Project Report

RISC-emV

A RISC-V graphical emulator by
Alessandro Sartori and Davide Zanella

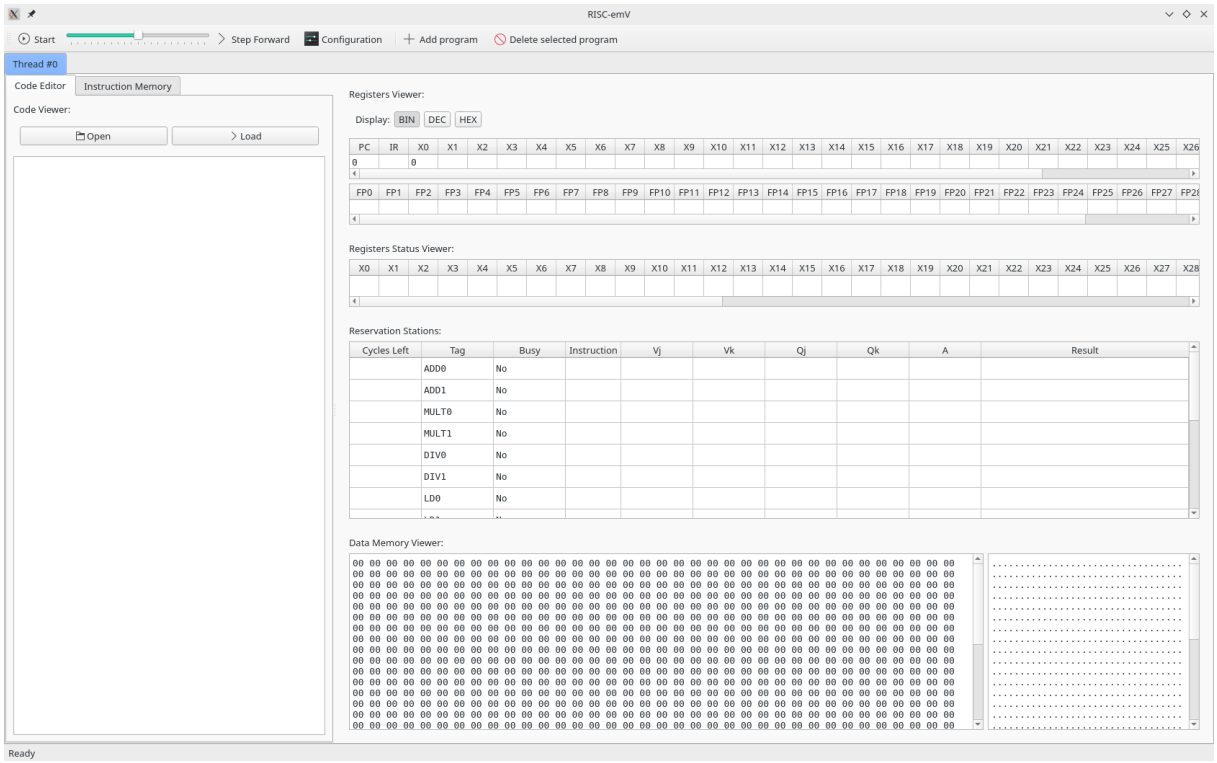
`https://github.com/AlexSartori/RISC-emV`

1 Overview

This software aims at emulating a RISC-V CPU implementing the *Tomasulo Algorithm* with support for *simultaneous multithreading*, and is mainly conceived for academic or research purposes. The name comes from the conjunction of the term "RISC-V" with the abbreviation "emu" for emulator.

Multithreading operates with the "double-issue" technique, while branch prediction, as of now, is absent and the emulator simply stalls until the next instruction is known.

2 User Interface and Usage



From top to bottom, the home screen contains a toolbar with the main emulator controls, a tab-bar to navigate and interact with the different threads (only one is loaded in the above screenshot), and an interface to the active thread's components. A per-component description follows below:

- **Toolbar**:
  - **Start Emulator**: run the emulator executing one instruction after the other automatically

  - **Step Delay**: delay to interleave between one instruction and the other
  - **Step Forward**: perform one step and halt
  - **Configuration**: open a configuration window to tune many parameters, such as number of functional units or per-instruction computation time in cycles. In the future it may hold more interesting possibilities such as the type of branch predictor to employ
  - **Add thread**: create a new thread tab to load a program
  - **Delete active thread**: delete the thread associated with the currently active thread tab

- **Thread tab bar**: list of the currently active threads (with their associated colors for clarity). Clicking on one will update the frame below to match the thread's execution environment

