a1**, and **a2**.

With the simulator and source (or disassembly) open side-by-side, step through **collatz**. Before each step, predict how **a0**, **a1**, **a2**, and PC will change.

**Discussion Question 3:** In test 1, does the simulation branch to `<Else>` (the case when **a0** is initially odd) or continue on to `<Then>` (the case when **a0** is initially even)? How about in test 2?

## Main Memory

In this section we're going to (A) learn how to access main memory, and (B) get more practice by finishing another program. (After that, you'll write your first complete **RISC-V** program!)

At the top of the next page is a Python function designed to find the largest word in an array. An *array* is a collection of elements (such as bytes, words, double words, *etc.*) that are evenly spaced. As a result, the location of the  $n$ th element is always

$$\text{array starting address} + (n \times \text{constant}).$$

For an array of words, the  $n$ -th word in the array is located at (start of array +  $4 \times n$ ), since **RISC-V** memory is indexed by byte and each word is four bytes long.

The function takes two arguments:  $p$ , which is the start of the array, traditionally called the *pointer* to the array; and  $n$ , which is the number of elements in the array. To make our Python function a little more like assembly (so that it's easier to translate), we'll imagine that we have memory arranged just like it is in **RISC-V** (as a really, really long sequence of bytes).

```
Memory = ... # lots and lots of bytes
```

```
def load_word(addr):
    return int(Memory[addr:addr+4])
```

```
// . . .
```

Idiomatic Python

Alternate Version (closer to **RISC-V**)

```
def maximum(p, n):
    largest_so_far = 0
    for i in range(n):
        w = load_word(p + 4*i)
        if w > largest_so_far:
            largest_so_far = w
    return largest_so_far
```

```
def maximum(p, n):
    largest_so_far = 0
    while n != 0:
        w = load_word(p)
        if w > largest_so_far:
            largest_so_far = w
        p = p + 4
        n = n - 1
    return largest_so_far
```

Our assembly version of this program will be subject to these constraints:

- When our function is called, the argument  $p$  (the starting address of the array) will be stored in **a0** and the argument  $n$  (the size of the array) will be stored in **a1**.
- Our function should store its result (the largest word in the array) in **a0** and end with a **ret** instruction.
- Our function should only use the **a0-a7** registers.

One good recipe for translating Python code into **RISC-V** is the following four-step process. You'll be responsible for most of step 2:

- Decide which registers will be used for which variables. It was stipulated that  $p \leftrightarrow \mathbf{a0}$  and  $n \leftrightarrow \mathbf{a1}$ . Let's (arbitrarily) decide that  $\text{largest\_so\_far} \leftrightarrow \mathbf{a2}$  (which we'll copy to **a0** when we're done) and  $w \leftrightarrow \mathbf{a3}$ .

2. Convert variable assignments and mathematical operations into register-register arithmetic and load/store instructions.
3. Convert if/else statements into branches.
4. Convert loops (for, while, etc.) into branches and jumps.

There are several ways to implement loops; the following examples on this page demonstrate two ways of doing step 4. The ellipses (...) indicate an instruction that you will eventually fill in.

#### Condition at Bottom

```
maximum:
    li    a2, 0           // load immediate 0 into a2 (largest_so_far)
loop:
    lw    a3, 0(a0)       // load word at [[address stored in a0] + 0] into a3 (w)
    ble   a3, a2, continue // skip next instruction if a3 (w) ≤ a2 (largest_so_far)
    ...
continue:
    ...
    ...
    bnez  a1, loop        // go to top of loop if a1 (n) ≠ 0
done:
    ...
    ...
```

#### Condition at Top

```
maximum:
    li    a2, 0           // load immediate 0 into a2 (largest_so_far)
loop:
    beqz  a1, done        // go to <done> if a1 (n) = 0
    lw    a3, 0(a0)       // load word at [[address stored in a0] + 0] into a3 (w)
    ble   a3, a2, continue // skip next instruction if a3 (w) ≤ a2 (largest_so_far)
    ...
continue:
    ...
    ...
    j     loop            // go to top of loop
done:
    ...
    ...
```

The really new feature here is the `lw a3, 0(a0)` instruction. The opcode `lw` stands for *Load Word*. It gives us a way to move data from main memory (which is spacious but slow) to a register, so that we can do arithmetic with it. It works like this:

`lw rd, imm(rs)`      Suppose the content of the register `rs` is some number  $n$ . Then the word at address  $(n + \text{imm})$  in memory will be loaded into register `rd`. For example, if the value of `rs` was `0x00040100` and `imm` was 8, then the word located at address `0x00040108` will be put into `rd`.

