Due date: Thursday February 27th 11:59:59pm EST.

Points: This lab is worth 12 points (out of 200 points in 6.004).

Getting started: To create your initial Lab 2 repository, please visit `https://6004.mit.edu/web/spring20/user/labs/lab2`. Once your repository has been created, you can clone it into Athena by running:

```
git clone git@github.mit.edu:6004-spring20/labs-lab2-{YourMITUsername}.git lab2
```

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 required 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 are required to go to the lab for a check-off meeting by Wednesday March 4th. Checkoffs for this lab will begin on Friday, February 21st. 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.

To be able to checkoff this lab you must do all exercises except the one in Section 6, answer all discussion questions, and PASS all the tests in Didit. Required exercises are outlined with a solid line while optional exercises are outlined with a dashed line.

Introduction

As we write larger and more complicated programs, it becomes increasingly important to structure our programs as a collection of small, separate, and reusable components:

- Monolithic code is often difficult to understand, so we can make our code more understandable by giving it some structure. Not only is this kind to others and your future self, but it increases the odds of writing programs correctly the first time around.
- There may be blocks of code that we wish to reuse, so we can turn those blocks into procedures that are capable of being run multiple times, perhaps with different arguments. This also means that, if you decide to change the procedure, you only have to make changes in one place instead of many.

So we expect that there will be many procedures in memory at the same time, possibly written by different people, and those procedures will frequently call each other (and even recursively call themselves). How do they know how to play nicely together? If I want to use your function, where do I put the arguments? Where will I get the results? This is a problem! The solution is a *calling convention*—a set of rules that isn't enforced by hardware, but that everyone agrees to follow.

1 Registers

The registers `a0`–`a7` are function argument registers. If a procedure takes any arguments, this is where they will be when the procedure starts. The registers `a0` and `a1` are also the return value registers. If a procedure is supposed to return a value, the value will be available in these registers after the procedure ends.

Consider the two functions below, **`triangular`** and **`square`**. Both take one argument in `a0` and return their result in `a0`.

```
//computes n-th triangular number,
//using closed formula (n^2 + n)/2
triangular:
    mv    a1, a0
    call  square
    add   a0, a0, a1 //a0 = n^2 + n
    srli  a0, a0, 1  //divide by 2
    ret
```

```
//computes n^2 for unsigned numbers,
//using repeated addition
square:
    mv    a1, a0     //a1 is counter
    li    a2, 0      //a2 is running sum
loop:
    add   a2, a2, a0
    addi  a1, a1, -1
    bnez  a1, loop
done:
    mv    a0, a2
    ret
```

As-is, the **triangular** function does not actually work. The number n stored in a1 gets overwritten by the **square** function. In fact, when **square** finishes, a1 always contains 0, so **triangular** is actually returning $n^2/2$. This sort of conflict can happen all the time, and so calling conventions state that a procedure must either **(A)** save the registers that it cares about before calling another procedure, or **(B)** trust that the procedures it calls will leave the registers the way they found them. Which means that if someone calls *your* procedure, you must save registers before you use them, and put things back the way they were before you return. The **RISC-V** calling convention actually uses both strategies (summarized in the appendix):

- Before calling a different procedure, the procedure is responsible for saving every register it cares about that *doesn't* begin with the letter "s" (this is strategy **(A)**, and applies to ra, a0, a1, a2, . . . , and t0, t1, . . .). These are the *caller-saved* registers.
- Before using a register that *does* begin the with the letter "s", the procedure must save the register in order to restore it later (before returning to whatever function called *the current procedure*—this is strategy **(B)**, and applies to sp, s0, s1, . . .). These are the *callee-saved* registers.

To properly follow the calling convention, we need to consider the cases in which our functions are callers *and* the cases in which our functions are callees:

1) Caller is unknown and Callee is **triangular**
   - The **triangular** function does not modify any of the s- registers, and so it does not need to save them.

2) Caller is **triangular** and Callee is **square**
   - The **triangular** function is responsible for saving a1 and ra, since a1 will used in the **add** instruction after the call to **square** and ra is used in the **ret** instruction. Both **call** and **ret** are pseudo-instructions, expanding to `jal ra, label` and `jalr zero, (0)ra` respectively. In this lab, you are required to use the pseudo-instruction forms rather than the **jal**/**jalr** forms. This is also good practice since it makes your code easier to read. The **square** function does not modify any of the s-registers, and so it does not need to save them.

