68k Disassembler

Thuan Tran

# Introduction

This project mimics the disassembling function of the easy68k program. What it does is that given a range of address in hex, assembled by writing instructions in 68k, it will disassemble (revert) them back to its original state

For example: In 68k, we can write MOVE.L #$025C30F7,D2 , which will be assembled to 243C 025C30F7

Feeding the hex address to the disassembler, it will return back the original MOVE.L #$025C30F7, D2 statement

# What my 68k disassembler can do

For this assignment, the 68k disassembler needs to be able to do the following commands

* MOVE, MOVEA, MOVEM
* ADD, ADDA
* SUB, SUBQ
* MULS, DIVS
* LEA
* OR, ORI
* NEG
* EOR
* LSR, LSL
* ASR, ASL
* ROL, ROR
* BCLR
* CMP, CMPI
* Bcc (BCS, BGE, BLT, BVC)
* BRA, JSR, RTS

Since I am a single person team, I do not need to implement: MOVEM, BCLR, EOR, ORI, OR. However, I was able to add some additional instructions:

* SUBA, BSR, MULU, ADDQ, OR, MOVEQ, NOP

And be able to handle the following address mode

* Data Register Direct:          Dn
* Address Register Direct:     An
* Address Register Indirect:  (An)
* Address Register Indirect with Post incrementing: (A0)+
* Address Register Indirect with Pre decrementing:  -(SP)
* Immediate Data: #
* Absolute Long Address: (xxx).L
* Absolute Word Address: (xxx).W

# Design section

In this section, I will go over my design strategy for the program, I will summarize them by words and then going to some diagrams:

## Overall program

I can separate my program into these 3 components:

* The input/parsing components: This component oversees getting user’s input (starting address, ending address), getting the next opcode
* The jump table components: This component oversees redirecting the flow of the program, by utilizing helpers to print out appropriate data. Some functionalities included in this section are: Hex jump table, first nibble jump table, data mode jump table, suffix jump table (byte, word, long)
* The print out components: This section oversees printing out the instructions, also performing logic in determining the suffix to print out, the data mode to print out, etc

## How the program flow

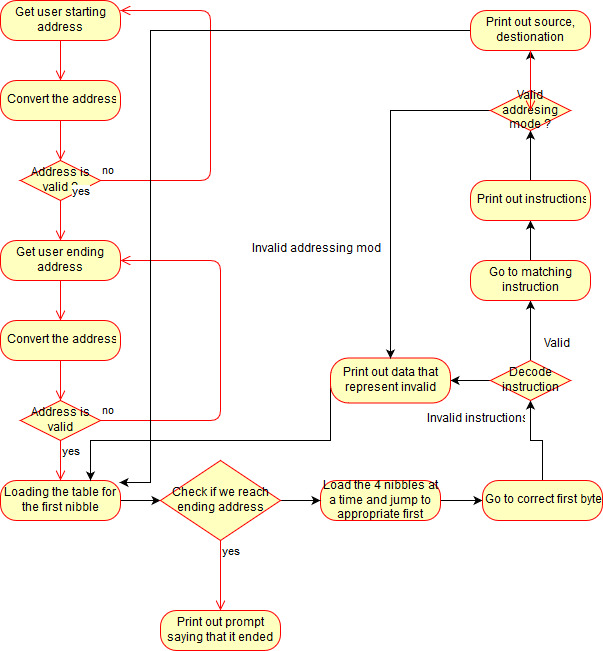
When starting to do the program, I realize that there is an emerging pattern that appear between instructions, in this section, I will describe that pattern that also leads to my implementation:

* All instructions are categorized by their first nibble: the left most 4 bit, by utilizing this, I can redirect my program flow to instructions that start with that nibbles. For example, MULS has the first nibble is C
* Going to the appropriate instructions, it was only a matter of comparing bits within the 4 nibbles opcode to determine what to print
* When going to the appropriate instruction, just load the hardcoded instruction and print out.

## Some of the knowledge that I get:

* Utilizing MOVE’s architecture, I can separate the remaining 12 bits ( after the 4 bits , first nibble) to get destination number, destination mode, source mode, source number. Utilizing a jump table allows me to print them out easily
* Utilizing the size bit, usually the 2 left most bit of the third nibble, allow me to print out .B, .W or .L . Thank you jump tables
* Same pattern with the conditions for Bcc where a condition jump table was created
* Some instructions requires explicitly data or address register, so I directly call it and print out the register number ( usually with instructions where the size bit are 11 )

Below is a simple diagram that represent the flow of the entire program



# Sections of code that I find particularly clever or difficulty

Realizing the emerging pattern, I create some utility function and jump table to help me with decoding: jump table for first nibble, getter for first, second, third, fourth nibble, jump table to print out the size suffix, jump table to print out ascii based on hex

There are also some difficulty as well, such as instructions that deal with 1-8 data. For those instructions, 000 is 8. So I need a special case

Instructions that was hardcoded with Dn or An in the destination number, while in fact, it will be the source instead. Something like SUB D5, $0000, the D5 will be on the destination number. But if it was DIVS D5,D4, then it follows the MOVE pattern where D5 is in the source section and D4 is in the destination section

Another difficulty is the Rotate instructions, depending on the direction they are rotating or they are rooting immediate or register ,..etc.

# Source that I used

<http://68k.hax.com/> Good website that helps explain in a formal way of how instructions works

<http://mrjester.hapisan.com/04_MC68/> Good website that explains in an informal way

<http://www.mwftr.com/ucF08/uc-07-68K-PROG.pdf> Great PDF that explains how to convert hex to ascii and vice versa

<http://goldencrystal.free.fr/M68kOpcodes-v2.3.pdf> Life saving document that goes into details how instruction works. Over 90% of my work is based on this