To put a word *back* into memory, we can use the `sw` (*Store Word*) instruction, which works similarly.

`sw rs2, imm(rs1)` Suppose the content of the register `rs1` is some number  $n$ . Then the word in register `rs2` will be copied into memory at address  $(n + \text{imm})$ .

If you open up `maximum.S`, you'll notice that the staff have already written the load instruction for you. This gives you an opportunity to see it before you use it; in the graded exercises below, you'll have to write your own loads and stores.

**Exercise 1:** Complete `maximum.S` and get it to pass all of the tests in the simulator. Although this code is not tested by Didit or assigned points, we may ask you to go over it during checkoff.

## Triangular Numbers

The triangular numbers count the number of objects in arrangements that look like triangles. The first few are 1, 3, 6, and 10.



If you open up `triangular.S`, you'll find a Python implementation of a function that computes the  $n$ th triangular number given  $n$ . Your task is to translate this function into assembly. Your version will be subject to these constraints:

- A. When the function is called, the argument  $n$  will be stored in `a0`.
- B. The function should store its result in `a0` and end with a `ret` instruction.
- C. The function should only use the `a0–a7` registers.

**Exercise 2 (30%):** Complete `triangular.S` and get it to pass all of the tests in the simulator.



## Bubblesort

Bubblesort—so named because smaller elements float up to the top of the array over time if you visualize it—is a relatively simple algorithm that works by swapping adjacent elements until everything is in order. The file **bubblesort.S** contains a template for bubble sort; inside is a blank function called **sort**.

**Exercise 3 (70%):** Implement the bubblesort function in **RV32** assembly, subject to the following constraints:

1. The array to sort is stored in memory in a contiguous range of addresses, and is passed to **sort** using two arguments. When **sort** is called, the starting address of the array will be stored in **a0** and the number of elements in the array will be stored in **a1**.
2. The **sort** function should modify the array directly, and need not return any value.
3. The **sort** function should only use the **a0-a7** registers.
4. Your implementation should only use the subset of **RV32 I** that we use in this course. This includes all its instructions except sub-word loads and stores (**lb**, **lbu**, **lh**, **lhu**, **lb**, **sb**, **sh**) and **auiop**. *The simulator only supports this subset. Using an instruction outside this subset will cause an illegal instruction exception.*

The file **bubblesort.S** also contains Python and C implementations of the Bubble sort algorithm to serve as the basis for your work. As before, assemble your code by running **make** and execute the generated binary by running **rv\_sim\_gui**. Alternatively, you can run the tests by entering:

```
student@some-dialup.mit.edu:~/lab1$ rv_sim bubblesort
```

To run tests one a time, enter:

```
student@some-dialup.mit.edu:~/lab1$ rv_sim bubblesort k
```

where *k* is 1, 2, 3, 4, or 5. For more information on the command line interface to the simulator, see the appendices.

**Discussion Question 4:** How many instructions did you use to perform the swap of **p[i]** and **p[i+1]**? Can you do it with just two?

# 1 Appendix: RISC-V ISA Reference

**Notes:**

- For this lab, you should only use the subset of RV32I instructions presented in this appendix. Using unsupported instructions (sub-word loads and stores and AUIPC) will cause errors when you try to simulate the program (using `rv_sim` or `rv_sim_gui`). You are encouraged to use pseudo-instructions to simplify your code.
- Follow the RISC-V calling convention when writing your code. The test program that your code will be linked to follows the RISC-V calling convention. If you overwrite registers that the test program uses (e.g., using a callee-saved register without saving and restoring it as the convention mandates), you may cause it to misbehave, even if there is no other error within your code.