3) Caller is **square** and Callee is unknown
   - The **square** function does not call any other functions, and so it does not need to save any of the non-s registers.

We save registers by pushing them onto the *stack*, which is simply a region of memory that we've set aside for the purpose of saving registers. The *stack pointer*, which is just the register sp, contains the address of the top of the stack at all times. Confusingly, the stack grows *downwards* in memory as the stack gets bigger. We traditionally write small address at the top of the page and big addresses at the bottom of the page, so the top of the stack is at the lowest address. As the top of the stack moves up the page, it moves down in the address space. Taking this altogether, we present correct versions of **triangular** and **square** with the minimal number of alterations:

```
triangular:
    mv    a1, a0
    addi  sp, sp, -8  //push ra and a1
    sw    ra, 4(sp)   //onto the stack
    sw    a1, 0(sp)

    call  square
    lw    a1, 0(sp)   //pop a1 and ra
    lw    ra, 4(sp)   //off the stack

    addi  sp, sp, 8
    add   a0, a0, a1  //a0 = n^2 + n
    srli  a0, a0, 1   //divide by 2
    ret
```

```
square:
    mv    a1, a0      //a1 is counter
    li    a2, 0       //a2 is running sum
loop:
    add   a2, a2, a0
    addi  a1, a1, -1
    bnez  a1, loop
done:
    mv    a0, a2
    ret
```

2 Factorial

Exercise 1 (20%): Your first task is to write the procedure **factorial** in **factorial.S**. **factorial** should take a number n as an argument and return $n!$ as its result. The argument n will be passed through a0, and the result should be placed in a0. The code skeleton in **factorial.S** contains a python implementation for reference.

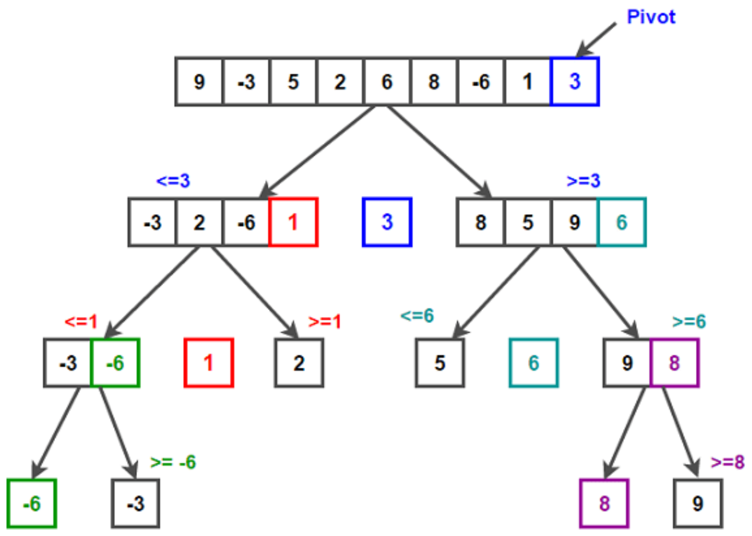
To compute the factorial of n, you should call the **mul** procedure (using **call** mul) to do each multiplication, which has been defined for you. The **mul** procedure takes the multiplier and multiplicand in a0 and a1, and returns the product in a0.

- You are permitted to use any registers you'd like (as long as you follow the calling convention), *except* for gp and tp.
- The staff have modified the simulator to partially check for adherence to the calling convention. (At each **ret**, the simulator will ensure the values in the callee-saved registers are the same as they were when the procedure was called.) Because of this, you *must* use the **call** pseudoinstruction to make all procedure calls rather than **jal**.
- Your code must pass all five tests in the simulator. You may quickly run them all using the command-line verison of the simulator: `rv_sim factorial`
- The tests will fail if you do not use the provided **mul** procedure.
- The tests will never call **factorial** with a number large enough to cause overflow, and you are not expected to handle inputs that are large enough to cause overflow.

3 Quicksort

In Lab 1, we saw one way to sort a list of numbers—Bubblesort. Bubblesort is fairly simple to implement but is not very efficient unless the input array is tiny or it's almost sorted to begin with. Quicksort is a different sorting algorithm that's more complex but usually gives better performance.

