CS2323 - Lab 4 - RISC V Simulator

Shiven Bajpai - AI24BTECH11030

November 11, 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 re-compilation is also 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.
- Has keyboard shortcuts for common actions
- 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 back-end implementation that does the actual simulation, The Frontend that displays information and takes input from the user and the control code that coordinates these three and handles errors.

The overall program flow goes as follows,

- The program begins execution
- The UI is initialized and various memory allocations are performed
- The simulator waits for commands
- If a load instruction is issued then it trys to assemble the input.
- The simulator steps, runs and stops running the code as instructed. While the code runs, the frontend is continously updated.
- If a new load command is issued, then the simulator once again tries to assemble this input and replaces the loaded code with the new code if successful.
- If an exit command is recieved, It exits gracefully.

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 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

Several changes need to be made to the assembler made in the previous project to make it suitable for this project.

1. The main function needs to be adapted to take in buffers as arguments, instead of reading file names from command line arguments and opening them itself.
2. Similarly, all other functions will need to be adapted to use buffers instead of file pointers.
3. The main function will also need to be given pointers to where it can store objects with information such as label positions which was previously not required once the assembling process was done
4. An extra pre-processing step is required to read in the .data segment, and only pass on the .text segment to original first pass of the assembler. All error reporting needs to be

updated to account for the line number offset due to the data segment. If no label like .text or .data is encountered, this step does nothing

5. All error reporting needs to be changed to use the `show_error()` funtion defined in the frontend, instead of printing to stdout.

Other than this, I updated the custom types `struct label_index` and `struct vec` to have some more utility functions. I also changed the type of the values stored in `struct vec` to be `uint64_t` instead of `int` so that I can use it to store memory locations without any concerns of overflow.

4 Writing the Backend

The simulator itself is relatively straightforward. It just has to load an instruction, decode it, and execute it. Then we increment the PC and done! 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.)

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 PC being moved outside of the text segment and we're done.

The `run()` function runs intructions only at a pace of about 5 per second. This is done using a timer and components from the builtin `ftime` library. While waiting for the delay, the program keeps updating the frontend to ensure the application remains responsive. It checks between every frame if it is time to run the next instruction yet or not.

The implementation of the individual instructions and its parsing is done via a pair of large switch-case blocks. The details are trivial and I won't discuss it here.

To maintain a stacktrace, we create another `struct vec`. At the start of the program, the location of the first instruction is pushed onto this stack. Whenever an instruction is executed, the line number of the last entry is updated. Every time a jump instruction is taken, a new entry is pushed onto this stack and every time a `jalr` instruction is encountered, an entry is popped off of the stack. Of course, this only works when jal and jalr are only used according to the convention and only for function calls.

Implementing breakpoints is also trivial, We just maintain a list of all breakpoint positions. On every instructino execution, we check if the next instruction has a breakpoint on it. If it does, then we stop the execution loop.

5 Loading in a file

Loading in a file is a series of simple steps.

1. Find the file, if not found, raise error

2. Allocate buffers required to hold the results of assembly, Then transfer control to the assembler.

3. If the assembler failed, report the error and abort. Otherwise, Reset the stack and breakpoints, Copy the output of the assembler into the simulated memory, Set the PC and the registers to 0. Pass the cleaned code to the frontend.

4. If there is no label before the first, insert the label `main` before the first line. Also remove any cases of multiple labels for the same line, keeping only the first.

Since the simulator described in the problem statement has no method to revert the simulator to how it was before execution, the user command `load` would have to be used for this purpose. The problem with this is that it clears any breakpoints that are set, which may not be desired. It also reruns the assembler which is not necessary.

So I made another user command `reset` that does the same things as `load` with the differences that it a) does not clear the breakpoints set and b) does not recompile the code, instead using the output from the previous run of the assembler.

6 Frontend

The frontend is designed to adapt to the size of the terminal by using the size of the terminal window either directly or indirectly in all of the position calculations, If the window is too small to show what needs to be shown, It skips drawing that section until the window is resized.

Since the frontend is not a part of the project requirements, I wont go into much detail on its implementation, But a brief description is as follows:

The frontend is made using the `ncurses` terminal control library. The `frontend_update()` is called regularly from the main loop to take input and redraw the screen.

Any command sent to the application is held by the frontend, as well as parsed and processed by the frontend itself, It is only passed to higher control code if it requires code execution outside the frontend (e.g the `run` or `break` commands). Commands such as `mem` and `regs` are handled by the frontend itself.

Any data that needs to be shown is read directly from the backend's copy of the same if possible. The top level control code is responsible for ensuring the pointers for the same are set correctly using a series of `get_<>_pointer()` and `set_<>_pointer()` functions exposed by both the frontend and the backend.