## MIT 6.004 ISA Reference Card: Instructions

Instruction	Syntax	Description	Execution
LUI	<b>lui</b> <i>rd</i> , <i>immU</i>	Load Upper Immediate	$\text{reg}[rd] \leftarrow \text{immU} \ll 12$
JAL	<b>jal</b> <i>rd</i> , <i>immJ</i>	Jump and Link	$\text{reg}[rd] \leftarrow \text{pc} + 4$ $\text{pc} \leftarrow \text{pc} + \text{immJ}$
JALR	<b>jalr</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Jump and Link Register	$\text{reg}[rd] \leftarrow \text{pc} + 4$ $\text{pc} \leftarrow \{(\text{reg}[rs1] + \text{immI})[31:1], 1'b0\}$
BEQ	<b>beq</b> <i>rs1</i> , <i>rs2</i> , <i>immB</i>	Branch if =	$\text{pc} \leftarrow (\text{reg}[rs1] == \text{reg}[rs2]) ? \text{pc} + \text{immB} : \text{pc} + 4$
BNE	<b>bne</b> <i>rs1</i> , <i>rs2</i> , <i>immB</i>	Branch if $\neq$	$\text{pc} \leftarrow (\text{reg}[rs1] != \text{reg}[rs2]) ? \text{pc} + \text{immB} : \text{pc} + 4$
BLT	<b>blt</b> <i>rs1</i> , <i>rs2</i> , <i>immB</i>	Branch if < (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] <_s \text{reg}[rs2]) ? \text{pc} + \text{immB} : \text{pc} + 4$
BGE	<b>bge</b> <i>rs1</i> , <i>rs2</i> , <i>immB</i>	Branch if $\geq$ (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] >_s \text{reg}[rs2]) ? \text{pc} + \text{immB} : \text{pc} + 4$
BLTU	<b>bltu</b> <i>rs1</i> , <i>rs2</i> , <i>immB</i>	Branch if < (Unsigned)	$\text{pc} \leftarrow (\text{reg}[rs1] <_u \text{reg}[rs2]) ? \text{pc} + \text{immB} : \text{pc} + 4$
BGEU	<b>bgeu</b> <i>rs1</i> , <i>rs2</i> , <i>immB</i>	Branch if $\geq$ (Unsigned)	$\text{pc} \leftarrow (\text{reg}[rs1] >_u \text{reg}[rs2]) ? \text{pc} + \text{immB} : \text{pc} + 4$
LW	<b>lw</b> <i>rd</i> , <i>immI</i> ( <i>rs1</i> )	Load Word	$\text{reg}[rd] \leftarrow \text{mem}[\text{reg}[rs1] + \text{immI}]$
SW	<b>sw</b> <i>rs2</i> , <i>immS</i> ( <i>rs1</i> )	Store Word	$\text{mem}[\text{reg}[rs1] + \text{immS}] \leftarrow \text{reg}[rs2]$
ADDI	<b>addi</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Add Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] + \text{immI}$
SLTI	<b>slti</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Compare < Immediate (Signed)	$\text{reg}[rd] \leftarrow (\text{reg}[rs1] <_s \text{immI}) ? 1 : 0$
SLTIU	<b>sltiu</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Compare < Immediate (Unsigned)	$\text{reg}[rd] \leftarrow (\text{reg}[rs1] <_u \text{immI}) ? 1 : 0$
XORI	<b>xori</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Xor Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \wedge \text{immI}$
ORI	<b>ori</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Or Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \vee \text{immI}$
ANDI	<b>andi</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	And Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \wedge \text{immI}$
SLLI	<b>slli</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Shift Left Logical Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \ll \text{immI}$
SRLI	<b>srli</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Shift Right Logical Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \gg_u \text{immI}$
SRAI	<b>srai</b> <i>rd</i> , <i>rs1</i> , <i>immI</i>	Shift Right Arithmetic Immediate	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \gg_s \text{immI}$
ADD	<b>add</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Add	$\text{reg}[rd] \leftarrow \text{reg}[rs1] + \text{reg}[rs2]$
SUB	<b>sub</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Subtract	$\text{reg}[rd] \leftarrow \text{reg}[rs1] - \text{reg}[rs2]$
SLL	<b>sll</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Shift Left Logical	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \ll \text{reg}[rs2]$
SLT	<b>slt</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Compare < (Signed)	$\text{reg}[rd] \leftarrow (\text{reg}[rs1] <_s \text{reg}[rs2]) ? 1 : 0$
SLTU	<b>sltu</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Compare < (Unsigned)	$\text{reg}[rd] \leftarrow (\text{reg}[rs1] <_u \text{reg}[rs2]) ? 1 : 0$
XOR	<b>xor</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Xor	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \wedge \text{reg}[rs2]$
SRL	<b>srli</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Shift Right Logical	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \gg_u \text{reg}[rs2]$
SRA	<b>sra</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Shift Right Arithmetic	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \gg_s \text{reg}[rs2]$
OR	<b>or</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	Or	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \vee \text{reg}[rs2]$
AND	<b>and</b> <i>rd</i> , <i>rs1</i> , <i>rs2</i>	And	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \wedge \text{reg}[rs2]$

