ECE 486/586
Winter 2025
Final Project
Simple RISC-V ISA Simulator

Introduction

Simulation has a number of uses, but simulation at the instruction set architecture level is particularly useful for computer architects, designers, and software developers. Without needing to simulate at a deeper machine organization (e.g. microarchitecture) or register-transfer level, you can explore the impact of instruction set architecture changes on CPI, look at the instruction counts for various program mixes (or benchmarks), generate trace files useful for analyzing and comparing results of decisions in branch predictor algorithms and cache designs, and of course, develop and debug compilers and system software before prototype hardware is available.

For this project, you are to write an instruction set architecture (ISA) level simulator for RISC-V. You must write your simulator in C or C++.

You are only responsible for implementing the RV32I Base Integer Instruction Set as described in Chapter 2 of the RISC-V ISA Instruction Set Manual, Volume I (https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf). You can ignore the memory ordering instructions, the environment call and breakpoints, and hint instructions (Sections 2.7 – 2.9).

Rather than read an ELF executable file (which is complex and would need to contain libraries, etc.) your simulator will read a simple ASCII file that represents a memory image. You can write simple assembly code and assemble it with the RISC-V GNU toolchain assembler (invoke with **`rvas`** alias you defined in **`.bashrc`** in earlier homework). You can also write (simple) C code and use the RISC-V GNU compiler (invoke with **`rvgcc -fpic -march=rv32i -mabi=ilp32 -S`**). Because you're not linking to produce an ELF executable file, you'll be limited to a single object file and can't call any functions not defined in the single source file. You also won't be able to use global or static variables unless you write assembly code (and even then it may require you to be a little tricky). You can assume (and enforce) a constraint that all programs be 64KB or smaller.

Your simulator should accept the following three input parameters. Changing parameters should not require recompilation. Ideally for Linux programs invoked from the command line, that means reading command line arguments (argc, argv). Worst case, you can prompt for them after program invocation:

Input memory image file (default to program.mem)
Starting address (default to 0)
Stack address (default to 65536)

The only registers the simulator should initialize or load (except as a result of simulating the loaded code) are the **`sp`**, **`pc`**, and **`ra`** (which you should initialize to 0). Your simulation should continue until you encounter a **`jr ra`** instruction with **`ra`** == 0. This will make life easy if you're compiling C code. First define **`main()`** and define any other functions following **`main()`**. This will ensure that your code

begins at location 0 and the return from **`main()`** (effected by a **`jr ra)`** will cause simulation to terminate.

If you're writing snippets of assembly code you can force the simulation to stop by including an explicit **`jr ra`** instruction at the end of your code. Because RISC-V defines an instruction word of all 0s to be a trap, you should also use an attempt to execute an instruction of all 0s as an indication to terminate the simulation. Note: this is helpful for catching bugs – a branch to an incorrect location in unused memory will result in an immediate trap.

Extra credit

You can get extra credit for the following:

- Implement the **`ecall`** instruction to provide at least the three system calls below. The contents of the register **`a7`** determines which system call to invoke. In general, syscall arguments are passed in registers **`a0`** through **`a5`** and values are returned in **`a0`**.

		Input registers			Output register
a7	**System Call**	**a0**	**a1**	**a2**	**a0**
63	read	File descriptor (0 for STDIN)	Address of buffer	Maximum number characters to store	Number of bytes read (-1 if error)
64	write	File descriptor (1 for STDOUT)	Address of buffer	Length of string	Number of bytes written (-1 if error)
94	exit	Return code (0 for OK)			none

- Provide ability to single step and print contents of register, memory, display current instruction. Optionally provide ability to set breakpoint, "watch" a register or memory location.
- As above but with a GUI.
- Instruction set extensions (e.g. M (integer multiply/divide), F (floating point)).

Format of Memory Image File

The memory image file is an ASCII file consisting of lines with a (hexadecimal) memory address followed by a colon followed by whitespace (e.g. spaces or tabs) and the hexadecimal contents of that memory location. An example is below:

```
0:    fd010113
4:    02812623
8:    03010413
c:    fca42e23
```

You can generate this file from an object file with the following (where **prog.o** is the object file and **prog.mem** is the memory image file):

```
% rvobjdump -d prog.o | grep -o  '^[[:blank:]]*[[:xdigit:]]*:[[:blank:]][[:xdigit:]]*' > prog.mem
```

Simulator Output

Your simulator should run in two modes. In verbose mode it should print the PC and hexadecimal value of each instruction as it's fetched along with the contents (in hexadecimal) of each register *after* the instruction's execution. In silent mode it should print the PC of the final instruction and hexadecimal value of each register only at the end of the simulation. You can, of course, conditionally print additional debug information but it must be possible to suppress it during the demo without recompiling. That doesn't mean that it needs to be a runtime option – just that the version you compile for the demo needs to be capable of not printing debug information.

Grading

Your project will be graded as follows:

50%	Correctness (generates complete and correct output on all test cases)
25%	Code quality (organized, readable, maintainable, robust)
20%	Testing (sensible test strategy articulated and executed)
5%	Project report (concise summary of design and testing, presentation of results)

Plagiarism

Plagiarism on this assignment will not be tolerated. All code must be your own. You may make use of other tools to validate your assumptions about the RISC-V architecture but you may not incorporate any outside code in your project. "Borrowed" code will result in a zero for all team members for the assignment.

Suggestions

- Start early
- Consider incremental development
    - Get the basic memory loading working
    - Get input parameter code working
    - Get the fetch/decode/execute framework working with ability to examine registers, etc.
    - Decode each instruction type
    - Implement and debug one instruction of each type
    - Complete the remaining instructions for each instruction type
- Have a *written* test strategy and plan
    - Create an outline that covers all functionality (e.g. types of instructions, addressing)
    - Write a 1-2 line description of each testcase
    - Include the setup conditions, the functionality being tested, the anticipated result
    - Have someone other than the coder of the functionality be the one to write the test case
    - Verify that the results are as expected

- Include useful debugging information
    - Consider debugging modes/levels
    - Have ability to turn off debugging output!
    - Useful, but not too verbose