In Bubblesort, the array is sorted one element at a time, moving it further up the array until it was in the right position. In Quicksort, we use a divide-and-conquer approach with three key steps. First, a pivot is selected, often just the first or last element in the array (in your implementation you'll use the last element). Then, the elements of the array are rearranged (or "partitioned") into two sub-arrays: all the elements less than or equal to the pivot and all the elements greater than the pivot. Finally, each of the sub-arrays is recursively sorted using the same quicksort procedure. The figure below illustrates this graphically and a reference implementation is provided in the code skeleton. It is not essential that you understand the algorithm in detail to complete this exercise but it may help you with debugging or the Design Project.



Exercise 2 (80%): Implement Quicksort in **quicksort.S** (abbreviated to **sort** in the code skeleton), which should take the starting address of an array of words to be sorted (in `a0`), the starting index of the region to be sorted (in `a1`), and the ending index of the region to be sorted (in `a2`). The starting index and ending index are element indices, not addresses or byte offsets.

- Your implementation of Quicksort should be recursive, and must include a separate **partition** procedure to better organize your code.
- You are permitted to use any registers you'd like, *except* for `gp` and `tp`.
- For the time being, you should ignore the **kth_smallest** procedure at the bottom of the code skeleton.
- The staff have modified the simulator to partially check for adherence to the calling convention. (At each **ret**, the simulator will ensure the values in the callee-saved registers are the same as they were when the procedure was called). Because of this, you *must* use the **call** pseudoinstruction to make all procedure calls rather than **jal**.
- Your code must pass all five tests in the simulator. You may quickly run them all using the command-line verison of the simulator: `rv_sim quicksort`

4 Memory-Mapped I/O (MMIO)

As fabulous as computers are, they'd be pretty useless unless they could interact with the outside world. There are many ways to do this; one common way is *memory-mapped input and output* (MMIO). As the name implies, a computer with MMIO gets input by reading from a special location in memory (using `lw`), and sends output by writing to a special location in memory (using `sw`).

In all likelihood, memory does not actually exist at these locations, but load and store requests to these *addresses* trigger something special. Instead of sending the requests to memory, the requests are redirected to some other piece of hardware (like a keyboard or display). That hardware pretends to work like memory but may not store data at all. This may be a bit confusing at first, so let's look at a few examples.

If you open the file **`MMIO-addresses.txt`**, you'll see a list of I/O channels. Each channel is mapped to an address (left column), is specified as read- or write-only (middle column), and has a description (right column). If we wanted to print the number 17 to the console in decimal, we would write:

```
li a1, 0x40000004
li a0, 17
sw a0, 0(a1)
```

If we wanted to print the same number in hexadecimal instead, we would write:

```
li a1, 0x40000008
li a0, 17
sw a0, 0(a1)
```

To get a number that's typed into the console from the keyboard, we could write:

```
li a1, 0x40004000
lw a0, 0(a1)
```

Note that our simulator will pause execution and wait for input when this code is executed. In the GUI simulator, you will need to click the "Step" or "Run" button to resume execution after typing your input into the box at the bottom. In the command-line simulator, just type your input and press the "Enter" key on your keyboard.

Exercise 3: Now write **`kth_smallest`**, which should take the starting address of a sorted array (passed through `a0`), read a number k from the console, and then print the k-th element of the array to the console. The function does not need to return anything; the value of `a0` at the end of the function does not matter. You can test **`kth_smallest`** by running Quicksort; **`kth_smallest`** will automatically be called after each array is sorted.

This section is not graded by Didit, but staff will check for completion during checkoff.

5 Monitoring Program Performance

MMIO can be used for more than just printing, or reading input from the keyboard. The flexibility of the interface means that real-time clocks, sound controllers, video buffers, and other devices are all accessible using instructions that we're already familiar with. We don't cover external hardware in 6.004, but there's one type of device that's usually built into processors—hardware counters.

Nearly all modern computers include hardware counters. They are used to keep track of the number of instructions that the processor executes, the time it takes to run these instructions, or to count events of interest (*e.g.,* perhaps you want to know how many times a `lw` instruction gets executed while runnning a particular program).

Our simulator provides two hardware counters. First, the *instruction counter*, which is located at address `0x40005000`, counts the number of instructions executed since the simulation started. Second, the *performance counter*, which is located at address `0x40006000`, is a *pausable instruction counter*: it increments after each instruction is executed, but can be paused and resumed by writing the values `0x0` and `0x1` to address `0x40006004`, respectively. The performance counter starts in the paused state so it will start counting from zero the first time you write `0x1` to address `0x40006004`.