NOTE: All immediate values (*immU*, *immJ*, *immI*, *immB*, and *immS*) are sign-extended to 32-bits.

## MIT 6.004 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
<b>li</b> <i>rd</i> , <i>constant</i>	Load Immediate	$\text{reg}[rd] \leftarrow \text{constant}$
<b>mv</b> <i>rd</i> , <i>rs1</i>	Move	$\text{reg}[rd] \leftarrow \text{reg}[rs1] + 0$
<b>not</b> <i>rd</i> , <i>rs1</i>	Logical Not	$\text{reg}[rd] \leftarrow \text{reg}[rs1] \wedge \sim 1$
<b>neg</b> <i>rd</i> , <i>rs1</i>	Arithmetic Negation	$\text{reg}[rd] \leftarrow 0 - \text{reg}[rs1]$
<b>j</b> <i>label</i>	Jump	$\text{pc} \leftarrow \text{label}$
<b>jal</b> <i>label</i>	Jump and Link (with <i>ra</i> )	$\text{reg}[ra] \leftarrow \text{pc} + 4$ $\text{pc} \leftarrow \text{label}$
<b>call</b> <i>label</i>		
<b>jr</b> <i>rs</i>	Jump Register	$\text{pc} \leftarrow \text{reg}[rs1] \& \sim 1$
<b>jalr</b> <i>rs</i>	Jump and Link Register (with <i>ra</i> )	$\text{reg}[ra] \leftarrow \text{pc} + 4$ $\text{pc} \leftarrow \text{reg}[rs1] \& \sim 1$
<b>ret</b>	Return from Subroutine	$\text{pc} \leftarrow \text{reg}[ra]$
<b>bgt</b> <i>rs1</i> , <i>rs2</i> , <i>label</i>	Branch > (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] >_s \text{reg}[rs2]) ? \text{label} : \text{pc} + 4$
<b>ble</b> <i>rs1</i> , <i>rs2</i> , <i>label</i>	Branch $\leq$ (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] <_s \text{reg}[rs2]) ? \text{label} : \text{pc} + 4$
<b>bgtu</b> <i>rs1</i> , <i>rs2</i> , <i>label</i>	Branch > (Unsigned)	$\text{pc} \leftarrow (\text{reg}[rs1] >_u \text{reg}[rs2]) ? \text{label} : \text{pc} + 4$
<b>bleu</b> <i>rs1</i> , <i>rs2</i> , <i>label</i>	Branch $\leq$ (Unsigned)	$\text{pc} \leftarrow (\text{reg}[rs1] <_u \text{reg}[rs2]) ? \text{label} : \text{pc} + 4$
<b>beqz</b> <i>rs1</i> , <i>label</i>	Branch = 0	$\text{pc} \leftarrow (\text{reg}[rs1] == 0) ? \text{label} : \text{pc} + 4$
<b>bnez</b> <i>rs1</i> , <i>label</i>	Branch $\neq$ 0	$\text{pc} \leftarrow (\text{reg}[rs1] != 0) ? \text{label} : \text{pc} + 4$
<b>bltz</b> <i>rs1</i> , <i>label</i>	Branch < 0 (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] <_s 0) ? \text{label} : \text{pc} + 4$
<b>bgez</b> <i>rs1</i> , <i>label</i>	Branch $\geq$ 0 (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] >_s 0) ? \text{label} : \text{pc} + 4$
<b>bgtz</b> <i>rs1</i> , <i>label</i>	Branch > 0 (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] >_s 0) ? \text{label} : \text{pc} + 4$
<b>blez</b> <i>rs1</i> , <i>label</i>	Branch $\leq$ 0 (Signed)	$\text{pc} \leftarrow (\text{reg}[rs1] <_s 0) ? \text{label} : \text{pc} + 4$

