C Project - Checkpoint Report

ARM Emulator

Group 9

Carlos Valencia Vargas, Mihnea Gogu,
Madi Baiguzhayev and Alex Constantin-Gómez

June 18, 2020

1 Group Organisation

The first task we set ourselves was to individually read through the specification and try to understand the task as a whole. After that, we held a meeting via voice call to clarify any doubts as a group and start planning how we would split the work equally across all four members. Before splitting the work, we collaboratively identified key variables, structures and functions that were likely going to be required. We decided that Mihnea and Carlos would work on together on the pipeline decoding functionality and both the multiply and branch instructions (since they were simpler to implement), Madi would work on the data processing instruction and Alex would work on the single data transfer instruction. Once we all finished, we combined our code together and all contributed to debugging and improving the code.

As a group we all think we have been collaborating well as a team due to several reasons. From a logistical point of view, we have a text chat where we discuss different design approaches, clarify doubts related to the specification, schedule voice calls (often daily) and provide regular updates to the other members of the group.

With regard to collaborative programming, we initially defined a set of stylistic conventions to ensure our code is consistent and readable to the rest of the team and the marker. To enforce these conventions, we often review each other's code and make corrections where necessary.
We have also made extensive use of Git branches so that each member can work on their assigned task individually. Before merging branches into the master branch, we all check that our code compiles and that all other members are happy with this decision. The only times where we directly committed our work to master was at the beginning, to provide some starter code and the file structure and at the end, to fix any bugs (such as memory leaks) and refactor code.
We also try to write descriptive but concise commit messages, unless it was a very small change to the existing code.
Throughout the development of the emulator, we had no merge conflicts. Since for three of us it was our first time using Git for collaborative programming, we all think we've used it fairly well and learnt how and when to create branches and merge them. In general, we are all quite happy with how the team has worked so far.

2 Implementation Strategies

2.1 Emulator structure

Our program structure consists of an `emulate.c` source file which contains the `main()` method. It then calls the required functions from the `emulator/` directory, which contains a source and header file for all four types of instructions, `utilities.{c,h}` with common utility functions, `barrel_shifter.{c,h}` with shifting operations shared by two instruction types, and a source and header file for the CPU state and pipeline functionality.

We are likely to reuse the `Instruction` struct and the `instruction_type` enum. We also think that utility functions such as `process_mask()` and `print_bits()` will probably be reused since they are either very commonly called throughout the emulator code or they are just useful functions for debugging and testing.

2.2 Mitigating future challenging tasks

Testing: Although the given test suite had several test cases for each type of instruction, without integrating all the different instructions together it was difficult to test the emulator. To mitigate this issue, we will try to write unit tests for each instruction type in the assembler, since we can more easily check if an instruction string maps to its correct binary representation. In the emulator, this was more tedious because we had to check the entire state of the processor (the registers and the memory). *Comment by Carlos, Mihnea, Madi and Alex.*

File structure: Our initial file structure seemed like a good start however we realised towards the end writing an efficient Makefile was challenging because of the dependencies of each source file. To prevent this for the following parts, we will try to design a file structure such that there are no complex dependencies between files by drawing an dependency graph. This will make it easier for us to write a good Makefile and for future projects, it will hopefully motivate us to plan and design an effective and structure codebase from the start. *Comment by Alex and Mihnea.*