Exercise 4: Copy over your partition and quicksort procedures into the template in the **`benchmark`** folder. Then, modify your code so that it uses MMIO to count and display two values: 1) the total number of instructions executed while running your Quicksort implementation, and 2) the total number of instructions executed *only* while inside the **`partition`** procedure. Run your modified code on test 1 *inside* the **`benchmark`** folder.

This section is not graded by Didit, but staff will check for completion during checkoff.

Notes:

- Your modified code should only print two values, the two totals that we asked for. It should not print a separate number for each time that partition is called.
- The instruction counter begins counting when the simulation begins. Because our test code does some things before your sort routine is called, this counter will not be zero when your routine is first entered.
- You should try to count the number of instructions as accurately as possible but it's OK to miss one or two at the beginning and/or end of your procedure.
- In this exercise, you are only required to measure the performance; you do not have to optimize or improve the sort.

Discussion Question 1: Assuming every instruction takes the same amount of time, what percentage of time does your code spend in **`partition`**? If you wanted to improve the performance of your code, which portion would you look at first?

6 Improving Performance Optional

Exercise 5: If you'd like more practice with assembly, try to optimize your Quicksort implementation! The compiler used by the staff generates code that can sort the array in about **150000 cycles**; the best implementation we know of does it in about **50000 cycles**. Usually, it's possible to beat the compiler by a sizable margin without having to resort to really advanced maneuvers. This exercise is optional and does not carry extra points but will give you a headstart on the Design Project.

