**68k Disassembler**

by Enbai Kuang, David Landry, and Preston Mar (group RandomPeople) for CSS 422 B, Fall 2018, University of Washington, Bothell.

**Contents:**

**I. Description 2**

**II. Specifications 3**

**III. Test Plan 4**

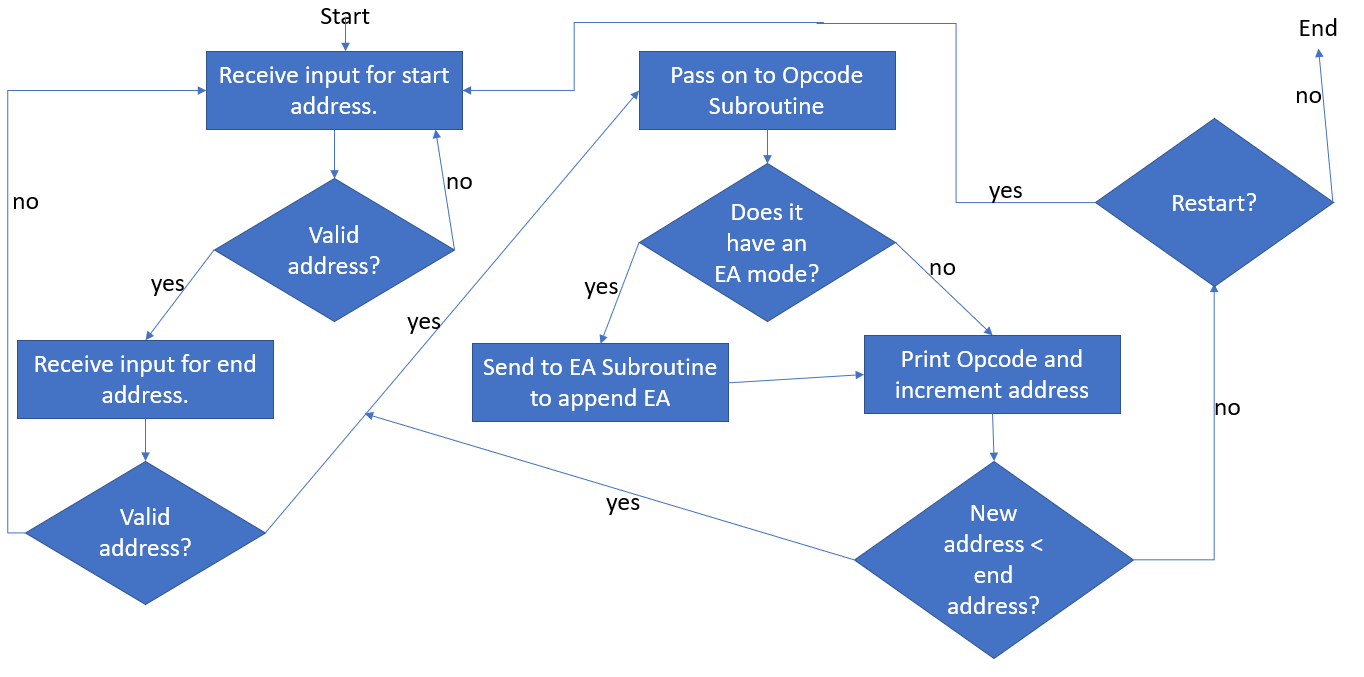
**IV. Exception Report 5**

**V. Team Assignments and Report 6**

**I. Description**

We the members of RandomPeople have created a disassembler. This program takes a segment of raw data and translates the range of input addresses to their equivalent 68k assembly code.

To approach this problem, we decided to divide labor into three main parts: input/output, opcode translation, and EA mode translation. Our input/output specialist, David Landry, would take user input, convert it to two hex numbers, and pass along the starting address to the Opcode subroutine. Our opcode specialist, Enbai Kuang, analyzed the raw data stored at the address for the equivalent opcode. He then would pass the opcode on to our EA specialist, Preston Mar, who would append the EA to the end of the opcode. The address would then be incremented where our input/output specialist would pass along the next address, and so on.



**Input/Output**

Input/output for this project was fairly straightforward. Parsing text and converting to a hex number was somewhat challenging, but easy to handle. The most challenging part of constructing the IO Subroutine was molding it in a way that was compatible with the Opcodes Subroutine.

**Opcodes**

The opcode section obtains the machine code from the address register and increments it for the EA section or the next opcode accordingly. The machine code is stored in a data register to be compared with different possible op-code hexadecimals. A loop function is implemented to take out bits from the machine code for easier comparison, such that shifting out 4 bits might populate D3(used to store the loop output) with $0101 for ADDQ, the program would then jump to the opcode or a specific check section to distinguish from opcode with similar bits. The Op-code section then prints out the op-code with its appropriate size and stores the necessary information (direction, registers, EA mode, EA register) in a specific address register for the EA section to use before using JSR to go to the EA section for the specific op-code.

**EA**

The EA section reads in the parsed register, mode, and other relevant information from A5 as given by the opcode section. The EA register and mode are moved to data registers where they are then parsed to determine what the output should be.

For immediate data and MOVEM which effects where the next instruction is read from, opcode passes the address register where instructions are stored and the word or longword of the immediate data/MOVEM list are parsed for their data while the address register is incremented accordingly.

**II. Specifications**

* This program translates raw data into assembly code.
* The program is run from the EASy68k Assembly Simulator.
* It requires a valid .S68 file, opened from the EASy68k simulator.
* It accepts user input for starting and ending address, converting from a string entry to a hex number.
* Both the opcode and the EA mode are translated from the machine code.
* The program outputs the translated assembly code to the output console.
* The program prompts for an enter key press every 30 lines to ensure the user can easily see every line of code being output.
* After translating the range of entered address, the user has the option to test a new segment of memory or exit the disassembler.
* The program is contained within 4 .X68 files:  
   - IOSubroutines contains the start of the program and user input handling.  
   - OpcodeSubroutine contains the translation algorithms for opcodes.  
   - EA contains the translation algorithms for EA addresses.  
   - strings contains the collection of strings used throughout the program.

**III. Test Plan**

Given that we as a group were pressed for time, we elected to use test-driven design, testing our program with a provided test file as we made changes. We also tested how our program integrated between subroutines, such as how IOSubroutines interfaced with OpcodeSubroutine.

For our test bench, we elected to use demo\_test, a test file provided by Professor Yang Peng. This test file tests each required opcode and most EA modes.

**IV. Exception Report**

**I/O**

* One of the requirements for this project was printing the address of the line of instruction in addition to the opcode. However, due to time constraints, we were not able to implement this feature.

**Opcodes**

* All required opcodes have been implemented and prints correctly with appropriate sizes.

**EA Modes**

* Some immediate data print outs get mixed up and output incorrectly.