- **Emulator Components**:

  - **Code text box**: read and edit the assembly code to be loaded and executed. The **open** button allows to import external **.s** and **.o** files, while the **load** button parses (or disassembles) the code and loads it into memory. This last step is necessary every time a new document is opened or a modification to the code is done, in order to update the process image in memory
  - **Instruction Memory**: table of all loaded instructions with statistics on their timings when run (issue time, execution cycles, write-back time). In the future this section will allow the extraction of more detailed and informative reports about instruction frequency and functional units usage.
  - **Registers**: interact with integer and floating point registers with the selected display format (binary, decimal, hexadecimal)
  - **Registers Status**: inspect the status of registers, e.g. for which reservation station a register is waiting
  - **Reservation Stations**: for each functional unit inspect its current instruction, data dependencies, and execution progress
  - **Data Memory**: inspect the contents of the program's memory, both in hexadecimal format and in ASCII characters

3 Source Code and Project Contents

This project has been developed using **Python 3.x** and **PyQt5** for the GUI. The repository is structured in three main directories, containing the actual package (riscemv/),

a bunch of sample programs to play with (sample_programs/), and a collection of tests (tests/) used during the development of the software to ensure the absence of regressions.

RISC-emV/:

- riscemv/:
  - Main implementation of the emulator architecture. **DataMemory.py**, **InstructionBuffer.py**, **RegisterFile.py**, **ReservationStations.py**, and **RegisterStatus.py** provide classes equivalent to their respective hardware elements, which are then collected and instantiated by the emulator object implemented in **Tomasulo.py**. Lastly, **Program.py** represents a process image, possibly generated by **ELF.py** if loaded from such format.
  - gui/:
    - Files that generate the user interface. Many of these are the graphical counterparts of the previous mentioned classes, while **MainWindow.py** and **ConfWindow.py** serve as their respective windows launchers.
  - ISA/:
    - Implementation of the 7 types of RISC-V instructions (R, I, S, B, U, UJ, R4). Each of these types is available as a dedicated class, but all of them inherit common characteristics from a parent class contained in **Instruction.py**.
    - Extensions/:
      - JSON files describing each ISA extension. Files in this folder are found and parsed automatically, and are composed of a list of instructions categorized by their type. Each instruction is represented as a JSON object containing its essential attributes, such as opcode, equivalent pseudocode, associated functional unit, etc.
- sample_programs/:
  - Ready to use assembly and object files, mainly carrying out very basic operations aimed at showcasing emulator's functionalities without introducing real world programs' structural complexities.
- tests/:
  - Pytest-compatible tests, achieving (at the time of writing) 70% code coverage. Continuous Integration was ensured thanks to TravisCI.

4 Challenges and Difficulties During Development

A central challenge in projects with such a (somewhat) elevated number of classes and components is certainly proper synchronization and collaboration among teammates. Thanks to an appropriate use of version control systems and efficient communications, however, we managed to split work and responsibilities with great accuracy, so that every problem that arose was quickly traceable back to a source and an author that could efficiently take care of it.

Speaking of specific implementation challenges instead, support for ELF files proved to be quite a difficult feature to provide: although this format is very regular and easy to read thanks to the fixed-size tabular format, it has several fields with hard-to-comprehend meaning/purpose if one doesn't have a thorough knowledge of the lifecycle of a source/program/process, as is our case.

Multithreading instead, we admit against our expectations, was only a matter of providing a means to instantiate a second emulator. This simplicity, as a matter of fact, was only allowed by the modularity we aimed to achieve from the beginning.

5 Possible Future Advancements

Thorough debugging might be considered our number one priority. This raises the need to extend current code testing to cover many more details, possibly including strict checks on data types, since in Python these are not enforced enough and often cause mismatches and unexpected type casts, leading for example to imprecise instruction outputs.

The strong modularity of this project also allows us to include in this list the development of further ISA extensions. For instance, RV32A to handle atomicity constraints to better handle multithreading, RV32C to support compressed instruction formats (RISC-V compilers currently default to this, unless otherwise requested), or more in general, implement 64 bit (or 128 bit, if confirmed in the near future) instruction parsing, which already meets partial support in the **ELF.py** class.

ELF file parsing could also see some improvements, such as proper parsing of linked programs, as currently it is only guaranteed to correctly parse relocatable programs.

Emulation of system calls and system libraries, in conclusion, would be another interesting feature, allowing for example a program to invoke I/O functions such as **printf** or **scanf** to interact with the user.