ARM Checkpoint Group 22

Ruben Vorster, Arron Stothart, Pip Moss, Faris Chaudhry

1 Group Organisation

Upon starting the project, we were all committing to the master branch at once; this allowed us to implement the structure for the skeleton of the emulator quickly (i.e., loading binary files, printing the processor state after halting, and the main fetch-decode-execution cycle) however this did lead to some code conflicts since work was being done asynchronously on a synchronised branch. However, after completing this initial section we created branches for each of the main four features (data transfer, immediate and register processing, and branching) where each one was primarily delegated to a different member as to modularize our workflow.
In order to ensure correctness and uniformity, pull requests were reviewed by a different member of the team and bugs were fixed throughout by pair-programming and code reviews. While we have created an issue board on GitLab with labels for different categories of issues for various stages of the project, we have not used it to the fullest extent, which we plan to do going forward. Furthermore, we did not test the emulator throughout development as much as we would have liked, since emulation requires many intertwined features to be implemented, despite testing various functions modularly, particularly utility functions. When developing the assembler, we would like to add our own smaller, more specific tests to the test-suite. Lastly, we could add documentation comments for functions which detail the inputs, result, and any preconditions more explicitly. If we were working a bigger project, we might consider creating a template for comments, documentation, and pull request/issue formatting to maintain a consistent codebase.

2 Implementation Strategies

The main functions is contained in emulate.c. We start by taking in the binary file and initialising the ARM; the ARM processor is a struct containing an int64 array of size 31 for registers, an int64 for the program counter, the PSTATE - which is itself a struct containing booleans for the program state flags, - and an char array for memory. This puts 1 byte for each entry in memory making the memory byte addressable; and since each character takes 1 byte, the size of this array is precisely the size of the memory, i.e. 2^{21}. The binary file is loaded into the ARM's memory 1 byte at a time until the end of the binary file is stopped, after which all memory addresses are zero. We then start an infinite loop of fetching, decoding and executing instructions unless the halt code is executed (therefore we assume that each binary file is in correct assembly code otherwise the program will fail). Fetching is done by retrieving the contents at the memory address of the program counter and forming it into a word; decoding is done by a switch statement in a utility function which discriminate between instruction types by bit masking and returning the corresponding enum for the type; execution is done by a switch statement which defers to a modular handler function for

the given type, with the exception of special functions NOP and HALT. In the case of halting, we jump out of both the for loop and switch statement, outputting the state of the processor, done identically to the spec, and exiting with a success. When outputting memory we must convert it into big endian since words are loaded in little endian.
For each instruction type, of which there are the four aforementioned, the implementation is approximately similar (though both types of data processing are subsumed under one). After being passed the instruction, we first ascertain what subtype it is. For branching, this is unconditional, conditional, and register; for immediate data processing, arithmetic and wide move; for register data processing, arithmetic, logical, and multiply; and for data transfer the addressing modes: pre and post indexed, register offset, literal, unsigned immediate access. A helper function and sometimes a respective enum is defined for this purpose. Then we use switch statements for each case and deal with the instruction by breaking it into each part (e.g., rd, rm, operand, simm19, etc.,) and carry out the appropriate operations. Noteworthy examples include immediate data processing, were function pointers can be used in place of a switch statement, since the opcode is of consecutive numbers (i.e., 0 for movz, 1 for movn, 2 for movk). Another example is in branch, where a helper function is used to check the condition before executing a conditional branch. In addition, we increment the program counter by 4 each time before executing the instruction, thus branch must account for this when computing its offsets.
Finally, we have created defs.h and utils.c. The former contains constants and enums used to eliminate magic numbers while the later contains utility functions for general purposes such as bitwise manipulation and converting integers between little and big endian. We intend to reuse these in the assembler if not fully then partially - especially functions such as getBitAt, getBitsAt, setBitsAt which generalise bit masking and perhaps bitwise rotations and shifts. However, we expect that we might need to rearrange the file structure to do so, as we currently have two disjoint subdirectories for the emulator and assembler. Hence, we might create a separate library file containing the shared helper functions that would be imported into both. Furthermore, we expect to use our emulator for testing the assembler.