ARM Final Report Group 22

Ruben Vorster, Arron Stothart, Pip Moss, Faris Chaudhry

1 Structure and Implementation of Our Assembler

The main function in assemble.c is used to run our assembler. It reads in source code from an AArch64 source file, whose filename is given as the first command line argument, and outputs AArch64 binary code to a file, whose file name is given as the second command line argument. We achieve this through a two-pass assembly process.

1.1 First Pass

The first pass generates a symbol table associating each label of the source file with its correct address. We used a global linked list to represent our symbol tables and equpped it with the neccessary functions: getAddress, which returns the address correspoinding to a label, and addSymbol, which adds labels if they are not already present. The contents of the source file are read to a buffer from which each line is parsed. If a line match the label regex formatting, it is written to the symbol table alongside the 4-byte aligned address, obtained by multiplying the line number by instruction size. Notably, blank lines do not increment the address and the address of a label is the address of the next instruction, i.e., on the next line.

1.2 Second Pass

The second pass reads each instruction/directive of the source file and generates the corresponding binary encodings. Our program moves linearly through each line of the buffer. When a line representing an instruction is reached, it is tokenized to produce an opcode and an array of up to 4 operands. This process uses calls to strtok_r to break the line into the mneominc and its tokens, and trimWhitespace to remove leading and trailing whitespaces for consistency. The opcode of the instruction then is hashed, and the hash value used to obtain the relevant function pointer. Here using a hash function allows us to avoid multiple if statements by hashing based on the characters of the mnemonic and finds the correct function to deal with it in constant time; although for mnemonics of length one or two (b and br) must be dealt with seperatley to avoid accessing unallocated memory. Funtions to assemble instructions are called with the obtained operands and the line's address (needed to calculate the offset) as arguments and are put into an array of uint32_t sized inststructions to be read into the outpute file. For every such function we used bitshifts to put the bits into the correct position and util functions such as getRegNum, getImmediate, and calculateOffset to parse operands into their correct form. Moreover, we added a collection of constants in defs.h aswell as local constants for each instruction type module; to avoid redefinition errors, ifndef was used.

Branch instructions are split into three categories: uncondtional, conditonal and register. We

created a base instruction for each type which contained the invariant bits. For register branches, the registers number needs to be shifted, and for unconditonal branches we calculate the offset using the label address and the line address. All offsets are stored as unsigned ints and are truncated to the correct length so to remove extra leading 1s. However, condiontal branches use a single static function to create the instruction and shift the binary value for their conditions after.

Nop was the only special instruction we explictily dealth with. This is easily done by returing the nop code that is outlined in the spec. We didn't need to do the same for halt because it is considered an alias for 'and x0 x0 x0' and therefore requried no special logic. Likewise, the only directive required to implement was .int. When the int directive function is called, it differentiates between hex and decimal values by checking if the operand is prefixed by '0x' and uses strtol to convert the strings into the correct data. Strtol is used instead of atoi to specify the base.

Data processing instructions are split into the categories of wide move, multiply, logical, and arithemtic. Of these, multiply is the simpliest with all variations (msub, mul, mneg) reducing to forms of madd. Both wide move and logical instructions have a centeral function that deals with the logic. Finally, arithmetic instructions may be split into a further two categories of register processing arithemtic and immediate processing arithemtic. We can differentiate between these by checking whether the 3rd argument is a register, which indicates the former, or an immediate value, for the later. Similarly to branch, all individual instrucion type functions call their umberalla function to which they add their opcode. The existence of optional shifts is validated by checking if the optional argument is not blank using strcmp, then processed if the arguments are present. Logical instructions which are negations of their counterpart, call that counterpart and shift the negated bit; for example, bic calls and, orn calls orr, eon calls eor.

Finally, data transfer instructions are split into str and ldr. Apart from when loading literals, ldr is the same as str with the load bit set. Load literal cases are dealt with directly in ldr by shifting the offsets and calculting the offset of the label. When doing other loads and stores, we must check the addressing mode. This is done using a combination of regex, sscanf, and manual checks (for example the third argument exists if and only if the transfer is post-index, and only pre-indexed transfer have have '!' at the end of the second argument). Despite this, there are some tests failing because they choose the wrong addressing mode, this difficulty is caused by operands being surrounded by square brackets, which merges two arguments into one; if we had time, we would have replaced instances of sscanf into regex to check which addressing mode the operand is followed strtok to tokenize it because we believe sscanf is improperly matching the input. Alternativley, we could change the tokenizer to split the operand more succinctley, but this might break other tests.

