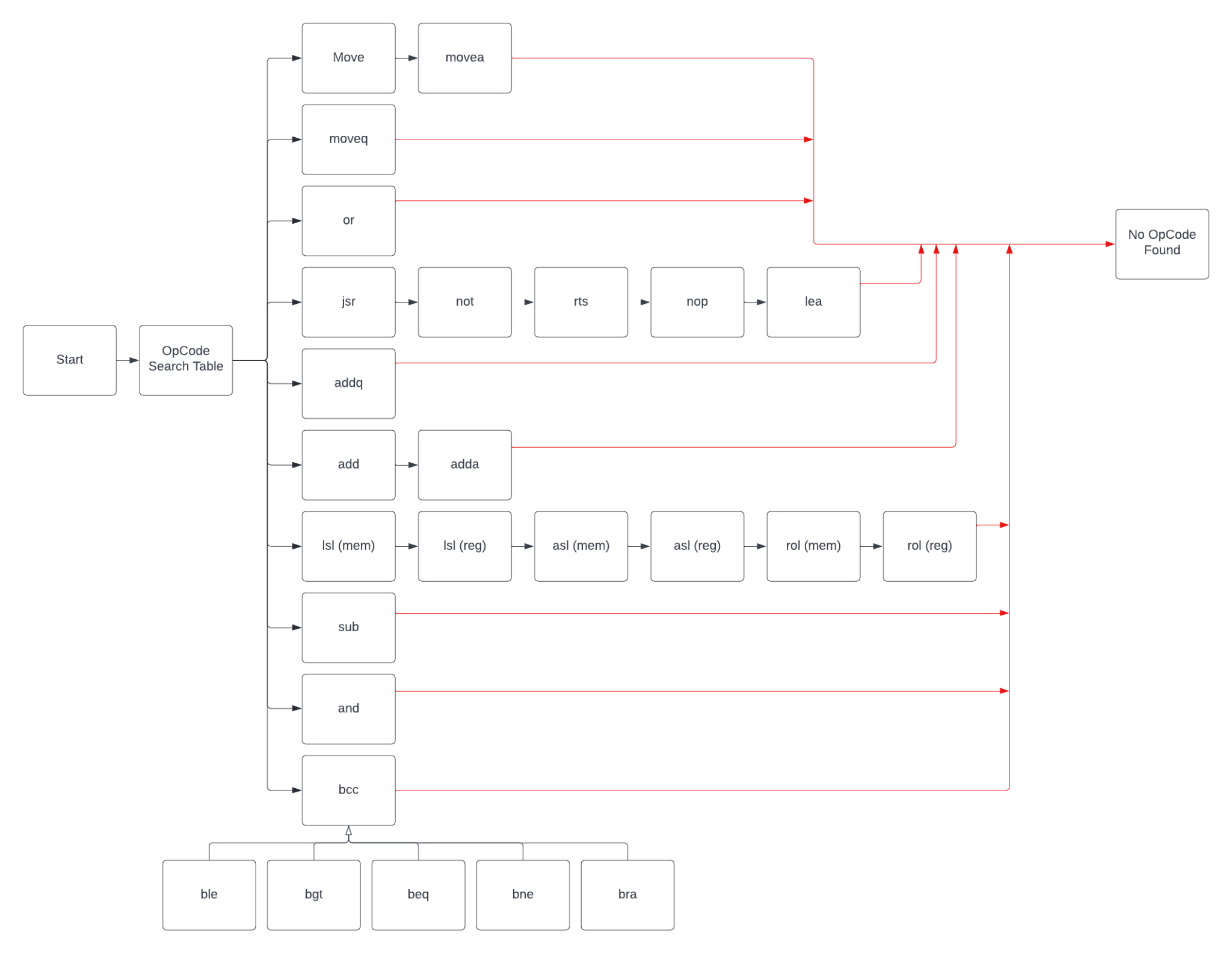
Team Teraflop

Project Report

Brennan Richards, Martin Rudzis, Sarah Asad

1) **Program description**:



Program Input:

The program input is read from the user using a combination of three primary subroutines. First, there is a subroutine that prompts the user to enter an address and stores it as a string of ASCII characters. From within this subroutine, it checks whether the input is valid. If the input is not valid, it clears the data that has been stored and starts over again, prompting the user for input. If the input is valid, the program then moves to a subroutine for converting the input from ASCII codes to hexadecimal digits. This subroutine performs the conversion by starting at the beginning address of the input string, subtracting the hexadecimal value 30 from the first byte, then moving to the next byte address, and so subtracting 30 from each character in the entire string. However, as it performs this task, it also checks whether the resulting value is between 10 and 16, inclusive. In this event, it moves A, B, C, D, E, or F as a byte value into the location, as appropriate for a correct conversion to hexadecimal. Upon reaching the end of the string, it moves to the next subroutine, which handles combining each hexadecimal digit together into a single number representing an address. This is performed by copying each bit value in the input string one-by-one into a data register, shifting the register contents one bit to the left, and so repeating until all of the hexadecimal digits are in the data register. The content of this data register can then be used by the disassembler to determine the start or end addresses. This entire process is repeated a second time for reading the end address.

