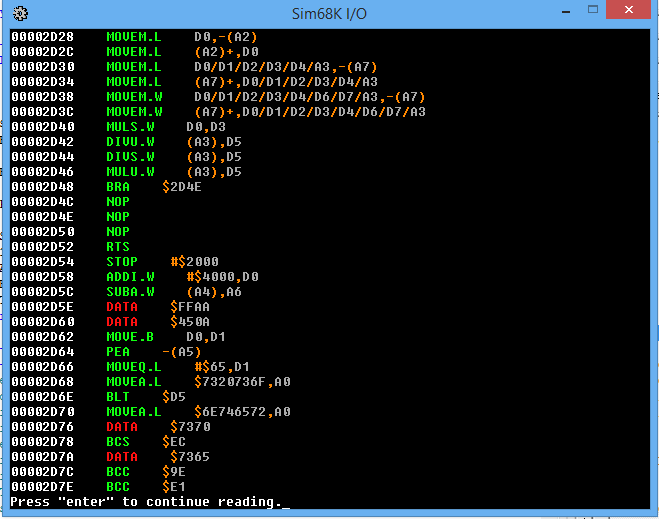
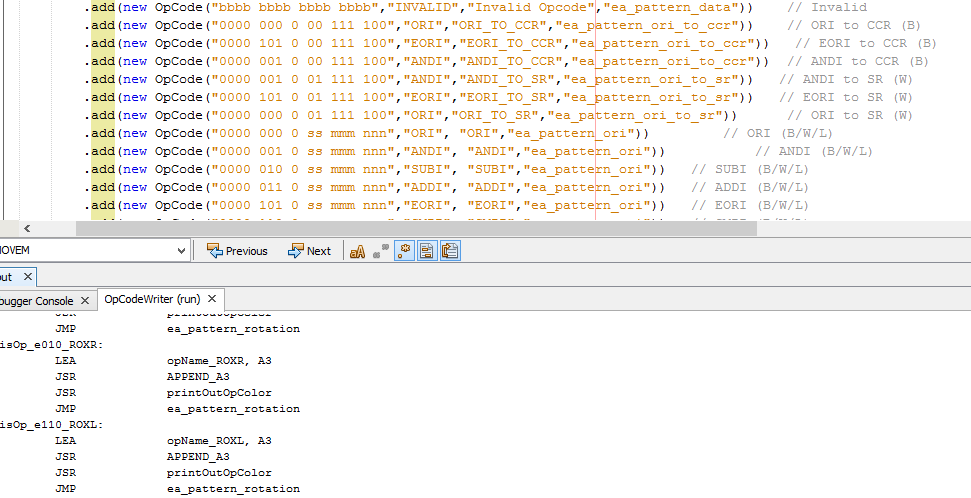
Team Chroma68k  
Members: Taylor Love  
CSS 422 A (Winter)  
Documentation  
March 13, 2016

# Program Description

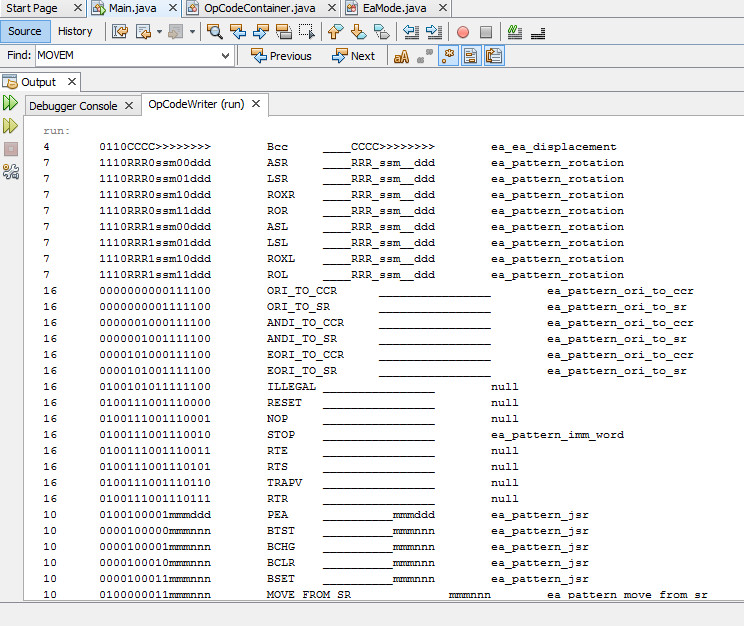
This program is written in Motorola 68000 assembly language (M68k), and its purpose is to disassemble data back into human readable opcodes and effective addresses. The program was set up to be easier to work with and use than simple black and white disassemblers. By adding color to the output display, the viewing quality is improved. Hence the name, “Chroma68k Disassembler”. Chroma means color. (Final version submitted has two lines commented out which, when uncommented, add color to the entire program again. Look for SET\_COLOR\_RAW subroutine) I wanted to add color to this because I personally couldn’t stand the black and white without syntax highlighting. Even with the added difficulty of doing this type of printing, I feel like it helped me structure the IO section in a much cleaner (modular) way than I would have done otherwise.



