RISC-V Assembly 1

Due on Thursday 14 February by 23:59’59” EST.

Getting Started To create your Lab 1 repository, visit the repository creation page at 6004.mit.edu/web/spring19/user/labs/lab1. Once your repository is created, you can clone it into your VM by running

    git clone git@github.mit.edu:6004-spring19/labs-lab1-KERBEROS.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 to verify that your submission passes all the tests. If you finish the lab in time but forget to push, you will incur the standard late submission penalties.

Grading the Lab To pass the lab, you must complete all sections and pass all tests. You must pass all labs to pass the course. However, you will only be graded on sections 3½ and 4.

Check-off Meeting After turning in this lab, you are required to go to the lab for a check-off meeting by Wednesday 20 February. See the course website for lab hours.

1 Introduction

Most of this course is dedicated to designing general-purpose, programmable machines — machines that can stream cat videos, solve differential equations, or display the front page of Reddit. But before we actually begin to design a programmable computer, we have to know how we want to program it. What things should it be able to do? Which operations will be designated by which inputs?

In 6.004, we’ll use an architecture* called RISC-V. There are a few optional parts of RISC-V, but we’ll be focusing on the common denominator, called RV32. There is only one data type — a binary number called a word (1 word = 32 bits in RV32). You can only store up to 32 words at a time, in the so-called registers. If you want more, you have to load words from main memory and store them back in main memory when you’re done. Besides moving words around, you can do some basic arithmetic (like addition and subtraction), looping, and if-then branching, but that’s it!

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

* An architecture, or more specifically an instruction set architecture, is a model of the way a computer is organized.
Python

\[
\begin{align*}
n = 9 \\
\text{# check if } n \text{ is even} \\
\text{if } n \% 2 == 0: \\
\quad n = n / 2 \\
\text{else:} \\
\quad n = 3 * n + 1 \\
\end{align*}
\]

RISC-V (assembly)

\[
\begin{align*}
\text{li } a0, 9 \\
\text{# check if } n \text{ is even} \\
\text{If:} \\
\quad \text{andi } a1, a0, 1 \\
\quad \text{bnez } a1, \text{Else} \\
\quad n = n / 2 \\
\text{Else:} \\
\quad n = 3 * n + 1 \\
\quad \text{slli } a2, a0, 1 \\
\quad \text{add } a0, a2, a0 \\
\quad \text{addi } a0, a0, 1 \\
\quad \text{End:}
\end{align*}
\]

RISC-V (binary)

\[
\begin{align*}
00: & \quad 00000000000000000000000000000000 \\
04: & \quad 00000100101010101010000010100111 \\
08: & \quad 00000000000000000000000000000000 \\
0C: & \quad 00000000000000000000000000000000 \\
10: & \quad 00000000000000000000000000000000 \\
14: & \quad 00000000000000000000000000000000 \\
18: & \quad 00000000000000000000000000000000 \\
1C: & \quad 00000000000000000000000000000000 \\
\end{align*}
\]

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

**li a0, 9**

The opcode li stands for *load immediate.* "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.\(^\dagger\) In English, this 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.

**Check yourself:** Take a moment to figure out why this is the case!

---

\(^*\) This is actually a *pseudo-instruction*; it is shorthand for addi a0, zero, 9. The register zero always contains zero.

\(^\dagger\) The thirty-two registers are numbered 0–31, and we could have referred to a0 by writing x10 instead.
To keep the words from getting tedious, from now on we'll use a right arrow ⇒ when we're describing the effect of an instruction, and the running prose will just be a commentary.

\**bnez a1, Else**  
The opcode \*bnez\* stands for *branch if not equal to zero.*\^ 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 otherwise.

⇒ 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 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 \*eab...wxy.\^ Since numbers are represented in binary, shifting the digits right by one has the effect of dividing by 2.

⇒ 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* \((= 2 \times a0)\) and place the result \((= 3 \times a0)\) 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 \*a0 ÷ 2\* if *a0* was even and \*a0×3 + 1\* if *a0* was odd.