Identifying Opcodes:

Beginning at the starting address the program will check if the current address we are pointing to is past the end address, or if we have output 15 opcodes. If so either we branch to the respective subroutines (enter new addresses or print next lines). If both of those are no we pull the first 2 bytes of memory into the d0 register. It will then load $a000 into a2 which will be the memory address of where we store the binary value of the 2 bytes (This should have been saved to an address that was less than $8000 but we only saw that requirement in the demo test after the program had been completed). Every time a bit is rotated out the carry flag is checked and 00 or 01 is moved to the place a2 is pointing at and is also increment by 1. At the end of the bit rotation, $a000 - $a00f will contain all 16 bits of the 2-byte data.

From here we branch to our opcode table which is a long switch statement that compares the first 4 bits in $a000 to possible opcodes based on our table we made for progress report 2. Some opcodes are tied to each other like asl, lsl, and rol. If the first 4 bits don't match any in the switch statement we branch to the no opcode subroutine which displays DATA and then the 2-byte data by decrementing the current memory pointer by 2 and moving it into d1 to be printed with trap 15. This method is called throughout the program in opcode subroutines if the 16 bits we are checking don't match with any opcodes. If we do find a match for the first 4 bits we branch to that subroutine and start verification. Verification is specific to each opcode but the general strategy was checking specific bits by loading $a00x into the a2 and calling the read4bits, read3bits, read2bits subroutines. That would then store the sum of the bits into d3 which we could then check for validation. If we found the bits to be invalid we would either branch to the next subroutine in most cases (as seen in the flow chart above), or we would branch to the no opcode subroutine. During this validation phase, we would also check the effective address to make sure it was valid. By valid we mean everything the manual said the opcode could run with. If the mode was 5 or 6 we would output invalid EA even if it was valid for those opcodes. This is because they weren’t required to address modes for this program

Once validation was done we would start printing the opcode and the effective address out. Since the opcode subroutine does all the verification the effective address subroutine assumes all valid input and simply prints the data being passed in.

Effective Address Subroutines:

There are two set up subroutines to call when wanting to print out an effective address. The source and destination subroutine. The source assumes the mode is before the register and the destination is the reverse. This was needed for the move opcode. Both subroutines move the mode and registers into their respective data registers and then call the main effective address subroutine. This subroutine compares the mode saved in the data register with numbers to branch to the proper subroutine to print the effective address. If a matching mode is not found in this switch statement it will print out ‘INVALID EA’. An example of a subroutine is the dnsub that prints out ‘d’ and then the register # that is held in another data register. If the mode is 7 it will call a special subroutine that will handle if any data from the memory needs to be moved into a variable, as well as incrementing the current address pointer. This is the end of the effective address subroutines.

At the end of every opcode there is a checkvar subroutine call that is necessary to print out any data stored in variables that were set during the effective address subroutines. Once this is done the opcode subroutine is done and it branches back to scanning the next 2 bytes of code.

The only limitation of the program is that it is not a complete disassembler that has every opcode programmed into it. Because of this data tied to some opcodes get read and misinterpreted as valid opcodes when they are not.

2) **Specification**:

Our program begins by displaying input instructions and prompting the user for a starting address in hexadecimal. If the input is invalid, we prompt the user again for a starting address.

Valid input should follow these specifications:

* Should be hexadecimal format - numbers: 0-9 and capital letters: A-F.
* End address should be greater than the starting address.
* Address length should not be longer than 6 characters or less than 1 character.
* Address should end in 0, 2, 4. 6, 8, A, E, C.
* Should be within the range of 0 to FFFFFF.

The input program verifies that some of these specifications are followed, but not all of them, so the user must strictly adhere to the instructions. If the address is valid, we store the value. The same process is applied to get the end address from the user. We take care of the case where the end address is greater than the starting address. In this scenario, we prompt the user to re-enter the addresses. Our program then grabs the first hexadecimal value which is then converted to binary. This is where we then begin the opcode search.

We read the first hex value and then do a search in our opcode table. If the opcode is not found, we then print an error indicating an invalid opcode and then go to the next hex value. If the opcode is found, we jump into a loop of checking bits to verify that it is the correct opcode.

If the bits match an opcode, we then verify that the effective address is valid. We then move onto checking modes. If the mode is not 7, we jump straight to a subroutine which prints out all the data to the console.

