ARM Project Checkpoint Report

Dina Duong, Maciej Rytlewski, Kritik Pant, Matty Williams

June 8, 2023

1 Group Organisation

Task Allocation

- Kritik Pant: Representing machine, memory initialisation, instruction processing, and data processing immediate and registers instructions. Then moved on to Part 2 (Assembler).
- Maciej Rytlewski: Single data transfer instructions, bug fixes, code reduction.
- Dina Duong: Branch instructions, bug fixes, code reduction and report.
- Matty Williams: Code optimization through function pointers and moved on to Part 2 (Assembler).

To coordinate our work, we made a separate branch in the Git Repository, where we experimented with the code. We tried various approaches to representing memory and the state of the machine before eventually settling on an array for memory and a struct for the state of the machine containing register values, PSTATE, and the memory array. After fixing some initial errors and testing our representation, we pushed the code to the master branch. After that, we divided the task of implementing instructions equally between the group members (stated above) and built some useful utilities to reduce code duplication. When the emulator could pass some basic test cases, Dina and Maciej focused on bug fixing while Kritik and Matty moved on to Part 2.

Group Evaluation

Overall, our group has been working well together with everyone doing a fair and equal amount of work. We found Part 1 to be quite sequential so there were limited opportunities for parallel work. However, the division of tasks among group members helped us progress efficiently. Regular communication was maintained through discussions, ensuring everyone was aware of the progress and tasks assigned to each member.
As we progress to Part 2, we expect to have more opportunities for parallel work. However, this will require better coordination and communication between group members. We will need to ensure that everyone is aware of the progress and tasks assigned to each member and that we are not duplicating work. We aim to hold regular progress updates and meetings to ensure everyone is on the same page and to address any challenges promptly. We will also do code reviews to ensure the code is clean and well-structured.

2 Implementation Strategies

Emulator Structure

We used a modular approach by splitting the code into separate files. Our file structure included:

- Setup File: This file handled memory initialisation, processing instructions, and other setup-related operations. We implemented memory as an array and used structs for modelling PSTATE and state of the memory and registers.
- Instructions File: Each instruction was implemented as a separate function within this file, allowing for easy modification and addition of new instructions.
- IO File: This file handled reading and writing to files.
- Utilities File: We created a utilities file containing commonly used functions, such as 32-bit and 64-bit mode handling and sign extension. These utility functions were reusable across various instructions and improved code organisation.
- Main File: The entry point for running the emulator.

Reusability for Assembler

In the Assembler, we expect to reuse some of the utilities and IO which we used in the Emulator.

- read_from_file: We should be able to use the function almost exactly to read the assembly file before tokenizing it.
- write_to_file: Similarly, we should be able to use a variation of write_to_file to write the binary output file of the assembler.
- set_bits: The set bits function generates a mask and is used to set bits in registers in the emulator. We should be able to adapt this slightly to set bits in binary lines in the assembler output file.

Future Challenges

- Splitting Up Work: As mentioned earlier, the project is somewhat sequential, making it hard to parallelise work since certain tasks depend on others to be completed. To address this, we have maintained good communication and made good use of git branches and commit messages to help us keep track of each other's progress. We aim to continue this approach as we progress to Part 2.
- Reducing Code Duplication: As code length increases, we have seen how it can be challenging to manage and maintain it. To mitigate this, we have adopted an iterative approach of working; building a function and then refactoring it to reduce code duplication. We have also held code reviews to identify areas of improvements which has helped us clean up the code.
- Identifying Bugs: We are conscious that the assembler involves many different parts such as the file reader, tokenizer, parser, and binary generator. As such we imagine that bug fixing will become more challenging. We aim to address this by holding code reviews to identify each other's mistakes as well as use debuggers such as GDB to help identify bugs and Valgrind to check for memory leaks.