For a design philosophy, I decided I didn’t want to take any chances on messing up the project, so I wanted to finish the project earlier rather than later. To make the OpCodes section I wrote a program in Java that takes in opcode formats and prints out the opcodes section of the project. This was really helpful for also identifying patterns between different opcode formats, and to help identify where certain EA patterns could be re-used, and it could also be used to identify certain coverage gaps where I’d need to watch out for illegal opcode + EA combinations. It also eased making quick major changes to the op-determine section.



I did not use any algorithms from outside sources, except for the “cool” splash screen and “cool” ask to go again message screen. Those were made from a modified version of the bitmap painting example located on the easy68k website. I was particularly proud of the IO sections of the project. I got everything going through the same output subroutine by always printing to the buffer even when using long strings. Also by colorizing symbols and EA codes and addresses and opcodes differently.



# Specification

The program begins by welcoming the user with a “splash” screen. The splash screen shows the Chroma68k welcome screen and tells the user to tap “enter” to begin. The screen then clears, and the user is given instructions for proper input regarding starting and ending addresses. These addresses are validated before continuing to the opcode loop. If the addresses are invalid, or if the ending address comes before the starting address, then the user must retry the input process.

After collecting start and ending addresses, the program then reads through the data to detect opcode words. If a specific opcode could not be determined, DATA will be assumed for the opword. For each detected opcode, an effective addressing pattern might be used as well. The current address, the effective addressing pattern, and the opcode as well as size hints may be printed to the screen as needed. If it is “data”, then the contents of the data will be printed to screen. The print methods go through the IO portion of the program.

This continues until the next address to read from is greater than the ending address. At this point, the program reaches the “exit” phase where the user is asked, “Do you want to continue? Enter Y to continue.” If the user enters Y, the program will restart. Otherwise, the program finishes.

## Supported Opcodes

|  |  |  |  |
| --- | --- | --- | --- |
| **Required** | **Required** | **Extra** | **Extra** |
| **CMPI**  **CMP**  **SUB**  **AND**  **ADD**  **MOVEQ**  **DIVS\_WORD**  **MULS\_WORD**  **MOVEA**  **MOVE**  **ANDI**  **ANDI\_TO\_CCR**  **ANDI\_TO\_SR**  **NOP**  **MOVE\_FROM\_SR**  **MOVE\_TO\_CCR**  **MOVE\_TO\_SR**  **LEA**  **ADDA** | **SUBA**  **ASR**  **LSR**  **ROR**  **ASL**  **LSL**  **ROL**  **BCHG**  **JSR**  **RTS**  **ASR**  **ASL**  **LSR**  **LSL**  **ROR**  **ROL**  **MOVEM**  **Bcc** | ROXR  ROXL  ORI\_TO\_CCR  ORI\_TO\_SR  EORI\_TO\_CCR  EORI\_TO\_SR  ILLEGAL  RESET  STOP  RTE  TRAPV  TST  CMPA  OR  DIVU\_WORD  MULU\_WORD  EOR | RTR  PEA  BTST  BCLR  BSET  NBCD  TAS  JMP  ROXR  ROXL  ORI  SUBI  ADDI  EORI  NEGX  CLR  NEG  NOT |

## Supported EA

Data Register Direct

Address Register Direct

Address Register Indirect

Immediate Data

Address Register Indirect with Post incrementing

Address Register Indirect with Pre decrementing

Absolute Long Address  
Absolute Word Address

# Test Plan

**For testing the program, I placed opcodes with their EA modes within the TestData.x68 file, then I used dependency injection to swap out the** io\_address\_inputs.x68 **file include for the** io\_address\_inputs\_test.x68  **include file. The test file, rather than ask the user for their preferred inputs, just loads the effective addresses of specific branch labels (TD\_START/ TD\_STOP) to A5 and A6 where start and end addresses would normally be stored in an actual program run. Then, I let the program run through and analyze the test data appropriately.**

