Lab 1: Instruction-Level RISC-V Simulator

Assigned: Tuesday 1/17; Due **Monday 2/13** (midnight)

Instructor: James E. Stine, Jr.

1. Introduction

For this assignment, you will write a C program which is an instruction-level simulator for a limited subset of the RISC-V instruction set. This instruction-level simulator will model the behavior of each instruction, and will allow the user to run RISC-V programs and see their outputs. As we mentioned in class, these types of queuing-type simulators are important in determining good choices in microarchitecture design and subsequently its architecture.

This lab's objective is really accomplishes two tasks. First, it introduces you to software and the process in running code. It introduces you to the basics of compiling in C and what simulators mean. Second, it will introduce you to the RISC-V ISA. All computer architects, programmers, and digitial designers know how processors work by reading the reference manual. And, this lab, will certainly introduce you to this as well as being your first in-depth lab in this course.

The simulator will process an input file that contains an RISC-V RV32i program. Each line of the input file corresponds to a single RISC-V instruction written as a hexadecimal string. For example, `00208f33` is the hexadecimal representation of `add t5, ra, sp`. We will provide several input files. But you should also create additional input files in order to test your simulator more comprehensively.

The simulator will execute the input program one instruction at a time. After each instruction, the simulator will modify the RISC-V architectural state: values stored in registers and memory. The simulator is partitioned into two main sections: the (1) shell and the (2) simulation routine. Your job is to implement the simulation routine.

The source code for the lab are provided through GitHub Spring Repository linked through Canvas, we provide two files (`shell.c` and `shell.h`) that already implement the shell. There is a third file (`sim.c` and `isa.h`) where you will implement the simulator routines.

2. Compilation Environment

We will use the C language for our programming language and although many of you may not have had experience in this, it is similar in terms of its usage as Java. Also, this lab gives you the underlying framework and more complicated items and you are left to implementing architectural items. A wonderful introduction to the C programming language is available in the text by Y. Patt and S. Patel, which is also available on reserve in the Edmon Low Library [1]. More terse and complete texts are available in other texts including the one of the original texts on the subject by the inventors of C, Brian Kernighan and Dennis Ritchie [2]. There is also a great reference from your ECEN 3233 textbook in Appendix C [3][1].

The problem with programming languages, like C and Java, is sometimes you do not know how to compile someone else's program. For Verilog in Lab 0 we solved this with the DO file. With the DO file you can basically compile and simulate any program from another user provided a DO file exists (i.e., `vsim -do file.do`). Most programming languages solve this similarly with a `Makefile`. A Makefile is a file that tells your system how to compile your files for a specific programming language as well as provide any needed command-line arguments necessary to have the program compile correctly. In fact, you can actually use a `Makefile` with any type of compiled or interpreted language, even Verilog. For this lab, a `Makfile` is provided in the repository and all you have to do to compile the program is type `make` provided the files are located in your subdirectory that you are working in. To compile the initial skeleton, go to the subdirectory where you files exist and type `make`.

There are many C compilers available, but you should have an RISC-V tool chain that you can install on the Windows boxes in Endeavor 350. You can also download from the link on Canvas and install on your laptop/desktops (which is highly encouraged). Although there are many ways to interact with the compiler,

[1] Available on Canvas

I would recommend using the command line interface. Its easier to use the Command Line or Power Shell through Windows 11. There are other methods, but this is probably the most recommended.

3. RISC-V sim

The goal of this lab is to get you to think about how to model the RISC-V ISA. I have tried to give you a good framework for the simulator written in C. The RISC-V sim simulator is broken into two main C files: `shell.c` and `sim.c`. The first file handles the simulation from the user's point of view (i.e., the interface). The second file is the main simulator that makes sure instructions are modeled corrected. Each instruction is decoded and then `isa.h` handles the emulation in a function.

3.1 The Shell

The purpose of the shell is to provide the user with commands to control the execution of the simulator. The shell accepts one or more program files as command line arguments and loads them into the memory image. In order to extract information from the simulator, a file named `dumpsim` will be created to hold information requested from the simulator. The shell supports the following commands:

- go: simulate the program until it indicates that the simulator should halt. (As we define below, this is when a SWI instruction is executed with a value of `0x0A`.)
- run <n>: simulate the execution of the machine for n instructions.
- mdump <low> <high>: dump the contents of memory, from location low to location high to the screen and the dump file (dumpsim).
- rdump: dump the current instruction count, the contents of R0 – R14, R15 (PC), and the CPSR to the screen and the dump file (dumpsim).
- input `reg_num` `reg_val`: set general purpose register `reg_num` to value `reg_val`.
- ?: print out a list of all shell commands.
- quit: quit the shell.

