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 arg3 exists if and only if the transfer is post-index, only pre-indexed transfer have have '!' at the end).

1.3 File Organisation

Our src/ directory is divided into the assembler, emulator, and blink for the Pi. A makefile is included in the main 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 each c 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 bit manipulation functions, since they are espcially portable.

2 Testing Strategy

Compared to the emulator, we used test suite far more effectivley. Although we did not create our own tests,

3 Design Details

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

4 Group Reflections

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

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 tempor 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.