Team YML

Disassembler

**CSS 422, Spring 2017**

*Alvin Manalastas*

*Max Lauzon*

*Youngmin Lee*

Contents

[Program Description 2](#_Toc483900406)

[Figure 1: Program Flow 2](#_Toc483900407)

[Specification 3](#_Toc483900408)

[Test Plan 4](#_Toc483900409)

[Exception Report 4](#_Toc483900410)

[Team assignments and report 4](#_Toc483900411)

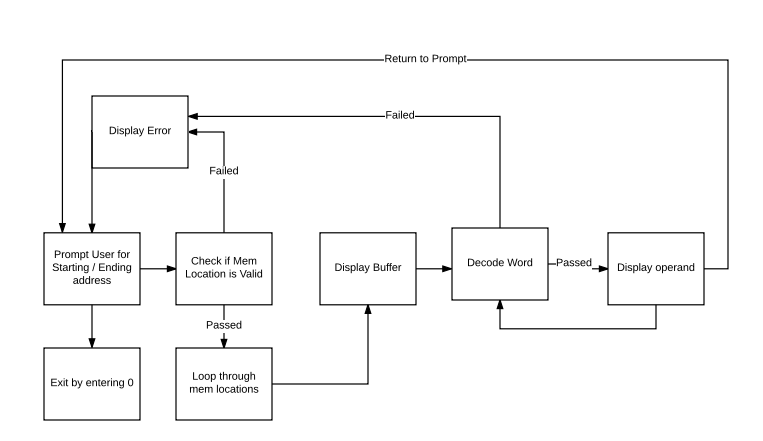
# Program Description

This project is a Disassembler for the Motorola MC68000 Microprocessor. The Disassembler scans a section of memory and attempts to convert the memory’s contents to a listing of valid assembly language instructions. This particular Disassembler is designed to run on the EASy68K Editor/Assembler. Once the user executes the program, they are greeted with a console that prompts them to input a starting and ending address in memory. Upon inputting valid addresses, a list of instructions is printed in the console. When one instruction has been decoded and printed to the console, the program loops until the ending address has been met. The flow of the program follows the diagram in Figure 1. Like most disassemblers, our Disassembler does not recreate symbolic or label information. Any unsupported opcodes are printed as DATA during disassembly.

Instructions are printed in ASCII-readable format using the following arrangement:

Memory address Op-code word

## Figure 1: Program Flow



# Specification

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Supported opcodes:**   |  | | --- | | * ADD | | * ADDA | | * ADDI | | * ADDQ | | * AND | | * ANDI | | * ASL | | * ASR | | * Bcc (BCC, BGT, BLE, BHI, BLS, BCS, BNE, BEQ, BVC, BVS, BPL, BMU, BGE, BLT) | | * BCHG | | * CLR | | * CMP | | * CMPA | | * CMPI | | * DIVU | | * EOR | | * EORI | | * JSR | | * LEA | | * LSL | | * LSR | | * MOVE | | * MOVEA | | * MOVEM | | * MOVEQ | | * MULS | | * NOP | | * OR | | * ROL | | * ROR | | * RTS | | * STOP | | * SUB | | * SUBA | | * SUBI | | * SUBQ | | **Supported effective addressing modes:**   * Absolute Long Address * Absolute Word Address * Address Register Direct * Address Register Indirect * Address Register Indirect with Post-incrementing * Address Register Indirect with Pre-decrementing * Data Register Direct * Immediate Data |

# Test Plan

Since the program uses a lot of jump tables, testing that the code was properly manipulated and that the table jumped to the correct section of the program was important. The testing of the program began with writing valid machine code and checking with a subroutine that the bits were correctly shifted. After testing that all the opcodes jumped correctly, we began writing and testing the next level of the machine code. If necessary, the program also reads additional instruction words and decodes that as well.

Once the opcodes and addressing modes have been tested on their own, we started the integrated testing. The integrated tests made sure that the opcode was printed then the correlating effective address was decoded and printed properly for that certain opcode.

As far as coding guidelines, we made sure that jump tables were used wherever we could. This made it easier to handle any errors and it also made adding any other opcodes easier since subroutines would just have to be inserted into the jump table.

# Exception Report

# Team assignments and report