CS2323 - Lab 4 - RISC V Simulator

Shiven Bajpai - AI24BTECH11030

October 1, 2024

Contents

1 Preface

The objective of this lab project was to extend the RISC-V assembler made in the previous lab project by making a Simulator that can run said code. I chose to add some additional flair to this project in the form of a Text-UI built using the ncurses terminal control library. The full features of my simluator are as follows

- Visual representation of assembled and source code next to each other, sectioned by labels. The current line being executed is highlighted.
- Live view of registers that can be seen while running or stepping, with a live highlighting of the last change to register file.
- User can view the state of the memory that updates live as the program runs
- The user can specify breakpoints explicity using the `break` command, or breakpoints can be implied using the `ebreak` instruction, in either case the program stops execution at the breakpoint
- The simulator is built in a highly modular fashion. This means that the assembler, backend and frontend are largely interdependent, and only require a simple interface of a few functions to be exposed. So developing a new frontend (say a proper GUI) or a new simulator backend (Say a pipelined multi-stage processor implementation) can be done without having to make any significant changes to the rest of the project.
- Most errors are gracefully handled and shown within the TUI without exiting the program, so the user can recover from any mistakes and continue.
- There is an extra command beyond the requirements of the project called `reset` that resets the simulator to its state when the file was initially loaded. This is useful because the only other way to reset would be to reload the file, which would override the set breakpoints. Using `reset` means you do not lose any breakpoints, and also re-compilation is avoided.
- The program checks for Invalid memory access by the RISC-V instructions and ceases execution when such an instruction is encountered. Modifying the text section is normally not allowed, however if an experimental flag `--self-modifying-code` is set then the simulator allows the program to write to the text section, thus allowing it to modify itself. Note that since the feature is experimental, the dissassembled representation is not guaranteed to be correct if this flag is specified, as that is read from the input file rather than being actual dissassembled code. The behaviour of the simulator in reponse to invalid instructions is also not defined.
- Lastly, although not a technical feature, the TUI gives it a cool hacker-esque vibe.

The project was quite fun, I often found myself procrastinating on my other assignments by working on this project instead. I learnt quite a bit about ncurses and also C libraries in general as this was the first time I was using an external one. I would like to thank the professor for assigning such a fun and engaging project, but I do wish it had greater weightage in the final grade

2 Overview

2.1 Breaking down the problem

The simulator has four main parts. The assembler that is called when a file is loaded, The backend implementation that does the actual simulation, The Frontend that displays information and takes input from the user and the control code that serves coordinates these three and handles errors.

The overall program flow goes as follows,

<Flowchart of high level program flow>

2.2 Application is not responding!

Before we can talk about the technical parts in more detail, we need to talk about something much more important. The UI here's needs to remain responsive while the simulator is running. This means our program needs to be structured in a way such that it can asynchronously run the frontend and the backend. One such way to do this would be to make two threads. One for the frontend and the another for everything else. However in this project for the sake of simplicity, I have used a single thread and manually implemented a form of polling to keep the UI responsive.

3 Adapting the assembler

4 Writing the Backend

The simulator itself is relatively straightforward. It just has to load an instruction, decode it, and execute it. The validity of instructions is already guaranteed by the compiler (provided the `--self-modifying-code` flag is not set, but since it is just an experimental flag I decided not to implement instruction validation for it.). Then we increment the PC and done! We can define the function that implements this process as the `step()` and use it for the `step` command. The `run` command then can be implemented by simply calling `step()` over and over again. Make an array to serve as the simulated memory and registers. Add a few conditions to test for breakpoints, invalid memory access and we're done.

- Keeping a stacktrace - Step() and run() - breakpoints - SMC experimental feature

5 Loading in a file

- resetting

6 Frontend

- Keep it brief because it is not the focus of this project

7 Appendices

7.1 Build instructions

7.2 How to use the sim

7.3 Folder structure

7.4 Possible ways to improve this simulator