Danmarks Tekniske Universitet

02155 - Computer Architecture and Engineering Fall 2018

Building a RISC-V ISA Simulator

(Hand-in 3rd of December 2018)

Hørdur Kai Andreasen (s173933)

December 3, 2018

Abstract

This report is a product of Course *02155 Computer Architecturee and Engineering* at the Technical University of Denmark. The aim of the course is to give the student a deep familiarity with computer systems and organization.

As the final project during the course we have built a RISC-V simulator with the base integer subset and the multiplication and division extension, with command-line functionality and a visual debugger built with the ncurses library.

The simulator can be found at the github repository: `https://github.com/publicBugi/RISC-V-Simulator`

Contents

1 Introduction

The goal of this project has been to cement the understanding of computer architecture by building an actual processor. Although a physical processor could have been a fun experience, to save time this is a simulation of one, which *should* be entirely as capable as the real world counterpart. I emphasize should as this is my first programming project made entirely in C, aside from introductory programming courses, and bugs have not been uncommon during this project.

The processor in question is implemented using the RISC-V **I**nstruction **S**et **A**rchitecture, abbrev. ISA. The benefit of this ISA results in a relatively simplistic implementation of a RISC-V processor, due to the straightforward encoding of each instruction. In comparison to other ISA's such as Intel's x86, RISC-V instruction fields are encoded the same across all types. This means that any field, such as *source register 1*, remains in the same position across any instruction type that contains *source register 1*.

The simulator has drawn a lot of inspiration from the Venus RISC-V Simulator that was introduced early on into the course. However I was dissatisfied with a lot of the design choices and found the simulator, although functional, very cumbersome to use. Having an interest in terminal based programs, my personal goal was to implement a visual debugger using the ncurses library.

2 Specification

The simulator is required to simulate at minimum a standard RISC-V32I in software. RISC-V32I consists of a program counter (PC), 32 registers and memory containing a compiled RISC-V program and data. Certain instructions were not required such as *fence*, *ebreak* and *csrrw*.

A basic execution of the simulator consists of reading a binary file containing a RISC-V compiled program that is stored into memory. Subsequently each instruction pointed by the program counter is fetched, decoded and executed until an exit environment call or the program counter passes the size of the program. Detailed specifications can be seen below

Required features

- RISC-V32I Instruction set
- Program Counter
- 32 Registers
- Memory
- Read binary files to memory
- Save binary file containing registers

Additionally I wanted to implement extensions and a visual debugger. Although ambitious, I limited myself to these as I was uncertain the difficulty

of the project, especially in the language I chose (C), and the difficulty in the library I had chosen for the debugger (ncurses).

Additional features

- RISC-V Extensions
- Visual Debugger

3 Design & Implementation

The design of the simulator is at its core very simple. The central loop can be seen in the diagram in fig. 1. The simulator can be reduced to three steps:

Initilization:
When the program is called memory is allocated to all the required data, such as registers and memory. The desired program is read into memory, and if the desired program is not found the program exits before anything else happens.
If the user has chosen the visual debugger, a large amount of setup is required before the terminal is ready. The windows are calculated and allocated, and the design of the visual debugger can be seen in figure. 2. A desired function was for machine code to be translated and viewed in assembly as well, to make the function of a program more clear. This is also allocated and done at this stage. *Note that there is currently an issue with the translation of the first few elements.*

Simulator Loop:
The central loop is the actual processor. It consists of the regular cycle of a processor in terms of **F**etch, **D**ecode, **E**xecution, **M**emory Access and **W**rite back. The difference here is that E, M and W are done in the same step.
If a verbosity or debugger is chosen these fields also require an update. Specifically the debugger halts the processor, such that the