\* This is also a pseudo-instruction; it is shorthand for \*bne a1, zero, Else.\* The register \*zero\* always contains zero.
\^ 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.
The Collatz conjecture states that for any $n_0$ the sequence $n_0, n_1, n_2, ...$ eventually reaches 1, where each $n_i$ after $i = 0$ is defined as

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

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

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

This is possible using only the instructions we've seen so far, but if you'd like to use others, take a look at the appendices.

---

### 2 The 9th of September, 1947

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.* To run it, open up the terminal and navigate to the lab repo; then enter the two following commands:

```
student@6.004-VM:~$ cd lab1
student@6.004-VM:~/lab1$ ./assemble-collatz.sh
student@6.004-VM:~/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 binary. Under that (beneath “Disassembly of section .text.init”) is an instruction-by-instruction listing of the program. The first five lines of the disassembly are reproduced at the top of the next page.

---

* Technically, `assemble-collatz` 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.
On the left (0:) is the address of each instruction. Beside that (00000513) 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, ...). The RISC-V address space is byte-indexed; and since each instruction is 32 bits, but a byte is 8 bits, each instruction spans four indices.

Fundamentally, all programs that your computer will ever run look like this. They may start 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!

Federal Regulations Require Me to Warn You that This Next Test... Is Looking Pretty Good

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 spec – 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 enter make, like so:

```
student @ 6.004-VM:/lab1$ make
```

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

Each lab release includes a RISC-V simulator; this is how you will run the tests. It also can help you debug your programs – plus give you a chance to see your code in action!

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, enter `python sim_gui.pyw`, like so:

```
student@6.004-VM:~/lab1 $ python sim_gui.pyw
```

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 **Exit Program**. If you modify a program, you must re-run `make` and then re-load 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 \( a_0, a_1, a_2, \) and PC will change.

**Check yourself:** In test 1, does the simulation branch to <Else> (the case when \( a_0 \) is initially odd) or continue on to <Then> (the case when \( a_0 \) is initially even)? How about in test 2?

---

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, &c) 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\( n \)), 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,\(^*\) 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])

...  
```

\(^*\) Traditionally called the *pointer* to the array.
Idiomatic Python

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

Alternate Version (more similar to RISC-V)

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:

A 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.

B Our function should store its result (the largest word in the array) in a0 and end with a ret instruction.

C 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!

1 Decide which registers will be used for which variables. It was stipulated that p \rightarrow a0 and n \rightarrow a1. Let's (arbitrarily) decide that largest_so_far \rightarrow a2 (which we'll copy to a0 when we're done) and w \rightarrow 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, &c) into branches and jumps.

There are several ways to implement loops; the next page demonstrates two 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 \texttt{lw a3, \%a0} instruction. The opcode \texttt{lw} stands for \textit{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:

\texttt{lw rd, imm(rs)}  \quad \text{Suppose the content of the register \texttt{rs} is some number \texttt{n}. Then the word at address \texttt{(n + imm)} in memory will be loaded into register \texttt{rd}.}

For example, if the value of \texttt{rs} was \texttt{0x 0004 0100} and \texttt{imm} was 8, then the word located at address \texttt{0x 0004 0108} will be put into \texttt{rd}.

To put a word \textit{back} into memory, we can use the \texttt{sw} instruction, which works similarly.

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

If you open up \texttt{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 last part of the lab, you'll have to write your own loads and stores.

\textit{Complete maximum.S and get it to pass all of the tests in the simulator.}

\textbf{3½ A000217} \quad \textbf{30 POINTS}

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

\begin{center}
\begin{tabular}{cccccc}
\hline
\textbullet & \textbullet & \textbullet & \textbullet & \textbullet & \textbullet \\
\hline
\end{tabular}
\end{center}

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

\textbf{A}  \quad \text{When the function is called, the argument \texttt{n} will be stored in \texttt{a0}.}

\textbf{B}  \quad \text{The function should store its result in \texttt{a0} and end with a \texttt{ret} instruction.}

\textbf{C}  \quad \text{The function should only use the \texttt{a0-a7} registers.}

\textit{Complete triangular.S and get it to pass all of the tests in the simulator.}
**4 Very Small Rocks**

**70 POINTS**

Bubble sort – 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. Write this function in RV32 assembly, subject to the following constraints:

**A** 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`.

**B** The sort function should modify the array directly, and need not return any value.

**C** The sort function should only use the `a0-a7` registers.

**D** Your implementation should only use the subset of RV32 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 `auipc`. **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 `python sim_gui.pyw`. Alternatively, you can run the tests by entering

```
student@6.004-VM:~/lab1$ python sim.py bubblesort
```

To run tests one a time, enter

```
student@6.004-VM:~/lab1$ python sim.py 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 Questions:** Be prepared to walk through collatz or maximum with a staff member during checkoff, and predict how the register contents will change after each instruction before actually stepping the simulator.

* Which apparently = lighter.
† It's also called sinking sort for the opposite reason.
# Appendix: RISC-V ISA Reference

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 sim.py or sim_gui.pyw). You are encouraged to use pseudo-instructions to simplify your code.

## MIT 6.004 ISA Reference Card: Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>lui rd, immU</td>
<td>Load Upper Immediate</td>
<td>reg[rd] &lt;= immU &lt;&lt; 12</td>
</tr>
<tr>
<td>JAL</td>
<td>jal rd, immJ</td>
<td>Jump and Link</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pc &lt;= pc + immJ</td>
</tr>
<tr>
<td>JALR</td>
<td>jalr rd, rs1, immJ</td>
<td>Jump and Link Register</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pc &lt;= {(reg[rs1] + immJ)[31:1], 1'b0}</td>
</tr>
<tr>
<td>BEQ</td>
<td>beq rs1, rs2, immB</td>
<td>Branch if =</td>
<td>pc &lt;= (reg[rs1] == reg[rs2]) ? pc + immB</td>
</tr>
<tr>
<td>BNE</td>
<td>bne rs1, rs2, immB</td>
<td>Branch if ≠</td>
<td>pc &lt;= (reg[rs1] !- reg[rs2]) ? pc + immB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>: pc + 4</td>
</tr>
<tr>
<td>BLT</td>
<td>blt rs1, rs2, immB</td>
<td>Branch if &lt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt; reg[rs2]) ? pc + immB</td>
</tr>
<tr>
<td>BGE</td>
<td>bge rs1, rs2, immB</td>
<td>Branch if ≥ (Signed)</td>
<td>pc &lt;= (reg[rs1] ≥ reg[rs2]) ? pc + immB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>: pc + 4</td>
</tr>
<tr>
<td>BLTU</td>
<td>bltu rs1, rs2, immB</td>
<td>Branch if &lt; (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &lt; reg[rs2]) ? pc + immB</td>
</tr>
<tr>
<td>BGEU</td>
<td>bgeu rs1, rs2, immB</td>
<td>Branch if ≥ (Unsigned)</td>
<td>pc &lt;= (reg[rs1] ≥ reg[rs2]) ? pc + immB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>: pc + 4</td>
</tr>
<tr>
<td>LW</td>
<td>lw rd, immI(rs1)</td>
<td>Load Word</td>
<td>reg[rd] &lt;= mem[reg[rs1] + immI]</td>
</tr>
<tr>
<td>SW</td>
<td>sw rs2, immS(rs1)</td>
<td>Store Word</td>
<td>mem[reg[rs1] + immS] &lt;= reg[rs2]</td>
</tr>
<tr>
<td>ADDI</td>
<td>addi rd, rs1, immI</td>
<td>Add Immediate</td>
<td>reg[rd] &lt;= reg[rd] + reg[rs1]</td>
</tr>
<tr>
<td>SLLI</td>
<td>slli rd, rs1, immI</td>
<td>Compare &lt; Immediate (Signed)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt; reg[rs1]) ? 1 : 0</td>
</tr>
<tr>
<td>SLTIU</td>
<td>sltiu rd, rs1, immI</td>
<td>Compare &lt; Immediate (Unsigned)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt; reg[rs1]) ? 1 : 0</td>
</tr>
<tr>
<td>XORI</td>
<td>xor rd, rs1, imm1</td>
<td>XOR Immediate</td>
<td>reg[rd] &lt;= reg[rd] ^ reg[rs1]</td>
</tr>
<tr>
<td>ORI</td>
<td>ori rd, rs1, imm1</td>
<td>Or Immediate</td>
<td>reg[rd] &lt;= reg[rd]</td>
</tr>
<tr>
<td>ANDI</td>
<td>andi rd, rs1, imm1</td>
<td>And Immediate</td>
<td>reg[rd] &lt;= reg[rd] &amp; reg[rs1]</td>
</tr>
<tr>
<td>SLL</td>
<td>sll rd, rs1, imm1</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] &lt;= reg[rd] &lt;&lt; imm1</td>
</tr>
<tr>
<td>SLR</td>
<td>srl rd, rs1, imm1</td>
<td>Shift Right Logical Immediate</td>
<td>reg[rd] &lt;= reg[rd] &gt;&gt; imm1</td>
</tr>
<tr>
<td>SRAI</td>
<td>srai rd, rs1, imm1</td>
