Group 27 Emulator Summary

William Springsteen Rory Fayed Stefan Klas Kiran Patel

October 29, 2015

At the start of the project, we decided that we would wait until we learned a bit more of the C language, and had a bit more practice in it, to start writing any code. This meant that we were going to wait until after the weekend to meet in labs and really crack on with writing the emulate program. Over the weekend, however, our group communicated and it was decided that Stefan and Billy would have a go at writing the outline of the emulate.c file. In the meantime, Kiran and Rory would carefully go through the specification, learn some more C, and get some more practice in C. They would also look at the code that Stefan and Billy would regularly push to git, as more code was written, and give feedback as we went along. All communication over the weekend occurred via online messages.

After the weekend, we decided to meet in the University labs at 10am, and had a talk about everything we had found out/worked on over the weekend. We had to make sure that everybody was on the same level of understanding on the code and how it worked, and we had to all agree on the way we were going to go about writing the program. For example, with respect to reading the instructions from the binary file, the instructions are little endian in the file, and we had to decide whether to store this in the memory in little endian form or big endian form. Another thing we had to agree on was what form we would put each instruction in so that we could manipulate it in various methods we could either use a normal number in binary form, use an array of 1 or 0 chars, or use an array of 1 or 0 ints. At the time, we didnt realise that any int, or `uint32_t` type could be converted to binary using 0b, so we thought we had to use the command `uint32_t` x = 1011 to make x have the binary value 1011, or 11 in base 10, and we obviously knew this would break when there were more than 10 bits, or some value around that number, so there was no way we could store a 32-bit value like this. However, we now know that `uint32_t` x = 11 is the same as `uint32_t` x = 0b1011. After making sure we were all doing things the same way, we shared/brainstormed ideas for the rest of the program, and decided who wanted to do what parts of the program, after deciding to use mostly Stefans current layout (with enums and structs too), and mostly Billys current implementations. We decided that for the next few days, Billy would work on decode, other general small bits, and make sure everybody was getting on ok with everything, Stefan would work on decode and execute, Kiran would work on file loading and main, and Rory would work on execute and other bits that people needed help on.

After we all finished our parts, all that was really left to do was debugging and testing. We decided it would be best to also split up our 1000 line file into lots of smaller .c files and header files, which would both make it easier to debug and make the code easier to read, more presentable, and more professional. Kiran and Rory were assigned to do the file splitting, while Stefan and Billy were assigned to do the testing and debugging. The initial debugging involved mainly syntactical errors, such as missing semicolons, as well as spelling mistakes on some variables/method names, so that we could actually compile the emulator. Once Kiran and Rory had finished the file splitting, they started solely working on testing and debugging too, as there was a lot of code to debug, and bugs can be hard to find and fix.

Overall, the group is working really well together. There has been lots of communication between all members, both when not together (via online group messages), and when meeting in labs. We were all willing to meet in labs every day after the weekend from the morning until the evening, and sometimes some

members would stay even later than that. We always made sure that everybody was kept up to speed on what we had all coded, and any new ideas any of us had. The only thing that might need to change is when we decide to really start the task. This is because, for emulate, we decided not to do too much until after the weekend, which we found meant that we were a little bit pushed for time to finish by Friday, as emulate took much longer than we had hoped. We would then have had more time to think about how to do part 2 (assemble) while making emulate, which might have changed the way we made the emulator, because we might have done something in a different way so that we could reuse some code/ideas for assemble.

For our emulator, we have some global constants made using `#define`, for things like the number of registers and the size of the memory, which stop magic numbers being used. We have lots of enumerated types, for the different conditions, the positions of the important CPSR flag bits, the different opcodes, the different shift codes, and the different instruction types (Data processing, branch, halt, cleared, etc.). These enumerated types help make it easier to tell what the different numbers mean when we are processing them. For example if we process the opcode as being 0xD for a particular data processing command, we know it is a MOV command, thanks to the enumerated type, and we can check this by saying /textttif (0xD == MOV). We have a big struct for the virtual ARM machine, which has an array of of 8-bit ints for the memory, as the memory is byte addressable, and an array of 32-bit ints for the values in the registers. We have a struct for the final fetched instruction and the final decoded instruction, each of which will only have one instance created, which will be the two parts of the 'pipeline'. The fetched struct has a `uint32_t` for the actual fetched instruction itself, and a bool called isFetchedCleared to be able to tell if the fetched stage in the pipeline is cleared. The decoded struct has an instruction type enum, so the program can tell what type of instruction it is, as well as if the decoded part of the pipeline is cleared, and a union of the decoded instruction's information. This union of the decoded instruction's information can be one of four structs (one for each type of instruction that isn't halt or cleared), each of which holds the information for each different instruction type, such as the condition enum, the opcode enum, the offset, the different register numbers, etc. To tell which one of these structs is set in the union at any given time, you can check the instruction type in the struct which contains the decoded union and the decoded instruction type.

The most important functions we have are fetch, decode and execute, as well as main, obviously. Fetch takes into account the litte endianness of the memory, converts the next instruction into big endian format, and stores it in the fetched part of the 'pipeline'. Decode changes the decoded part of the 'pipeline', and will decode any instruction it is given, although the program has been made so that decode doesn't handle the case where the decode part of the 'pipeline' is cleared, and instead decode simply won't be called if this is the case. Execute will execute the function that is in the decoded part of the 'pipeline'. Lots of helper functions are used throughout these big functions. The main function will load every instruction into memory in little endian form, and will then proceed to execute, decode, fetch and increment the PC register, in that order, in a loop, until a halt command is reached (all 0 instruction). Upon termination, the program will print out its registers and non-zero memory. We decided to split up all these methods into different files, so that it was easier to find specific methods, and it looked a lot neater. All methods were grouped together and put in a file with a logical name, so that all related methods are together as much as possible.

We aren't really sure how much we can reuse for the assembler, but the fact that we split our emulator into lots of different methods/helper methods means that we should be able to reuse a lot of these helper methods, such as the methods in the binMethods.c file, which are methods dealing with the manipulation of binary numbers.

We think we may have problems getting the next parts finished on time, especially as we think the extension will take quite a long time, so we are going to try to start everything early and not waste any time. We also found that there are a few things we were unsure on, and found it difficult to understand them just by looking on the Internet, as sometimes things can be ambiguous. Due to this, we will attempt to ask lab helpers or members of staff or other students for help understanding things if we are really stuck, which will hopefully stop us doing something wrong at the start of a program, and not realising until the end, which would cause the changing of a lot of code.