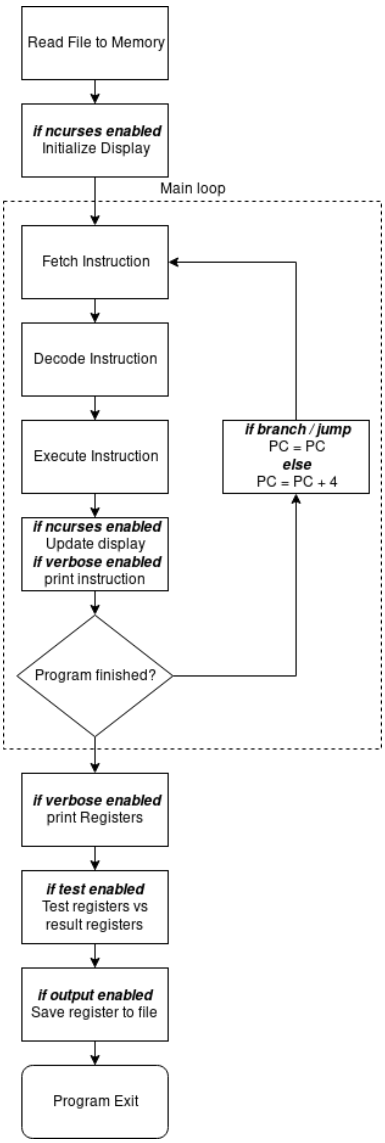

Figure 1: Simulator main loop

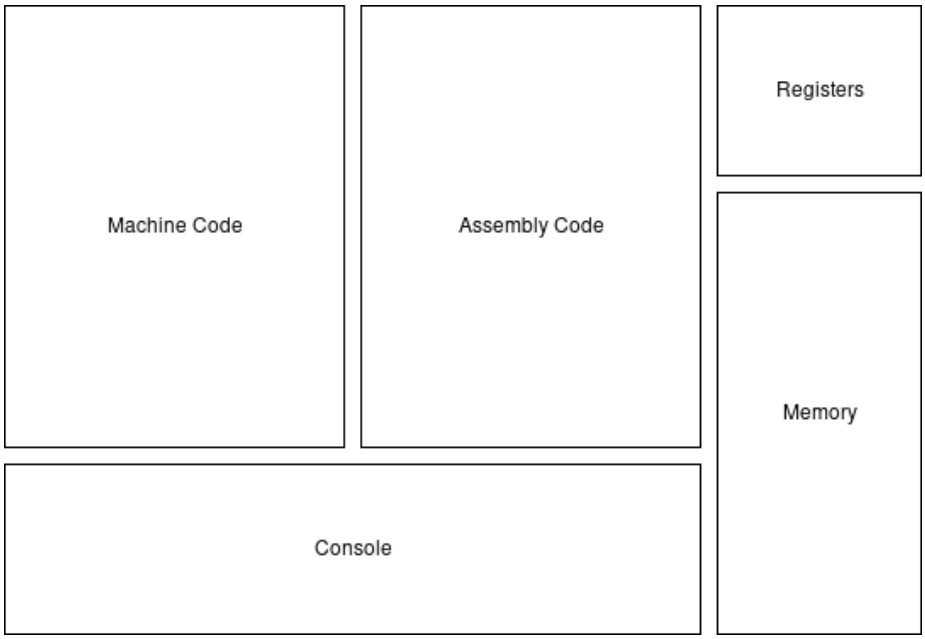

Figure 2: Design of the visual debugger

user can view the changes thoroughly.
There exists a simplified processor control signals that is returned by the execute stage, such that we know wether to update the PC counter or not (in the event of a branching instruction), or to exit the loop in the event of an exit environment call.

Exit stage:
The exiting stage is used to dump results into the terminal and a file if so desired. This is also where the final register state can be tested against a result file, and the result of this printed into the terminal. It is important to note that all the memory allocated on the heap is also freed in this stage.

3.1 Debugging

The entire purpose of the simulator is not just to execute RISC-V compiled programs, but to allow the user to actually view what is happening in the processor. This is why my personal goal was to make a proper debugger. The program has two states: **Command line** and **Visual Debugger**.

Command line:
The command line is the default state of the program. Whenever the program is called, it executes the given file and if verbose prints out any necessary information to do simple debugging. The primary use is intended to be able to execute software quickly with the intent of storing it or as a process in a bigger program.

Visual Debugger:
The visual debugger is called using the flag *-d*. A display similar to the one given in figure 2 is shown containing any information required for the user to step through the program. Upon stepping through the program each window is updated with all new information.
The purpose is to allow the programmer to have a complete overview over all instructions, the memory and current register addresses along with a console similar to the Venus Simulator.

I have based my design on the Venus simulator, although I found a lot to be desired from the visual interface. The venus simulator contains a lot of unused space, and what is used is not used efficiently. As for why I decided on a terminal based design was a personal interest of mine, and I figured it would be the most difficult addition to the project.

