68k Disassembler

Thuan Tran

# What my program does?

My 68k disassembler can take in a listing file from a 68k source code, enter the address it want to start decode and the address where it should stop. Then it will print out the original code from the hex value it read in memory

Steps:

1. Execute the source code I provided. Load new data (Open Data), by navigating to the folder containing the listing file that you want to disassembler. Listing file are generate when you execute your 68k file
2. Enter starting address all in long format (8 characters: 00000000 – FFFFFFFF): 0 – 9, A-F. And remember that the ending must be greater that starting. And must not be odd
3. After you enter the ending address, the program will run for about 30 lines and stop. Waiting for you to press enter again to run another 30 lines. All the way until the end

Here is the format what it will print out

Memory location Opcode Operands

If there is an unsupported op code, it will print out

Memory Location DATA $The hex value that it can’t decode

If there is an unsupported EA mode, it will print out

Memory Location Opcode DATA $The opcode value in hex

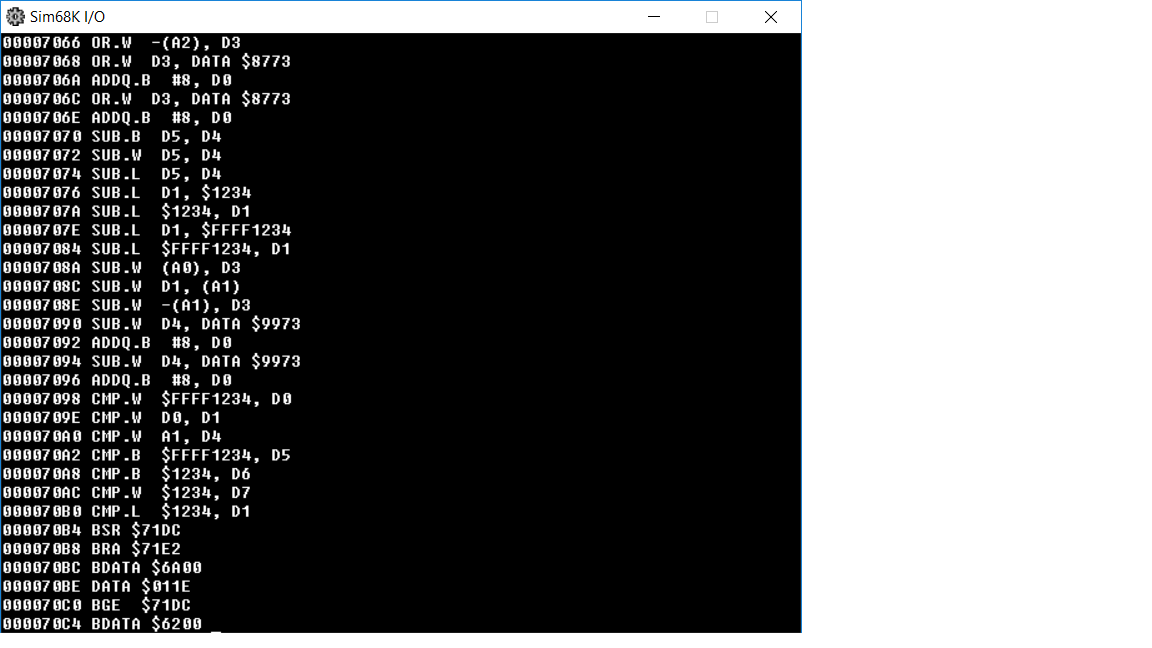
Please refer to the program description to know what operations it can support. And also what EA modes it can support

Figure What the program look like when running

# Coding Standard

Since I am a one-person team, I follow my own standard to be consistent throughout the program

## Alignment

Every source code and comment must be aligned to each other. Use Tab

## Separation

There are 3 main sections within the program. Separate them by using a long ‘\*’

For commenting method, use short ‘\*’

## Method calling

When calling a helper method, make sure the data that will be used within the method is prepare. Use anything as I see fit to store the temporary data. But there are some variables that should not be touch. If was touch, remember to restore them

* A2: Contains that address we are at
* A3: Contains the ending address
* TEMP\_CURRENT\_4\_NIBBLES: This temporary variables hold the opcode that is being decoded

And there are other temporary variables, but they can all be recovered by the above 3 variables. So, use them as I see fit. Since I can always restore them

## Helper method

If there is a recurring patter, create a jump table and helper method. Do this for anything I think appropriate

* Jump table for first nibble
* Jump table for EA mode
* Jump table to print out hex
* Helper method to get the first, second, third, fourth nibble