1.3 File Organisation

Our src directory is divided into the assembler, emulator, and blink subdirectories. The assembler itself is split up into the main program, assemble, the three types of instruction types, branch, data processing, and data transfer.c, in additon to the tokenizer, and symbol the symbol table data structure along and its functions. Special functions and directives are put in data transfer as there is only one of each. Each .c file has a related header file with all non-local function prototypes. A makefile is included in the src directory which calls the makefiles in assembler and emulator.

Currently these makefile manually specify which files to compile, but it would have been better to use variables and loops to automate this proccess, perhaps this could be done by having a single makefile in src which goes through each directory and compiles file. Furthermore, it would be beneficial to make a shared library for the utils functions and constants that are used in both the emulator and assembler. This would remove code duplication for functions like generateMask. This library would mostly contain the bit manipulation functions, since they are espcially portable.

2 Blinking the Pi

3 Testing Strategy

Compared to the emulator, we tested the assembler far more effectivley. During the time we were writing the main function, we partitioned the logic into its main components: reading the assembly file, parsing and tokenizing each line of the buffer, hashing the opcodes, and writing the instructions to a binary file. Each was in turn tested and debugged. This allowed us to fix and segmentation faults and correctness errors before we started the logic for the assembly functions so that they could be tested as quickly as possible. Furthemore, we carried out seperate testing for utility functions such as getRegNum, checking for the correct output for general and boundary cases. Since the test suite specifies the addresses of the incorrect functions as well as their values, it was relativley easy to track down the source of bugs by comparing which bits where incorrect with the layout format detailed in the spec. In constrast to the black box testing of the test suite, our use of debug messages to output variable values and step-through debugging to monitor which parts of the code were being executed allowed us to efficently carry out white box testing for each function. Issues were posted on GitLab or communicated through Discord to be dealt with. We did not create our own tests, since we believe the test suite is sufficently thorough. Overall, we ended with 592/594 tests passing for the assembler and 588/594 passing for the emulator.

4 Group Reflection

Our main method of online communication has been Discord but we have also met frequently to work together, especially during debugging. Addtionally, our group members have become more competent with the features Git and GitLab provide as the project has progressed, which has allowed us to collaborate effecitvley and divide the work up appropriatley. Firstly, we have continued to make use of issues by assigned them to individuals and labeling them suitably. Not every issue was filed formally on GitLab as we often discussed them in person or on Discord. In the future, when tackling larger scale projects, such as the PintOS, it would be benefical to add better description to our issues (maybe create a template for issues and merge requests to standardise them), reference the code that they pertain to, and create branches for each issue so they can be fixed and merged without interfering. A board was created on GitLab with some bespoke labels to differentiate issues for the emulator, assembler, and Pi; though it might have been better to do this on a third-party service like Trello, due to the increased versatility it offers. Furthermore, we have been using branches and merge requests for new features. Occasionally, we had code reviews when a feature was completed so that it could be approved but since we they were large updates, it was harder to productivly review. So we might next time split new features into smaller chunks so the changes are easier to understand and the merges easier to approve. For the assembler, we created the aptly-named divergent branch 'assembler' and commited and merged any updates to the assembler there. However, this meant that we had alot of commits to rebase when we finally merged the assembler branch back into master. Therefore, it would have been better not to create the assembler branch and merge new updates directly into the master branch.

With regards to the workload of the assembler, at first we allocated each member with a different aspect of the main function (as mentioned previously, this was to read the assembly file, write the binay file, tokenize the parsed lines, and create a hash function to be used for optimal access of function pointers) so that we would be in a position to test quickly. We believe this was the right couse of action because each part could be functionally modularised and we could indivually implement our own part without knowledge of the others, as a sort of black box. When this stage was complete, we performed the same process on the types of instruction. Overall, our workflow was much more streamlined than for the emulator. When all the code was written, we gathered together to test it; each member was assigned an incorrect test (which we believed to have a different source of error, otherwise two people might be fixing the same problem) until we whittled down the number of incorrect tests. Last of all, we regularly used pair programming to help debug because it allowed us to discuss the code and catch errors that others might have missed.
Throughout development, we used cbuild and a bash script to move the tests to the test suite folder, then had to run the test suite everytime. If we had more time, or were undertaking a longer proccess, we could use CI/CD pipelines on GitLab to automate testing, format all the incorrect tests into a single file with the differences between the expected and actual output. Furthermore, we probably should have created a discrete staging and deplyoment environments; in the staging environment we would keep debug messages to quickly identify the cause of problems when the arise, meanwhile the deployment environment would not be used for debugging and thus be free of the clutter of debug messages. This effect could be achieved with both GitLab environments (more elegant), or by having two parallel branches (far less elegant).

5 Individual Reflections

5.1 Ruben Vorster

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

5.2 Arron Stothart

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod incididunt ut labore et dolore magna aliqua.

5.3 Pip Moss

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

5.4 Faris Chaudhry

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.