7 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	**lui** rd, immU	Load Upper Immediate	reg[rd] <= immU << 12
JAL	**jal** rd, immJ	Jump and Link	reg[rd] <= pc + 4 pc <= pc + immJ
JALR	**jalr** rd, rs1, immI	Jump and Link Register	reg[rd] <= pc + 4 pc <= {(reg[rs1] + immI)[31:1], 1'b0}
BEQ	**beq** rs1, rs2, immB	Branch if $=$	pc <= (reg[rs1] == reg[rs2]) ? pc + immB : pc + 4
BNE	**bne** rs1, rs2, immB	Branch if $\neq$	pc <= (reg[rs1] != reg[rs2]) ? pc + immB : pc + 4
BLT	**blt** rs1, rs2, immB	Branch if $<$ (Signed)	pc <= (reg[rs1] <$_s$ reg[rs2]) ? pc + immB : pc + 4
BGE	**bge** rs1, rs2, immB	Branch if $\geq$ (Signed)	pc <= (reg[rs1] >=$_s$ reg[rs2]) ? pc + immB : pc + 4
BLTU	**bltu** rs1, rs2, immB	Branch if $<$ (Unsigned)	pc <= (reg[rs1] <$_u$ reg[rs2]) ? pc + immB : pc + 4
BGEU	**bgeu** rs1, rs2, immB	Branch if $\geq$ (Unsigned)	pc <= (reg[rs1] >=$_u$ reg[rs2]) ? pc + immB : pc + 4
LW	**lw** rd, immI(rs1)	Load Word	reg[rd] <= mem[reg[rs1] + immI]
SW	**sw** rs2, immS(rs1)	Store Word	mem[reg[rs1] + immS] <= reg[rs2]
ADDI	**addi** rd, rs1, immI	Add Immediate	reg[rd] <= reg[rs1] + immI
SLTI	**slti** rd, rs1, immI	Compare $<$ Immediate (Signed)	reg[rd] <= (reg[rs1] <$_s$ immI) ? 1 : 0
SLTIU	**sltiu** rd, rs1, immI	Compare $<$ Immediate (Unsigned)	reg[rd] <= (reg[rs1] <$_u$ immI) ? 1 : 0
XORI	**xori** rd, rs1, immI	Xor Immediate	reg[rd] <= reg[rs1] ^ immI
ORI	**ori** rd, rs1, immI	Or Immediate	reg[rd] <= reg[rs1] | immI
ANDI	**andi** rd, rs1, immI	And Immediate	reg[rd] <= reg[rs1] & immI
SLLI	**slli** rd, rs1, immI	Shift Left Logical Immediate	reg[rd] <= reg[rs1] << immI
SRLI	**srli** rd, rs1, immI	Shift Right Logical Immediate	reg[rd] <= reg[rs1] >>$_u$ immI
SRAI	**srai** rd, rs1, immI	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] >>$_s$ immI
ADD	**add** rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SUB	**sub** rd, rs1, rs2	Subtract	reg[rd] <= reg[rs1] - reg[rs2]
SLL	**sll** rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] << reg[rs2]
SLT	**slt** rd, rs1, rs2	Compare $<$ (Signed)	reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0
SLTU	**sltu** rd, rs1, rs2	Compare $<$ (Unsigned)	reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0
XOR	**xor** rd, rs1, rs2	Xor	reg[rd] <= reg[rs1] ^ reg[rs2]
SRL	**srl** rd, rs1, rs2	Shift Right Logical	reg[rd] <= reg[rs1] >>$_u$ reg[rs2]
SRA	**sra** rd, rs1, rs2	Shift Right Arithmetic	reg[rd] <= reg[rs1] >>$_s$ reg[rs2]
OR	**or** rd, rs1, rs2	Or	reg[rd] <= reg[rs1] | reg[rs2]
AND	**and** rd, rs1, rs2	And	reg[rd] <= reg[rs1] & 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
li rd, constant	Load Immediate	reg[rd] <= constant
mv rd, rs1	Move	reg[rd] <= reg[rs1] + 0
not rd, rs1	Logical Not	reg[rd] <= reg[rs1] ^ -1
neg rd, rs1	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
j label	Jump	pc <= label
jal label **call** label	Jump and Link (with ra)	reg[ra] <= pc + 4 pc <= label
jr rs	Jump Register	pc <= reg[rs1] & ~1
jalr rs	Jump and Link Register (with ra)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
ret	Return from Subroutine	pc <= reg[ra]
bgt rs1, rs2, label	Branch $>$ (Signed)	pc <= (reg[rs1] >$_s$ reg[rs2]) ? label : pc + 4
ble rs1, rs2, label	Branch $\leq$ (Signed)	pc <= (reg[rs1] <=$_s$ reg[rs2]) ? label : pc + 4
bgtu rs1, rs2, label	Branch $>$ (Unsigned)	pc <= (reg[rs1] >$_s$ reg[rs2]) ? label : pc + 4
bleu rs1, rs2, label	Branch $\leq$ (Unsigned)	pc <= (reg[rs1] <=$_s$ reg[rs2]) ? label : pc + 4
beqz rs1, label	Branch $= 0$	pc <= (reg[rs1] == 0) ? label : pc + 4
bnez rs1, label	Branch $\neq 0$	pc <= (reg[rs1] != 0) ? label : pc + 4
bltz rs1, label	Branch < 0 (Signed)	pc <= (reg[rs1] <$_s$ 0) ? label : pc + 4
bgez rs1, label	Branch ≥ 0 (Signed)	pc <= (reg[rs1] >=$_s$ 0) ? label : pc + 4
bgtz rs1, label	Branch > 0 (Signed)	pc <= (reg[rs1] >$_s$ 0) ? label : pc + 4
blez rs1, label	Branch ≤ 0 (Signed)	pc <= (reg[rs1] <=$_s$ 0) ? label : pc + 4

MIT 6.004 ISA Reference Card: Calling Convention

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

8 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. You can always run a program using the CLI version by running `rv_sim program` if you just want to check the result quickly, but it does not have the same debugging capabilities as the GUI version.

Before getting started, make sure that you have run '`make`' successfully. For lab 2, check that files **`factorial.vmh`** and **`quicksort.vmh`** exist.

8.1 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 8.2 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.

The box at the bottom of the window (with an '`Enter`' button next to it) is used for sending keystrokes to your program. If your program uses MMIO to read from the keyboard, the simulator will automatically pause when you try to read from the input MMIO channel. You can type the input you want to send to your program in the box and click the '`Enter`' button. The input will be queued up for your program to use but the simulator remains paused until you click '`Run`' or '`Step`' to resume it.

Click '`Stop Program`' to quit the execution of the current program. After that, the machine is ready to load another program.

8.2 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.

`quicksort.dump` can be very useful to debug **`quicksort.S`**. For instance, if you compile without any modification to **`quicksort.S`**, and search '`<sort>:`' in the generated **`quicksort.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`**.