The "cleaned code" and index of labels that is created as an output of assembly process is used for the display of code divided into sections by labels in the Code pane. The calculation for the position of each line after taking into account sectioning due to labels is slightly complicated and would be inefficient to repeat on every draw of the frame, so all the positions are calculated only once when a file is loaded in and then re-used on every draw.

7 Testing

For testing I used the sample cases given in the problem statement, along with a fibonacci calculation program, the program from GCD lab assignment and all testcases I had from my assembler. These are all available in the `\test` directory. Additionally I used the open source tool `valgrind` to search for memory bugs, It was a huge timesaver.

A Appendices

A.1 Build instructions

To build this project, the development libraries for ncurses need to be installed using the command:

```
sudo apt install libncurses-dev
```

Then build the project by running `make`
The binary is generated in `/bin`

A.2 Guide on how to use the simulator

After running the simulator, the UI loads up in the terminal. After this all operations are done using a set of commands:

`load <filename>`
Attempts to load a file of source code. The path is assumed to be relative unless it is a full path.

`run`
Runs the code from the current line until the end or next breakpoint. Executes about 5 instructions per second. Keyboard Shortcut: F5

`step`
Runs one instruction, Keyboard Shortcut: F8

Tip: Holding down F8 runs the program a lot faster than `run` does

`stop`
Stops program execution, only meaningful after `run`. Keyboard Shortcut: F6

`reset`
Resets the simulator to the state it was in when the program was loaded, but does not remove any breakpoints and avoids recompilation

`regs`
Is redundant since registers are always visible. Still implemented to give you an error saying the same because it was required in the problem statement

`show-stack`
Shows the stacktrace in the memory/stack-trace pane. If the stack-trace is already being shown then it does nothing

`mem <address>`
Shows the memory in the memory/stack-trace pane. The `<count>` argument from the problem statement is not implemented as the ability to scroll on the memory pane makes it redundant.

`break <line>`
Inserts a breakpoint at the specified line number, or removes it if already set. Line number must be as displayed in the code pane.

`exit`
Closes the simulator. Keyboard Shortcut: F1

Always use the `exit` command, trying to exit using Ctrl + C will cause problems as the terminal may get misconfigured and text may not render properly. If this occurs, you have to either restart the terminal or run the `reset` terminal command.

You can scroll in the code, memory and stack pane pane.

A.3 Ways that this simulator can be improved

There are several ways in which this simulator can be significantly improved, some of them dont even require significant changes. These are changes that I would've made if I had more time:

- Adding something resembling debugging symbols so that the stacktrace can show the arguments of each function call, displayed in the correct type, would make the stack trace a lot more useful.
- Adding pipelining to the simulator would be relatively simple. In code it would simply involve chopping up the step function according to the different stages and then ordering these stages in reverse order. Additionally some stalling/forwarding logic will be required.
- An indicator could be added at the bottom of the memory pane to show the current Stack size and percentage of memory being used
- ELF file parsing - Just to learn how ELF files are structured.
- Making a GUI instead of a TUI - Just to learn how to make GUIs

A.4 Project folder structure

```
/
 +-- bin
 +-- build
 +-- src
 |   +-- assembler             (files from previous Lab assignment)
 |   |   +-- assembler.c
 |   |   +-- assembler.h
 |   |   +-- index.c
 |   |   +-- index.h
 |   |   +-- translator.c
 |   |   +-- translator.h
 |   |   +-- vec.c
 |   |   +-- vec.h
 |   +-- backend               (implementation of the simulator)
 |   |   +-- backend.c
 |   |   +-- backend.h
 |   |   +-- memory.c
 |   |   +-- memory.h
 |   |   +-- stacktrace.c
 |   |   +-- stacktrace.h
 |   +-- frontend              (ncurses frontend)
 |   |   +-- frontend.c
 |   |   +-- frontend.h
 |   +-- main.c                (main loop, initialization, memory management)
 |   +-- globals.c             (some globals)
 |   +-- globals.h
 +-- report
 |   +-- report.tex
 |   +-- report.pdf
 +-- test
 |   +--all_instructions.s
 |   +--assembler_example_four.s
 |   +--assembler_example_one.s
 |   +--assembler_example_three.s
 |   +--assembler_example_two.s
 |   +--Example_One.s
 |   +--Example_Two.s
 |   +--fib.s
 |   +--gcd.s
 |   +--segfault.s
 +-- .gitignore
 +-- LICENSE
 +-- Makefile
 +-- report.pdf
 +-- README.md
```