ARM11 Project Checkpoint Report

Devesh Erda, Raihaan Rahman, Prabhjot Grewal, Eugene Lin

May 30, 2019

1 Group Organisation

For this project, our group decided that it would be beneficial if we each focused on our own sections of code, and help each other whenever possible. Any major decisions that could affect how the rest of the program would be implemented (such as the choice to use structs for our emulator), would be decided together, such that all of our individual components can work together. In order to minimise any merge conflicts, we decided to work on separate files (and also only worked on our own branches), which also improved our parallel coding.

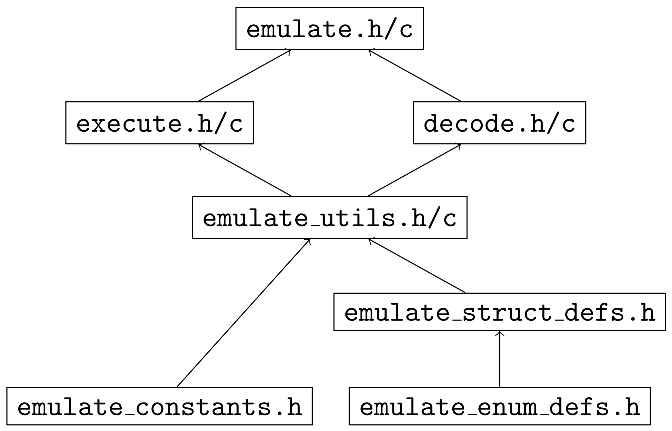
In order to maintain style, we decided on the style beforehand, and allowed the IDE to automatically reformat the code to match. Not only does this ensure consistent code, it prevents issues such as lines being overwritten (possibly leading to merge conflicts). As everything is done through GitLab, we never work directly on the `master` branch, and we prune (and delete) any branches that are no longer used (after it is merged). The LaTeX styling has each sentence on its own line, as it allows for easier reading, and tracking on version control.

Our group works well together, as we maintain frequent communication - which allows us to bounce ideas off of each other to get a different perspective on a problem. We meet quite frequently to code together, however a possible improvement we could make is a more organised system of distributing tasks.

2 Implementation Strategies

2.1 ARM Emulator (Part I)

In order to structure the code, we stored the machine state, as well as decoded instructions as structs (found in `emulate_struct_defs.h`). To improve readability, and allow for easier maintenance, we stored values that we reuse in `emulate_constants.h`, and `emulate_struct_defs.h`. The utilities file, `emulate_utils.h/c` holds general procedures such as initialising the machine (setting the states, clearing memory), as well as reading from a file, and some other execution utilities such as flag checks, memory access, and shifting. It also allows us to display the register contents, as well as any other memory used.



Our implementation for the decoding stage of the cycle is fairly simple, and is found in `decode.h/c`. The approach we took was to first check the type of instruction we were working on, which was done with bitwise AND on a mask, and checking with the expected patterns found in `emulate_constants.h`. The remaining work was to just mask the bits, to get the part of the instruction we want (based on the type), and then apply logical shifts to remove unused lower bits.

The first step in execution (found in `execute.h/c`) is to check the type of instruction, and then check each of the cases, which would've been set by the instruction decoder. The execution of instructions also sets the flags in the CPSR (Current Program Status Register), based on any unexpected behaviour.

Our pipeline implementation uses 3 flag bits, with `0x1` as decode, `0x2` as execution, and `0x4` as termination. While the termination bit isn't set, we move the decoded instruction to be the execution instruction, and if the execution bit is set, it is processed further. If the execution instruction is a branch, and the condition is checked, we set the flag to `0x0` (which represents the start state, or a clear pipeline), and then execute the instruction. However, if it's a termination instruction, we break out of the loop straight away, and terminate the program. Next, if the decode flag is set, the fetched instruction is decoded, and the execution flag is then set. Finally, the next instruction is read from memory, from the location specified in the program counter, and the program counter is incremented by 4 (as our memory is addressed in bytes), and the decode flag is set.

2.2 ARM Assembler (Part II)

Very few of the components we wrote for the emulator could be reused for the assembler (at least not without making significant modifications). For example, the memory handling (allocation, and freeing) is completely different, since the structs we've used are different. The reading process is completely different for the emulator, compared to the one we've used for the assembler, and the former doesn't need to write to disk.

However, we were able to reuse the shifter, since we noticed a possible issue while writing the assembly for Part III. In order to deal with the immediate data operand, in the data processing instructions, we'd have shift the operand to fit within 8 bits (if it was sufficiently large), and calculate the 4 bits for rotation.

As the assembler reads the file, it builds the symbol table, and adds to an array which contains the source code. Any blanks lines are skipped. It iterates over all the instructions, takes the opcode, and calls the corresponding functions.

2.3 Current & Future Challenges

During the first week, we found that some nuances in the language, and the lack of some features we take for granted was quite difficult. For example, dealing with memory usage isn't as big of a concern in some other programming languages, but having to manually allocate memory was something we had to consider here (since we use a lot of large structs). Dealing with integers of different sizes (for example, `uint16_t` versus `uint8_t`) was also an issue we found - if we were to use a smaller integer as a mask for a larger one, it would zero out the remaining bits, leading to unwanted behaviour. We also encountered some issues with the `ldr` instruction, due to the number of different cases we'd have to consider.

We anticipate that we may encounter possible issues with the extension task, especially since there is less guidance. At the moment, we haven't decided on an extensions that is sufficiently complex, and also unique - so we cannot comment on specifics about it.