3.2 The Simulation Routine

The simulation routine carries out the instruction-level simulation of the input RISC-V program. During the execution of an instruction, the simulator should take the current architectural state and modify it according to the ISA description of the instruction in the RISC-V Architecture Reference Manual that is provided on the course website. The architectural state includes the the general purpose registers and the memory image. The state is contained in the following global variables:

```
#define RISC-V_REGS 32

typedef struct CPU_State {
  uint32_t REGS[RISC-V_REGS]; /* register file. */
  uint32_t PC; /* program counter */
} CPU_State;

extern CPU_State STATE_CURRENT, STATE_NEXT;
int RUN_BIT;
```

Furthermore, the simulator models the simulated system's memory. You need to use the following two functions, which we provide, to access the simulated memory:

```
uint32_t mem_read_32(uint32_t address);
void     mem_write_32(uint32_t address, uint32_t value);
```

Note that in the RISC-V architecture, memory is byte-addressable. Furthermore, we will implement a little-endian architecture. This means that machine words (32 bits) are stored with the least-significant byte at the lowest address, and the most- significant byte at the highest address. To implement loads and stores of 8-bit values, you will need to use these 32-bit memory access primitives (hint: be sure to modify only the appropriate part of a 32-bit word!).

In particular, you should call `mem_read_32` and `mem_write_32` with only 32-bit-aligned addresses (i.e., the bottom two bits of the address should be zero). The simulator skeleton that we provide includes a function named `process_instruction()` in the file `sim.c`. This function is called by the shell to simulate one machine instruction. You can also write additional functions to make the simulation modular (Keep in mind that you will probably be using the code that you write in later labs in order to validate your work). We suggest spending time to make your code easy to read and understand, for your own benefit.

3.3 What do you need to do?

Your job is to implement the process instruction() function in `sim.c`. The process instruction() function should be able to simulate the instruction-level execution of the following RISC-V instructions:

ADD	ADDI	AND	ANDI	AUIPC	BEQ
BGE	BGEU	BLT	BLTU	BNE	JAL
JALR	LB	LBU	LH	LHU	LUI
LW	OR	ORI	SB	SH	SLL
SLLI	SLT	SLTI	SLTIU	SLTU	SRA
SRAI	SRL	SRLI	SUB	SW	XOR
XORI	ECALL				

For implementing these instructions, your tasks will be the following. First, implement each of these instructions as specified within the RISC-V Architecture Reference Manual accurately and completely. Each instruction should be compatible with conditional execution as described by the RISC-V manual.

It is important to note that for the ECALL instruction, you only need to implement the following behavior: if the bottom byte of the instruction's value is 0x73 (decimal 115) when ECALL is executed, then the go command should stop its simulation loop and return to the simulator shell's prompt[2]. If the bottom byte is any other value, the instruction should have no effect. No registers are modified in either case, except that PC is incremented to the next instruction as usual. The process instruction() function that you write should cause the main simulation loop to terminate by setting the global variable RUN BIT to 0. Also of note, you should not worry about implementing any mode changes or register switches on a ECALL. Thus, you must only worry about one set of registers.

NOTE: RISC-V assumes that the PC value is actually equal to $PC + 4$ when the instruction at PC is being executed. This is because of RISC-V's pipeline which keeps three instructions in flight at once. However, we do not ask you to keep the incremented PC value. This means that whenever you use an operation that requires the PC value (e.g., BNE), you must use PC+4 as the base offset.

The accuracy of your simulator is your main priority. Specifically, make sure the architectural state is correctly updated after the execution of each instruction. We will test your simulator with many input programs (some provided with the handout, some not) in order to ensure that each instruction is simulated correctly.

In order to test that your simulator is working correctly, you should run the input programs we provide you with and also write one or more programs using all of the required RISC-V instructions that are listed in the table above, and execute them one instruction at a time (run 1). You can use the rdump or mdump command to verify that the state of the machine is updated correctly after the execution of each instruction.

While the table appears to have many instructions, there are actually only a few unique instruction behaviors with a number of minor variations. You should tackle the instructions in groups: I Instructions, S Instructions, B instructions, and so on. The RISC-V Architecture Reference Manual contains the official definition for each instruction in this table (except for ECALL, for which we provide a restricted definition above). Please implement only the `RV32i` behavior of the instructions.