<td>Shift Right Arithmetic Immediate</td>
<td>reg[rd] &lt;= reg[rd] &gt;&gt; u imm1</td>
</tr>
<tr>
<td>ADD</td>
<td>add rd, rs1, rs2</td>
<td>Add</td>
<td>reg[rd] &lt;= reg[rd] + reg[rs2]</td>
</tr>
<tr>
<td>SUB</td>
<td>sub rd, rs1, rs2</td>
<td>Subtract</td>
<td>reg[rd] &lt;= reg[rd] - reg[rs2]</td>
</tr>
<tr>
<td>SLL</td>
<td>sll rd, rs1, rs2</td>
<td>Shift Left Logical</td>
<td>reg[rd] &lt;= reg[rd] &lt;&lt; reg[rs2]</td>
</tr>
<tr>
<td>SLR</td>
<td>srl rd, rs1, rs2</td>
<td>Shift Right Logical</td>
<td>reg[rd] &lt;= reg[rd] &gt;&gt; u reg[rs2]</td>
</tr>
<tr>
<td>SRA</td>
<td>sra rd, rs1, rs2</td>
<td>Shift Right Arithmetic</td>
<td>reg[rd] &lt;= reg[rd] &gt;&gt; reg[rs2]</td>
</tr>
<tr>
<td>OR</td>
<td>or rd, rs1, rs2</td>
<td>Or</td>
<td>reg[rd] &lt;= reg[rd]</td>
</tr>
<tr>
<td>AND</td>
<td>and rd, rs1, rs2</td>
<td>And</td>
<td>reg[rd] &lt;= reg[rd] &amp; reg[rs2]</td>
</tr>
</tbody>
</table>
### MIT 6.004 ISA Reference Card: Pseudoinstructions

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>li rd, constant</code></td>
<td>Load Immediate</td>
<td><code>reg[rd] &lt;= constant</code></td>
</tr>
<tr>
<td><code>mv rd, rs1</code></td>
<td>Move</td>
<td><code>reg[rd] &lt;= reg[rs1] + 0</code></td>
</tr>
<tr>
<td><code>not rd, rs1</code></td>
<td>Logical Not</td>
<td><code>reg[rd] &lt;= reg[rs1] &amp; ~1</code></td>
</tr>
<tr>
<td><code>neg rd, rs1</code></td>
<td>Arithmetic Negation</td>
<td><code>reg[rd] &lt;= 0 - reg[rs1]</code></td>
</tr>
<tr>
<td><code>j label</code></td>
<td>Jump</td>
<td><code>pc &lt;= label</code></td>
</tr>
<tr>
<td><code>jal label</code></td>
<td>Jump and Link (with ra)</td>
<td><code>reg[ra] &lt;= pc + 4</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>pc &lt;= label</code></td>
</tr>
<tr>
<td><code>jr rs</code></td>
<td>Jump Register</td>
<td><code>pc &lt;= reg[rs1] &amp; ~1</code></td>
</tr>
<tr>
<td><code>jalr rs</code></td>
<td>Jump and Link Register (with ra)</td>
<td><code>reg[ra] &lt;= pc + 4</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>pc &lt;= reg[rs1] &amp; ~1</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Return from Subroutine</td>
<td><code>pc &lt;= reg[ra]</code></td>
</tr>
<tr>
<td><code>bgt rs1, rs2, label</code></td>
<td>Branch &gt; (Signed)</td>
<td><code>pc &lt;= (reg[rs1] &gt; s reg[rs2]) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>ble rs1, rs2, label</code></td>
<td>Branch ≤ (Signed)</td>
<td><code>pc &lt;= (reg[rs1] &lt;= s reg[rs2]) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>bgtu rs1, rs2, label</code></td>
<td>Branch &gt; (Unsigned)</td>
<td><code>pc &lt;= (reg[rs1] &gt; s reg[rs2]) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>bleu rs1, rs2, label</code></td>
<td>Branch ≤ (Unsigned)</td>
<td><code>pc &lt;= (reg[rs1] &lt;= s reg[rs2]) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>beqz rs1, label</code></td>
<td>Branch = 0</td>
<td><code>pc &lt;= (reg[rs1] == 0) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>bnez rs1, label</code></td>
<td>Branch ≠ 0</td>
<td><code>pc &lt;= (reg[rs1] != 0) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>bltz rs1, label</code></td>
<td>Branch &lt; 0 (Signed)</td>
<td><code>pc &lt;= (reg[rs1] &lt; 0) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>buge rs1, label</code></td>
<td>Branch ≥ 0 (Signed)</td>
<td><code>pc &lt;= (reg[rs1] &gt;= 0) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>bgtz rs1, label</code></td>
<td>Branch &gt; 0 (Signed)</td>
<td><code>pc &lt;= (reg[rs1] &gt; 0) ? label : pc + 4</code></td>
</tr>
<tr>
<td><code>blez rs1, label</code></td>
<td>Branch ≤ 0 (Signed)</td>
<td><code>pc &lt;= (reg[rs1] &lt;= 0) ? label : pc + 4</code></td>
</tr>
</tbody>
</table>
6 Appendix: RISC-V Simulator Reference

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

Before getting started, make sure that you have run `make` successfully. For lab 1, check that the file `bubblesort.vmh` exists.

6.1 Command Line Interface

Start the simulator using the following command:

```bash
python -i sim.py <program_name> <test_number>
```

For lab 1, `<program_name>` is `bubblesort`, and `<test_number>` is the argument to the program, which should be an integer from 1 to 5. Omitting the `<test_number>` will run all test cases sequentially. The `-i` option denotes you want to run in interactive mode. If you omit it, the simulator will simply run the program to completion.

List all supported commands by typing `help()` after you start the simulator in interactive mode. The simulator will list all usable commands together with an example:

```
>>> help()
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>run()</td>
<td>Run program until it either finishes or reaches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a break point.</td>
<td></td>
</tr>
<tr>
<td>step(&lt;n&gt;)</td>
<td>Run &lt;n&gt; more instructions and stop.</td>
<td>...</td>
</tr>
<tr>
<td>setBps([&lt;pc0&gt;, ...])</td>
<td>Clear previous break points and set new ones.</td>
<td>...</td>
</tr>
</tbody>
</table>
|         | ... | ...
| addBps([<pc0>, ...]) | Add break points. | |
| clearBps([<pc0>, ...]) | Clear break points. | ...
| showStats() | List statistics, e.g. instruction counts | |
| showRegs() | List registers including pc register | |
| showMem(<start>, <end>) | List memory layout from address <start> to <end> | ...
| showStack() | List memory layout of the stack specified by | |
|             | register x2(sp) | ...
| showInstr(<pc>) | Display the instruction at <pc> | ... |
| showLablePC(<label>) | Display the PC of <label> | ... |
| restart() | Restart the execution. | |
```

Break points let you interrupt program execution before a particular instruction executes. They are a useful debugging tool, as they let you inspect the state of the machine at specific points of interest. ?? discusses how to locate the address of a specific instruction to set a break point.

Example:

```
$ python -i sim.py bubblesort 1
>>> setBps([0x38])
# Set a break point at 0x38
>>> run()
# Stop before executing the instruction at 0x38
>>> run()
# Stop before executing the instruction at 0x38
>>> showRegs()
# Print out all registers including PC
>>> clearBps()
# Clear all break points
>>> run()
# Run to completion
>>> restart()
# Restart the machine with the same program
# and the same test case(s)
>>> step()
# Execute only one instruction
>>> exit()
# Quit
```
6.2 Graphical User Interface

Start the simulator using the following command:

```python
python sim_gui.pyw
```

The GUI has similar functionality to that of the CLI.

**Example:** 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 6.3 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.

6.3 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:

```plaintext
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`. 