## MIT 6.004 ISA Reference Card: Calling Convention

Registers	Symbolic names	Description	Saver
<b>x0</b>	<b>zero</b>	Hardwired zero	—
<b>x1</b>	<b>ra</b>	Return address	Caller
<b>x2</b>	<b>sp</b>	Stack pointer	Callee
<b>x3</b>	<b>gp</b>	Global pointer	—
<b>x4</b>	<b>tp</b>	Thread pointer	—
<b>x5-x7</b>	<b>t0-t2</b>	Temporary registers	Caller
<b>x8-x9</b>	<b>s0-s1</b>	Saved registers	Callee
<b>x10-x11</b>	<b>a0-a1</b>	Function arguments and return values	Caller
<b>x12-x17</b>	<b>a2-a7</b>	Function arguments	Caller
<b>x18-x27</b>	<b>s2-s11</b>	Saved registers	Callee
<b>x28-x31</b>	<b>t3-t6</b>	Temporary registers	Caller

## MIT 6.004 ISA Reference Card: Instruction Encodings

31	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]				rs1		funct3		rd		opcode		I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]								rd		opcode		U-type
imm[20 10:1 11 19:12]								rd		opcode		J-type

## RV32I Base Instruction Set (MIT 6.004 subset)

imm[31:12]				rd	0110111	LUI
imm[20 10:1 11 19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

## 2 Appendix: RISC-V Simulator Reference

We provide two simulator versions, one with a command line interface (CLI) and one with a graphical user interface (GUI). This appendix details how to interact with the GUI simulator to debug your code.

Before getting started, make sure that you have run ‘make’ successfully. For lab 1, check that files **triangular.vmh** and **bubblesort.vmh** exist.

### 2.1 Graphical User Interface

**Start the simulator** using the following command:

```
rv_sim_gui
```

**GUI simulator workflow:** Select the program and the test case you want to run. Then click the ‘Load Program’ button.

Click ‘Run’ to let the machine execute the program to completion. Click ‘Step’ to let the machine execute the next instruction only. If a break point is reached, the execution will stop.

You can optionally set and modify break points before clicking ‘Run’ or ‘Step’. Make sure you enter the PCs of the break points separated by spaces before clicking ‘Apply’. [Section 2.2](#) explains how to locate the address of a specific instruction to set a break point.

Multiple tabs display the output of the program together with the machine state, including register contents and memory contents.

Click ‘Exit Program’ to quit the execution of the current program. After that, the machine is ready to load another program.

### 2.2 Finding the address of an instruction

After running ‘make’, a *.dump* file will be generated for each program. The *.dump* file shows the assembler mnemonics for the machine instructions from the program.

**bubblesort.dump** can be very useful to debug **bubblesort.S**. For instance, if you compile without any modification to **bubblesort.S**, and search ‘<sort>:’ in the generated **bubblesort.dump**, you would see contents similar to the following:

```
00000038 <sort>:
```

```
38: 00008067          ret
```

This indicates that the address of both the label <sort> and the (pseudo)instruction **ret** is 0x38, and the instruction encoding is 0x00008067. Therefore, you can set a break point at 0x38 if you want your program to stop right before executing **ret**.