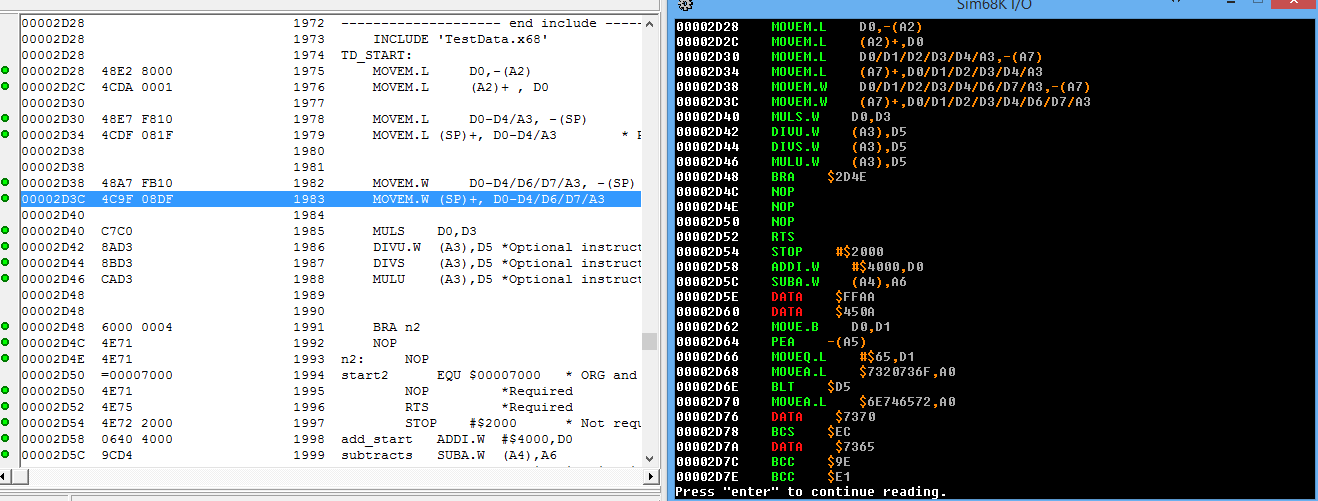
**The testing program contains what the professor recommended, as well as some more thorough testing code to supplement it. In large part the testing file acted as more of an archive though. For active testing, I’d put testing code in the MAIN file because it was easier to scroll down to look at the hex bits. Then I used the simulator to step through code and subroutines.**

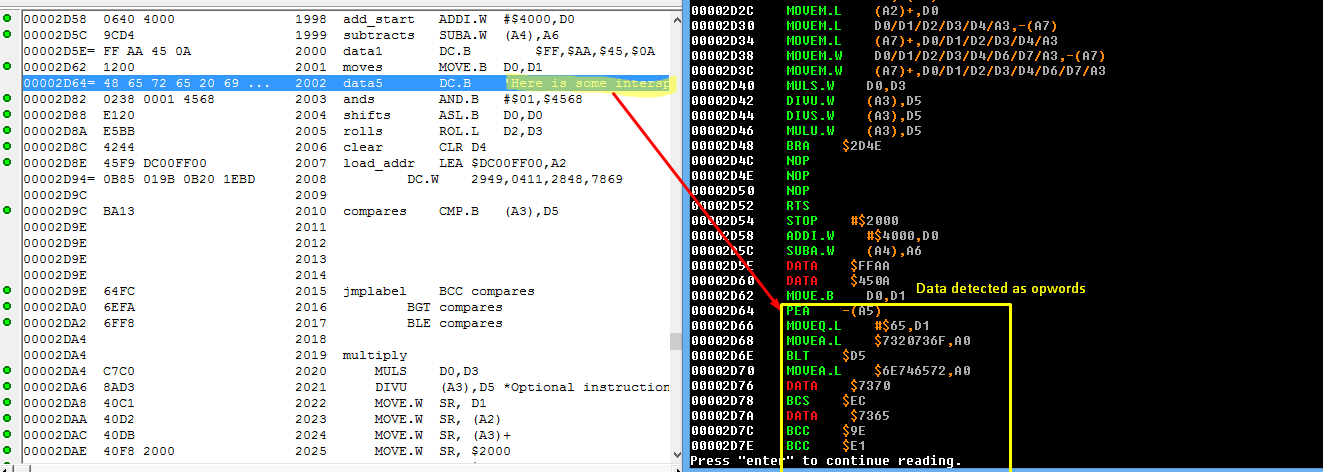
**For coding standards, I try to use MOVEM to push registers onto the stack when entering and leaving subroutines so that those methods can safely be called without messing up the registers from a calling method. Alternatively, I just make sure things are well documented. “D5 will be output. A3 is input.” Documentation like that.**