Output to the console is printed in the form:

Memory Location Op-Code operand

If the mode is equal to 7, we check to make sure that the register is greater than 1. If it is not, we go straight to printing out all the data. If it is then we find the correct size which is either a byte, word, or long which is then stored and printed.

We then move onto the next value to be read. The entire process is repeated until the end address is reached. The user can then either enter a new starting and ending address or can quit the program and exit.

3) **Test Plan**:

The testing process consisted of 4 separate parts. Testing each effective address, then testing the opcodes, testing user input whenever working on the input subroutine, and finally testing the end program once all three parts were connected together. The user input specifically could be tested at any time since it could be connected to the main program through branching. The opcodes could only be tested once the effective address subroutines had been completed.

Testing the effective address consisted of writing the subroutines for the required opcodes first. Next, a move opcode was read and converted to binary which is how the effective address subroutines know what to print out. We chose to use the move opcode for testing the effective address subroutines because it has two effective addresses and is able to use all modes and registers. For example, by changing the move opcode to move.b #100,d0 or move.l d2,(a1) we were able to test all possible variations for effective address calls. We were also able to test effective address calls when the mode was either in front or behind the register number.

Testing the opcodes was where most of the testing happened. Since the effective address assumes all passed in data is valid it took very little time to test. Because of this, all the verification of specific bits had to happen in the specific opcode subroutines. For each subroutine that was written, it was tested for varying sizes, each of its respective addresses, and any other opcode specifics. Such as ROL/ROR and reading the direction the rotation is. This process made sure the opcodes were tested thoroughly. The only thing this method required was time. Once this was done it was tested with the demo file to make sure again there were no problems. This was able to be done without the input subroutines by manually inputting the starting and end addresses into the program.

Input Testing consisted of entering different types of invalid user inputs and testing that the validation of those inputs were successful. Testing started with first ensuring that the program would ask for new input when the user provided an address containing no characters or more than six characters. Next, we added verification for the types of characters entered. Testing simply involved making sure that the program prompted for new input when ASCII values outside the range of 0-15 (0-F in hexadecimal) were stored due to user input. Lastly, we checked that the starting address was less than the ending address. Testing involved providing hexadecimal starting addresses that were out of order and ensuring that the program started over, prompted for input, and functioned as before.

Testing the entire program once all parts had been connected consisted of making sure the varying loops were functioning. These loops consisted of the number of times an opcode is displayed to the console before asking to see the next collection of opcodes and the loop back to the beginning of the program if the user wanted to enter new start and end addresses. Once this was done the majority of the testing was done and the only thing left to do would be to fix minor bugs if we saw any and structure the code more uniformly.

Our coding standards were making sure we left comments on our code so teammates would have an easy time understanding what the code meant, and if needed ask questions about specific areas, or make suggestions. Another coding standard was making subroutines for things we would be doing a lot of to hopefully conserve space. One of these being printstr which did the move.b #14 trap #15 for you without having to write it out every time. Another thing would be communicating bugs when we found them so whoever’s part it was coming from could hopefully get to bug fixing as soon as possible.

Unfortunately, the tests that were used for the opcodes were made for each opcode specifically while it was being worked on and we don’t have a master file with all of them. The file we used for final testing was the demo file provided. This was used to test the functionality of the program once all parts were put together.

4) **Exception report**: This is your opportunity to describe problems that you’ve encountered but couldn't fix, or chose not to fix. Anything that you feel deviates from your intended program. Also, this is where you can describe what you were able to complete in the time allotted versus what the assignment asked for. This should definitely include the results of your testing if you found defects but didn't fix them. In an ideal situation, I should be able to just read your documentation and your source listing and give you a grade, without needing to run the program. Of course, I will run the program, but I hope that you get the idea.

The only problem that arose was when the program would read an opcode it didn’t recognize, output that it was an unrecognized opcode, and then falsy recognize the data tied to that opcode. For example, if you had the CMP opcode it wouldn’t be recognized. But if #$5678 followed it then that would get falsely recognized as addq. This is only a problem because this program is not a complete disassembler. If this was a complete disassembler with every opCode in 68k programmed in this problem wouldn’t arise. Another limitation is that the bits of the 2-byte data are saved at $a000. We did not realize this would be a problem until we saw the demo file that asked the code be contained to below $8000. Another exception is that the user input has to end in 0, 2, 4, 6, 8, a, c, or e.

5) **Team assignments and report**:

We assigned work at the end of each meeting. We would go over all the things that were done the previous week and then talk about our next steps. These meetings usually happened Monday so we could work on the progress reports if they were due. Coding wise Martin did 35% with handling user input as well as some opcodes. Brennan did 40% with handling the effective address subroutines as well as some opcodes. Sarrah did 25% with handling the remaining opcodes but did a large portion of the planning for the general program flow.