**Team Name: Deux Poissons**

Kien Chin & Juliano Nguyen

July 2, 2017

Final Project Documentation [CSS422, Spring 2017]

**Program Description**

Our design philosophy was that we would look at the bits of each opcode, and those with similarities were able to reuse some of the methods to print out its EA section. By looking at the bits of the opcode, we were able to create a table that classifies an opcode based on the first 4 bits. Opcodes that did not have any similarities would have special EA cases that we had to make. Our flowchart is well detailed that describes how the program works. We split it into three separate flowcharts that dealt with the Input/Output section, Opcode section, and EA section. We made the I/O section will go to the opcode section once the user inputted the addresses. Then, the opcode section will get the first four bits of an opcode and look at the table of opcodes to find the matching first four bits. Then, it will go to the EA section to print out the source and destination of that line. Algorithms we are proud of are the use of the AND operation, the HEXTOASC, and ASCTOHEX method. The AND operation is used in our program to figure out the correct bits to determine if it belongs to the opcode, the size of it, and the EA section. We referred to the manual to know which bits we are looking for so that the program can jumpto the correct subroutine label. For example, we used the AND operation to find the first 4 bits when it reads a line from the test file to figure out what that opcode is classified as in our opcode table, such as MOVE.B has 0001 for the first 4 bits. We made a HEXTOASC method, which is a convertor that turns a hexadecimal value into an ASCII value, so that the program can output the correct value from memory instead of its decimal value. For example, it outputs the hexadecimal value $12345678 as the destination in the ADD operation. Another method is the ASCTOHEX, which was built from scratch by comparing the functionality of the HEXTOASC and reversing it. This simply converts an ASCII value into a hexadecimal value that is to be outputted; for example, our address locations per line. For both of these convertors, we utilized an online ASCII table to determine the correct hex value that needed to be added to convert a character into a letter from A to F. We utilized the TRAP tasks 1, 2, 3, 4, 5, and 14 in our program. Most of these TRAP tasks were used to allow users to input the addresses, display a value or a string in the output, and input the enter key.

The limitations that our program has is that it does not check for lowercase letters for the inputted addresses.

**Specification**

Our program displays a welcome message to the user and prompts them to enter a valid starting address and ending address. The program checks if the inputted addresses are valid or not; if it is invalid, then the program will output an error and prompt the user to re-enter the starting and ending addresses. For example, the user cannot enter an odd address, a higher starting address than the ending address, or an address that is not 8 hexadecimals long. Then, the program will disassemble the test file by outputting the address location, the opcode name, and its respective effective address. First, the code will get the first four hexadecimal digits from memory and store it in a data register, then it will jump to an opcode table in order to determine which opcode that is to be printed out. For example, ADD’s first four bits is 1101, so our program will get the appropriate bits from the data register to determine if its an ADD opcode or not. Then, if necessary, the code will determine the size of the data that the opcode will be working with, such as byte, word, or long. Finally, the code will branch to unique EA sections that is designed for that particular opcode. If an opcode has similarities for its EA section with another opcode, then the methods were reused to print out the EA section. If the program encounters an invalid opcode that was not required, then it will output DATA. The user can hit the “ENTER” key to display the next sequence of opcodes (10 lines) until it reaches the ending address. The user can press any other key than the “ENTER” key so that the program can prompt the user to quit or restart the program. If the user quits by hitting the “Q” key, then the program will end and display an ending message. If the user restarts by hitting any key other than “Q”, then the program will prompt the user to enter the starting and ending address.

**Test Plan**

We tested the program by having all of our files in one folder, running the main disassembler file, executing the program, opening data to select our testing file, then executing it once more to launch the program. We tested the invalid addresses, such as not being 8 hexadecimals long or odd address inputs. We tested our own test file as well as the demo file, then we hit the enter key until it reaches to the end of the file. In terms of our coding standards, we would have a block comment at the top of each main label, such as our table to determine the correct opcode, that describes what that specific section of code does. We also would put comments at the beginning of each of our opcode or EA sections to list which data and address registers we are using, so that we do not forget or replace it with another value. This helps us know which registers were previously used in another method. We also separated files that had a long code section, such as our general EA subroutines, so that we can organize what each file contains and makes our main program more cleaner. We also made some side comments throughout the code, so that we do not forget what each small section of code does (like what it prints out), and it helps us observe what is happening in the code manually. Another coding standard is that we made labels in all caps and used underscores with a different name to differentiate similar labels that were in the same section for an opcode. We made labels that were appropriate for the opcode we were doing, so that if we were dealing with the MOVE opcode, then related methods were labeled with MOVE. Furthermore, some of the sections had a block of asterisks, to allow us to know that block of code and all of the methods are within its own section.

**Exception Report**

In the time allotted, we were able to create ASCII art for our welcome message when the user starts the program. We added in other opcodes that were not required, such as CLR and SUBA, because we wrote comments in our code that listed some opcodes that were not from the required list. However, problems that we encountered was the opcode for ADDI and ADDQ, where if the immediate value was below 8, then it would automatically use the ADDQ opcode if the testing file listed it as ADDI. We spent time trying to figure it out, but it was Easy68k interpreting it as ADDQ. We chose not to implement the input error checking when a user inputs a lowercase letter in one of the addresses due to time constraints. We also print out the size for any of the Bcc opcodes that were required, where we usually just call, for example, BLT without the size. We found a defect during our testing, where it would usually print out an address location that is not from our testing file with DATA, but it continue to output the rest of the opcodes after that normally. We also encountered a problem where when the opcode is MOVE.L and the destination is only a word, $1234, then it will print out $1234XXXX, where X is numbers from memory. The program kept jumping to our long value subroutine and we were unable to fix it.

**Team assignments and report**

We organized our tasks by creating an excel sheet of the required opcodes and EA section. We started by creating a flowchart together on how the program works, what it does, and what it needed to output. We also created another sheet to list out all possible valid EA with their respective opcode. We updated the flowcharts when we discovered a new approach when we coded the program. For the majority of the program, we met in person every other day after class to work on the required opcodes and EA section. We checked off on our excel sheet which opcodes were completed and prioritized the next opcodes we need to do. We communicate with each other often on Discord to update what has been added to the program. When we were doing our own separate section of the program, we also explain to each other what problems we encountered, so that when we meet in person, we address those problems as soon as possible. We would say that 90% of the work was pair programming, where one of us explains how we should approach an opcode section and the other types up the code. We would switch roles every other day when we met. This allows us to to look at each other’s mistypes or spot any bugs as one types. Individually, Juliano’s part of the work would be 6% and Kien’s would be 4%. Juliano was responsible for doing the Input/Output section of the program. Juliano wrote our own test file that lists every possible valid opcode and their respective EA section. Kien was responsible for handling the special EA cases for some of the opcodes. Kien generalized our EA section of the program for opcodes that had similar coding from the manual, so that we can reuse those EA sections without having to create each EA section for each opcode.