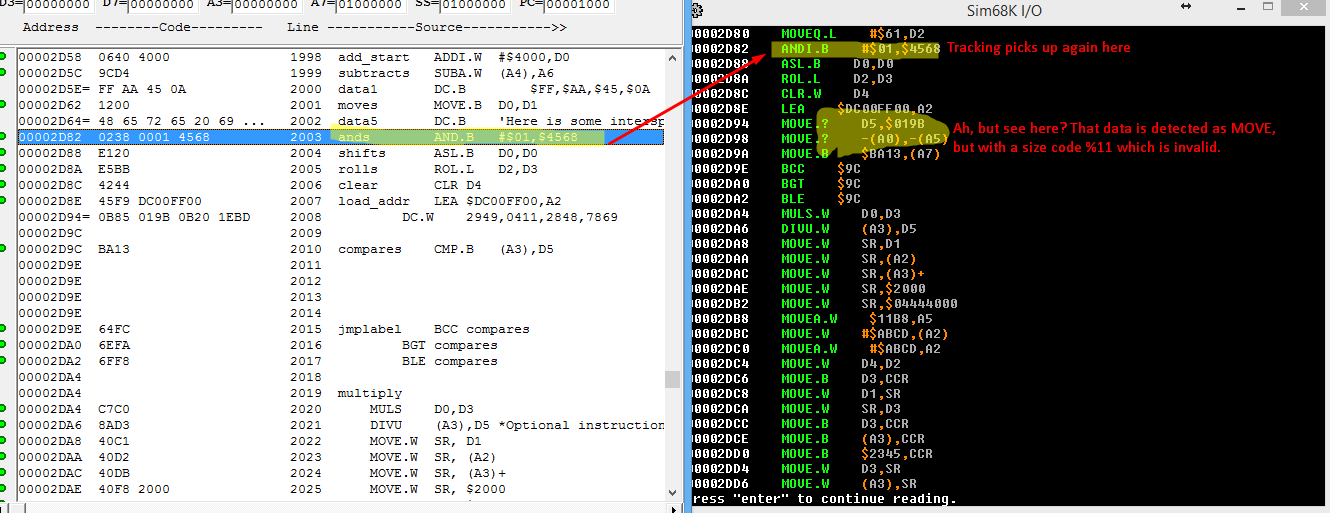
**I also try to separate huge sections with line comments. But… since I have a decent understanding of the assembly language codes and what they do, much of the code is uncommented. For example, something like “*LEA msgHelloWorld, A1”* wouldn’t have a comment because it’s pretty clear that you’re loading the address of the msgHelloWorld label to A1.**

# Exception Report

Right now, the program does not guard against invalid EA modes when performing effective addressing. This means that if you are loading in DATA, the program might not detect things as DATA because it instead detects it as a legitimate OpWord, but a legitimate opward that has an invalid addressing mode, or a bad size code. This leads to false positives for opword detection. When asking the professor about this behavior though, he did say no points would be marked off… I had hoped to have time to add a patch for this, but in the interest of getting documented and submitted on time I’m having to leave out this extra layer of opword invalidation. Even with this behavior, it did not seem to affect reading of other opcodes further down the page in memory.   
Also, not ALL ea modes are supported. (D16, PC) and other non-required EA modes are not supported. Sometimes\* the address input section glitches. Haven’t quite figured out what conditions are required to reproduce the glitch though. Most times it will work just fine as expected.







# Team Assignments and Report

**I did all the tasks on my own. I did 100% of the coding (Taylor Love), 100% of the opcodes, 100% of the EA, and 100% of the IO. Confidential evaluation report? I believe Taylor Love did a very good job with the IO section of the project. He also did a pretty good job with opcode detection. EA worked pretty well too, assuming the inputs were actual opcodes and not DATA that could be confused with actual opcodes using invalid addressing modes.**