[2] I might have implemented this for you :)

Finally, note that your simulator does not have to handle instructions that we do not include in the table above, or any other invalid instructions. We will only test your simulator with valid code that uses the instructions listed above. However, as noted below, adding more instructions will give you some bonus points.

3.4 Some Guidance

In order to give you some guidance and help you get started, since for many this will be something different, much of the simulator is given to you. Also, dealing with a larger program might be difficult, so we tried to avoid frustration with writing code from scratch. Most of the main functionality within the simulator works well and should be functional (i.e., `sim.c` and `shell.c`). However, if you notice within `sim.c` I put an example of the instruction call for you. For example, here is the call for `ADD`:

```
if(!strcmp(d_opcode,"0110011")) {
  printf("--- This is an ADD instruction. \n");
  ADD(Rd, Rs1, Rs2, Funct3);
  return 0;
}
```

These calls will be processed from `isa.h` and you can replicate the piece of code to simulate the other instructions. I also gave you one free instruction that I implemented (i.e., `ADD`) within the code.

The RISC-V instruction set does not utilize any conditional flags, so implementing the simulator should be reliatve straightforward if you understand each instruction's behavior. This field (bits `6:0`) determines the encoding under for which an instruction is to be executed.

3.5 Lab Files

In our lab repository, you will find the source code distribution with the subdirectories `src/` and `testing/`. In `src/`, we are providing you with the simulator skeleton as described above. You can compile the simulator with the provided Makefile. In `testing/`, we have written and compiled some input files for you. However, you could write more input files in order to be confident that your simulator is correct and that you simulate all the instructions you are required to implement.

3.6 Resources

If you have not done so already, we recommend that you become familiar with the RISC-V ISA and how the GNU compiler functions. The RISC-V instruction set architecture is defined in the referenced manual that we have provided on the course website. However, the best reference is the list of RISC-V instructions in Appendix B in your ECEN 3233 textbook [3]. This Appendix is great, because it lists all the instructions in a concise manner and also provides a pseudo-language to understand each instruction's operation called Register Transfer Language (RTL). We may post additional resources (clarifications, etc.) as required on the course website. Finally, please do not hesitate to ask the TAs for help if you become stuck or if something is unclear! Take advantage of Slack, office hours, and lab sections.

4. Compiling Code

As stated previous, several examples are given in the `input\` directory of the repository. Both assembly and hexadecimal versions are given. To compile the program into hex, you want to use the provided Makefile that compiles your program and outputs it into a hex file utilizing the elf2hex program.

GNU tools are important to the use of computers and have been instrumental in great advances in microarchitecture. As discussed in class, we will also use GNU tools for our work in this class. All of the RISC-V tools will have a program prefix of `riscv64-unknown-elf-`. That is, if you want to the GNU binutils program `gcc`, you would type `riscv64-unknown-elf-gcc`.

Getting GNU tools usually involves compiling them into binaries. This can take time and can also be difficult for those not familiar with GNU tools. Although you can use the compiled version of the files in the

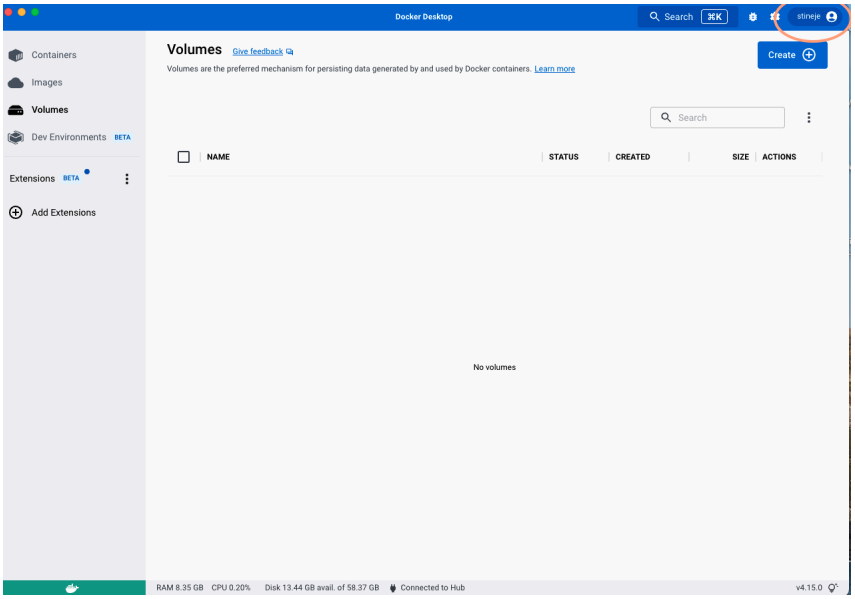

Figure 1: Docker Application Screen

`testing` directory of your repository, you can compile others using docker. For those that wish to compile any subsequent code, we will use Docker to help us compiler the tools. It is important to note that compiling programs with Docker is not necessary.

