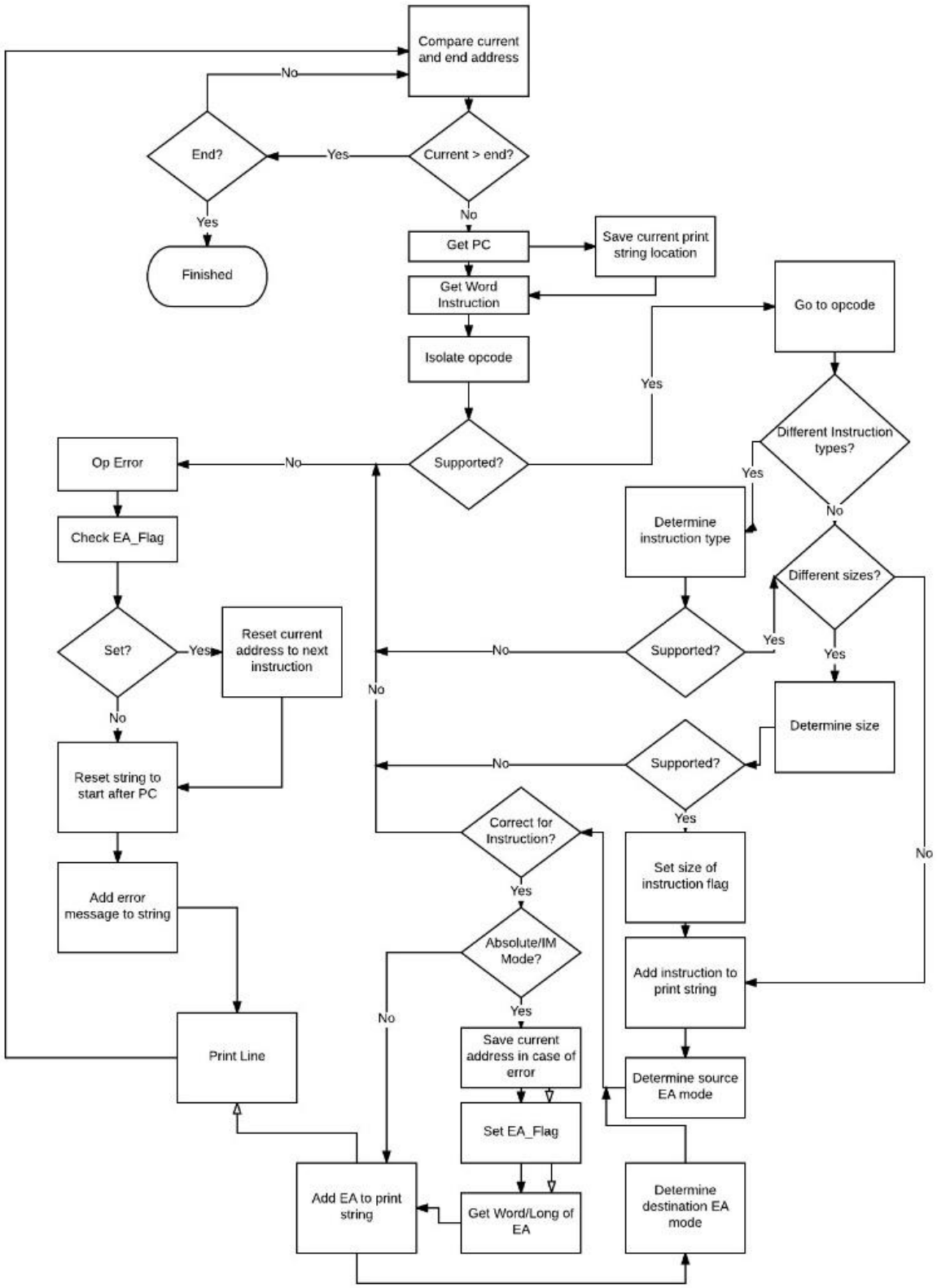
Kyle Farmer
CSS 422 – Spring 2016

Disassembler Project Documentation

Program Description:

I did some research before beginning the project to figure out how disassemblers are typically organized and what goes into the phases of decoding an instruction. This project was already difficult given that we not only had to learn a new assembly language, but also become good enough at it to disassemble every op code (but not every instruction). I found out that the basic idea was to fetch the opcode, and then filter instruction for bit signatures (such as mode), and then extract the operands in the instruction. Since op codes were in 4 bit pieces, and the modes/operands were typically in 3 bit pieces, I wrote three functions that take the first bits of of the instruction and added them up using a bit test. These functions was used in every part of the decoding. The flow of my program is a bit complicated in the diagram below. However, the basic flow was to gather the user input for starting and ending addresses, making sure they are not odd, have correct amount of characters, within bounds, and no invalid characters. I would then gather the instruction for a given PC and then swap the instruction to the front 16 bits. Then I would call the bit test function to return the opcode, then shift out the opcode after I was done with it. I would repeat this process throughout the instructions and for the various parts that I needed to extract from the instruction. The decision to do this was not because of performance but because it made more sense to me rather than shifting left and shifting right back to get the correct number. After getting the correct opcode and jumping to that opcode from a table, it would find the characteristics for that instruction word. If it these characteristics were missing, the program would branch to an error which displayed the word data that unable to be parsed and moved to the next instruction. If the instruction was able to be validated, it would branch to an EA table where it found the correct EA for the instruction. Again, if an incorrect EA was found, it would branch to an error and read the next word instruction. After the program was finished decoding all of the instructions for a given memory range, it would ask the user to loop back to the beginning and enter new starting and ending addresses or quit.