4 Testing

To ensure the proper function of the simulator several tests have been supplied. Initial tests are simple and only test initial add instructions, while later tests are larger and complicated, depending on branch instructions, jump instructions and store/load instructions. Individual instructions have also been supplied, that have been used to full extent.

Initial testing was done manually, although this quickly grew tiresome and all testing is now done by comparison by calling *./RISCVSIM* {*input file*} *-t* {*result file*}. It's important to note that the first 3 registers are ignored, as these can vary from simulator to simulator. Results of each test can be found in the github repository.

As of 2nd of December 2018, all officially supplied tests have been tested successfully, however the additional Instruction Tests that were referred have one example not passed. Unsigned division, and subsequent test_random10 which uses unsigned division, has one error in it.
The error lies in the return value of unsigned division of x by 0. The RISC-V ISA Manual says that all bits are to be set to 1, as the return value will be $2^{XLEN} - 1$, XLEN in this case being 32. The test, and Venus, insist the return value be x.

I decided not to follow the other simulators and trust the manual. This however results in errors in these two tests.

The final tests were done with the T.A on 26th of November. Initially I did not pass test 1 and 14, but this was later corrected as C likes to sign extend at random and I had forgotten to save the program to memory.

5 Discussion

The simulator functions as expected. We have managed to implement a RISC-V simulator with both the base integer subset and the multiplication and division extension. Additionally the visual debugger, although not without some visual issues, is entirely functional. In the following section I would like to discuss some of the issues faced during the project, in particular with C as this has been a definite weakness.

RISC-V Execution: Through thorough testing I can conclude that there currently are no known issues with the simulation portion of the program, but the road to achieving a functional simulator has been far longer and tedious than I expected.
When I originally chose C as the software language I expected it to fare excellently with bitwise operations, but several days were spent fixing unexpected sign extension. C does not have separate operations for logical and arithmetic shifts as they are done on a sign basis. This became a major issue as decoding instruction fields relies on a logical left shift followed by a logical right shift. If the outermost bit of the field desired contains a 1, the entire field is sign extended. Many of these issues were solved with a simple unsigned cast, but the issue was by far the biggest headache during development.

Loading program data: The first feature implemented was loading files, as manually writing test cases in an array was too tedious. This was done before I implemented memory, and once memory was allocated I had completely forgotten to move the program data to memory. This has been solved temporarily by loading the program into its own memory, and then storing it into processor memory. The entire program still runs off of the original memory, but data can now be accessed and viewed in the debugger. This should have no impact on the simulator, unless someone were to write over the compiled program instructions, and I have left it as is. Moving the simulator to run entirely off processor memory should not be difficult switch, it would require testing everything once again which there is no more time for.

Visual Debugger: From the original concept of the debugger, I am very pleased with the results. However there is one bug that bothers me, and a feature I wish I could have implemented.
The bug is the obvious odd characters on the first elements in the translated machine code, and I have fruitlessly tried to solve the issue. More information on it can be viewed on the github.
Although the design of the debugger is almost perfect, it lacks some way to indicate where the user is positioned in the memory, or what window is currently active. The original idea was to implement a highlight, either in bold or a color change, to indicate important positions to the user. There were some issues in terms of colors with ncurses, and I was unable to achieve this. The feature was low priority, and I decided to simply memorize the current position, as is described in the github.

Redundancy and bad style: This is more of a personal issue than one with the program itself. Due to working alone during this project, and being un-

familiar with proper practices in C, a lot of the coding style changes wildly throughout the development process. If I had chosen to work in a group or chosen a more familiar language the size of the program and redundant code could have been reduced significantly. Nonetheless it has been a learning experience, and despite some frustrating moments with sign extensions the overall experience has been fun.

6 Conclusion

In this project we built a RISC-V32IM simulator in the C language, with command-line functionality and a visuald ebugger built with the ncurses library. The simulator functions as expected, although the visual debugger is not completely intuitive and contains a major visual bug.

The simulator can be found at this github repository: `https://github.com/publicBugi/RISC-V-Simulator`

My concluding thoughts on the project is that it has been a fun, and frustrating, delve into C programming and the overall project has been well worth the effort.