Docker is a set of tools to run applications in a prepackaged container including all of the operating system support required. Wally offers a 30GB container image with the open-source tools pre-installed from Section D.1. In particular, using the container solves the long build time for gcc and the fussy installation of sail. The container runs on any platform supporting Docker, including Windows and Mac as well as Linux.

You will need the Docker application that you can download from `http://www.docker.io`. Docker can be run on most operating systems, including Linux, Windows, and Mac. The Wally Docker container is hosted at DockerHub (http://docker.io). A user must sign up for a free account at docker.io. To login, click the Username at the top right hand corner of the Docker screen as shown in Figure 1. This application allows you to login to our

Once Docker is installed, a user can pull the Wally container image. The user must sign up for a free account at docker.io, and will be prompted for the credentials when running docker login.

```
$ docker login docker.io
$ docker pull docker.io/wallysoc/wally-docker:latest
```

Docker sets up all the RISC-V software in the same location of `/opt/riscv` as the cad user as discussed previously. This shared directory is called `$RISCV`. This environmental variable should also be set up within the Docker container automatically and ready to use once the container is run. It is important to understand that Docker containers are self-contained, and any data created within your container is lost when you exit the container. Therefore, you may want to ask your TA to help you transfer off your container.

To allow your docker container to work in your home directory it is advisable to mount your home drive. You can do this by adding the `-v` argument to running your docker. This particulare command is run for Mac OS and maps my home drive (i.e., `/Users/stineje`) to `/home/stineje`. That way, if you change your directory to this directory, you should see your Mac OS home drive that is mounted. You can also run this on Windows and mount one of your Windows directories, as well.

```
$ docker run -it -v /Users/stineje:/home/stineje -p 8080:8080 docker.io/wallysoc/wally-docker
```

Docker is useful in getting access to the compiler and other RISC-V tools; however, as stated earlier it is not necessary. I think you will find it useful to get access to the RISC-V toolchain, so I encourage you to try this with Docker. Just a friendly reminder, pleaes remember to work in your mounted drive and not

the space within Docker. This way, if you write any files, they will be stored to your hard drive. Otherwise, once you exit the docker, the files associated with your docker will be removed. If you are confused about this, please ask me.

5. Handin

You should electronically hand in your code (all files in the src/ directory) into Canvas through its dropbox. Please contact the TAs for more help. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in a inputs/ subdirectory. If you feel the need to describe any additional aspects of your design in detail, please include these in a separate README. Make sure that you test your simulator extensively with a suite of test cases so that you are confident that you have implemented all instructions correctly. This is for your benefit in later labs!

If you get stuck understanding the code or figuring out what is wrong, it might help to either use a debugger or print out some variables. C has an excellent debugger called `gdb` and although it is a little hard to first get started with, it is extremely powerful. There is a tutorial posted on Canvas that might help. The other option is print out certain points of the code to the screen with a `printf()` statement. This is not a great technique for debugging, but sometimes it can solve problems quickly.

It is highly recommended you try to tackle things early and get most of the work done during the first week in lab. It is also important to split the work among your team to help get larger things done. As I will say for this and remaining labs that much of the work in these labs is involved. Therefore, do not procrastinate and get to it. This lab also has many practical and real world implications and can help you understand how to use C and programming languages.

In order to grade the lab, we will use some unknown assembly code that you will not see. To get the maximum grade for this lab, your simulator must be able to simulate all the instructions asked in Section 3.3. Your job is to really try to make sure your simulator covers all possible instructions that this unknown code may utilize.

5.1 Extra Credit

There are many opportunities for improving the simulator. Adding extra features or more instruction support. These extra items can earn you extra points that may help you later in the semester. Remember, if you try any extra credit or add any extra functionality that it should be documented in your lab report as well as upload to GitLab. As with any extra credit, this should only be attempted provided you have extra time with this course as well as your other courses.

References

[1] Y. N. Patt and S. J. Patel, *Introduction to Computing Systems: From Bits and Gates to C and Beyond.* New York, NY, USA: McGraw-Hill, Inc., 2003.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1978.

[3] S. Harris and D. Harris, *Digital Design and Computer Architecture, RISC-V Edition.* Elsevier Science, 2021.