Flow chart of main:



Specification:

The program would start by prompting the user to enter a starting address. If the starting address did not go beyond the bounds of the upper range, had less than 8 characters, and more than 0, had valid characters of 0-9, a-f, and A-F, and not an odd number, it was accepted. Then the user was prompted for the ending address which had the same checks as the starting address, except the it could not be lower than what the user entered for the starting address. The program then entered the main loop where it would first gather the PC and add it to the string. All of the op codes in 68k are supported but not every instruction. The following instructions are supported:

1. 0000: ADDI, ANDI, CMPI, ORI
2. 0001: MOVE.B
3. 0010: MOVE.L/MOVEA.L
4. 0011: MOVE.W/MOVEA.W
5. 0100: LEA, CLR, JSR, RTS, NOP, MOVEM
6. 0101: ADDQ, SUBQ
7. 0110: BRA, BCC, BCS, BNE, BLE, BLT, BGE, BGT
8. 0111: MOVEQ
9. 1000: DIVU, DIVS, OR
10. 1001: SUB
11. 1011: CMP
12. 1100: MULS, AND
13. 1101: ADD, ADDA
14. 1110: ASR, ASL, LSR, LSL, ROR, ROL, ROXR, ROXL

Supported instructions are output in the following format:

a- Current Address b- Op-code c- Operands

If a word is encountered that the program is unable to parse as an instruction, it prints the data value in hexadecimal format as the operand, “DATA” as the op-code, and the current address in memory and reads the next instruction word. On branch instructions, it shows which type of branch is used, if supported, and then the displacement in hexadecimal format following the type of branch. The 68k EA types supported are:

1. Data Register Direct

2. Address Register Direct
3. Address Register Indirect
4. Address Register Indirect with Post increment
5. Address Register Indirect with Pre decrement
6. Absolute Word Address
7. Absolute Long Address
8. Immediate Data

After the program has displayed 30 lines of instructions, it prompts the user to press enter to continue to display the next 30 lines. This repeats until the disassembly is finished. When the disassembly is finished, the program asks the user to repeat from the beginning with new starting and ending addresses, or it terminates.

Test Plan:

I did not specifically make a test program for my program, which probably made the debugging and testing of my program take longer. After I was able to gather user input for beginning and ending addresses, the way I tested the op codes was to place the op code I was focusing on as the first line in my disassembler and run through the debugger to see where the instruction was breaking down and fix it from there. Then I looked at the op code manual and saw which types of modes were invalid for a given instruction. For each of these instructions, I included these invalid modes in each instruction so that it would break to an error if it was encountered. I then made more rigorous tests for each instruction by testing the combinations of source and destination types. Finally, for the instructions I had time, I would actually put random data into a memory location that would have to be caught by the op code. For example, 68k assembler would interpret the instruction OR.B #$10,D1 to ORI.B. So I would place the data 1000 0010 0011 1100 (823C), which would be the equivalent if the assembler did not optimize the instruction. I check for the signature 11 1100, and throw an error. If I wanted to make it more complicated, I could re direct this towards an ORI instruction, but there is only so much time in a quarter. Also to test for invalid instructions, I would place 1000 1001 1100 0001 (89C1) at a memory location and run the program over that location. This instruction would be interpreted OR.B D1,#$10 if the assembler did not catch the error, but it is still invalid.

Exception Report:

I had some trouble with effective addressing. Extended instructions and instructions that had absolute addressing really was a big problem. It also took me awhile to figure out how to do the EA for the MOVEM instruction. Unfortunately, I did not figure out a decent method for this until it was too late. In the project I submitted, the method I had in place was not tested and I already know that there is bug in that I did not increment the current address when I read the extended word that held the registers for MOVEM. I also had a problem with outputting instructions, and random data, that were not supported. It seemed that if the instruction was determined early that it was not supported, this was not a problem. But I think that it was problem with the way I was incrementing the current address when the current address was on some unsupported instruction or random data. Sometimes, this would be interpreted as an entirely different instruction output with some registers that didn't make sense. Overall, I think the program is fairly solid and almost all of the opcodes work correctly as they should.