C Project Interim Checkpoint Report

Ifechukwude Ndozi, Amogh Shetty, Rohan Desai, Anshul Dani

June 6, 2025

1 Group Organisation

1.1 Splitting the work between the group

After reading and discussing the specification of the project together we decided to split the work based on the areas that each individual felt the most confident with. This led to Amogh being in charge of the fetch and decode sections of the execution pipeline, Ifechukwude being in charge of the state of the ARMv8 Processor and its initialisation along with the execution of the Data Processing Immediate instruction, Rohan being in charge of the execution of the Single Data Transfer instructions and the Branch instructions and Anshul being in charge of the Data Processing Register instruction and handling the outputting of the state at the end. This ensured the best utilisation of each group members' strengths and maintaining a balanced workload.

1.2 Coordinating our work

We coordinated our work using the pair-programming technique, meaning we split into 2 pairs of Amogh with Anshul and Ifechukwude with Rohan. As a result each pair was working on roughly half of the workload of part 1 due to what each individual was in charge of. We implemented the strategy by having one person type, while as a pair, we would discuss the logic needed for each part of the problem we were trying to solve, which made it easier to spot potential bugs and edge cases we would need to consider. We were also quite new to using collaborative version control, so we had to learn new concepts such as branching and merging to work effectively on different parts of the specification at the same time and then bring it all together on the master branch. We did this by using one git branch for each feature we made and frequently committing with clear comments to ensure that the group stays updated on what has been done and what still needs to be done.

1.3 Reflection on how well our group is working

Our group has worked well with pair programming, we found it has led to:

- improved code quality and style
- fewer errors through real-time debugging
- more efficient and reusable solutions
- better understanding of the project as a whole through clear communication

However, it has not always proved to be the best strategy. For work that is more repetitive and less complex, pair programming has not benefited us and we would be faster working individually. For the rest of our project, we aim to use pair programming more selectively in order to maximize efficiency.

As a group, we believe our code style has been good. We have made sure to use appropriately-named variables and functions throughout our work, as well as clear comments to improve readability and maintainability. By peer-reviewing merge requests on GitLab and working in pairs, code is always being reviewed by another person, which helps to reinforce good programming practices. We have also avoided using magic numbers by defining constants in `.h` files.

2 Implementation Strategies

2.1 Structure of `emulate.c`

We divided the structure of the emulator into the three stages of the pipeline: fetch, decode, and execute. We then subdivided both the decode and execute stages into the four possible instruction types we identified: data processing immediate, data processing register, load/store and branch.

As a group, we took the decision to handle the `halt` instruction during the fetch stage by setting a `haltflag` in the processor state when a halt instruction was fetched. This approach allowed the decode and execute stages to easily check for the halt instruction and skip processing if detected, ensuring the program is correctly terminated.

We split the file structure to match our problem structure and how we delegated the functions, and our Makefile includes all the dependencies we need so we can automatically compile our project.

- `emulate.c`: Contains the main function, which calls the binary file loader, carries out the pipeline process, and prints out the state of the processor after all instructions have been executed, either to `stdout` or a specified output file. It also contains both the fetch and increment-PC stages of the pipeline.
- `instr-decode.c`: Contains the decode stage of the pipeline, with functions for the four possible types of instructions.
- `instr-execute.c`: Contains the execute stage of the pipeline, with functions for the four possible types of instructions.

2.2 Data Structures

- **ARMv8 Processor State** (ARM-STATE) : We have chosen to use this struct to represent the current state of the ARMv8 processor. It consists of an array of 64bit unsigned integers to represent the general purpose registers (that can be used in either 64bit or 32bit mode) , an array of 8bit unsigned integers to represent the byte-addressable memory, the special program counter and processor state registers, the output file for the state to be printed to, and a boolean flag indicating whether the program has been terminated.
- **Decoded Instruction Type** (DECODED-INSTR) : We also chose to represent the decoded instruction as a struct, and store it in the current state. This struct contains all the different fields for each type of instruction as described in the specification, such as operand, opcode, immediate values (`imm12`, `imm16`), and offset. The decode stage splits up the instruction and stores the relevant fields, enabling the execute stage to directly use these fields. The design of this struct also greatly helped the team members working on the execute functions, as they could directly draw on the necessary fields.

2.3 Re-usability of code for `assemble.c`

Our assembler can make use of many of the same data structures and enumerated types we have used in the emulator, so we can reuse these. As an example, various parts of the decoded instruction need to be stored somewhere before we convert the full instruction to binary, so we can reuse the `DECODED-INSTR` struct we created for the emulator.

Furthermore, the logic we used to identify instruction types and bit fields can be inverted and reused, as the assembler will be encoding into binary rather than decoding it. This means we can reuse many of the enumerated types we created in order to identify instruction types and subtypes, such as branch-type, instruction-type, etc. Our binary file reader can also be reused and converted in order to design the binary file writer we need for the assembler.

2.4 Future Implementation Challenges

One future implementation challenge we will face is maintaining the modularity of the code. While we have attempted to make our code modular while writing `emulate.c`, members of the team working on different stages inevitably led to multiple functions with similar functionality and parts of the project spread in different files. Going forward, we plan to make our code more modular by planning the file structure of future parts of our project to better reflect the different problem stages, and grouping generic helper functions together in a separate file, so that relevant functions can be reused. This will help us to prevent inconsistent and duplicate code, make it easier to merge our git branches and make testing our code much simpler.