

Motorola 68000 Disassembler with EASy68K

12.9.2020

CSS 422 Autumn 2020

Jayden Fullerton

Gabriel Acuna

Sean Miles

Luo Leng

Table of Contents

Table of Contents	1
Program Description	2
Design Philosophy and Flowcharts	2
Algorithms	3
EA Identification	3
Hexadecimal to ASCII	4
Specification	5
Opcodes	5
Addressing Modes	5
Valid Memory Locations	6
An Important Note Regarding Auto-Conversion	6
Test Plan	7
Exception Report	8
Team Assignments & Report	9

Program Description

Design Philosophy and Flowcharts

Our disassembler takes Motorola 68K machine code and converts it back into assembly language instructions. The general outline of our code can be found in our flowchart (**Figure 2**). We decided it would be easier to try and distinguish between the fifteen unique opcodes before verifying they were indeed the correct opcode. Essentially, we would use the flowchart in **Figure 1** to determine which opcode we *thought* the opcode was, before verifying within the “subroutine” for each opcode (we didn’t actually use a formal subroutine, but rather branching).

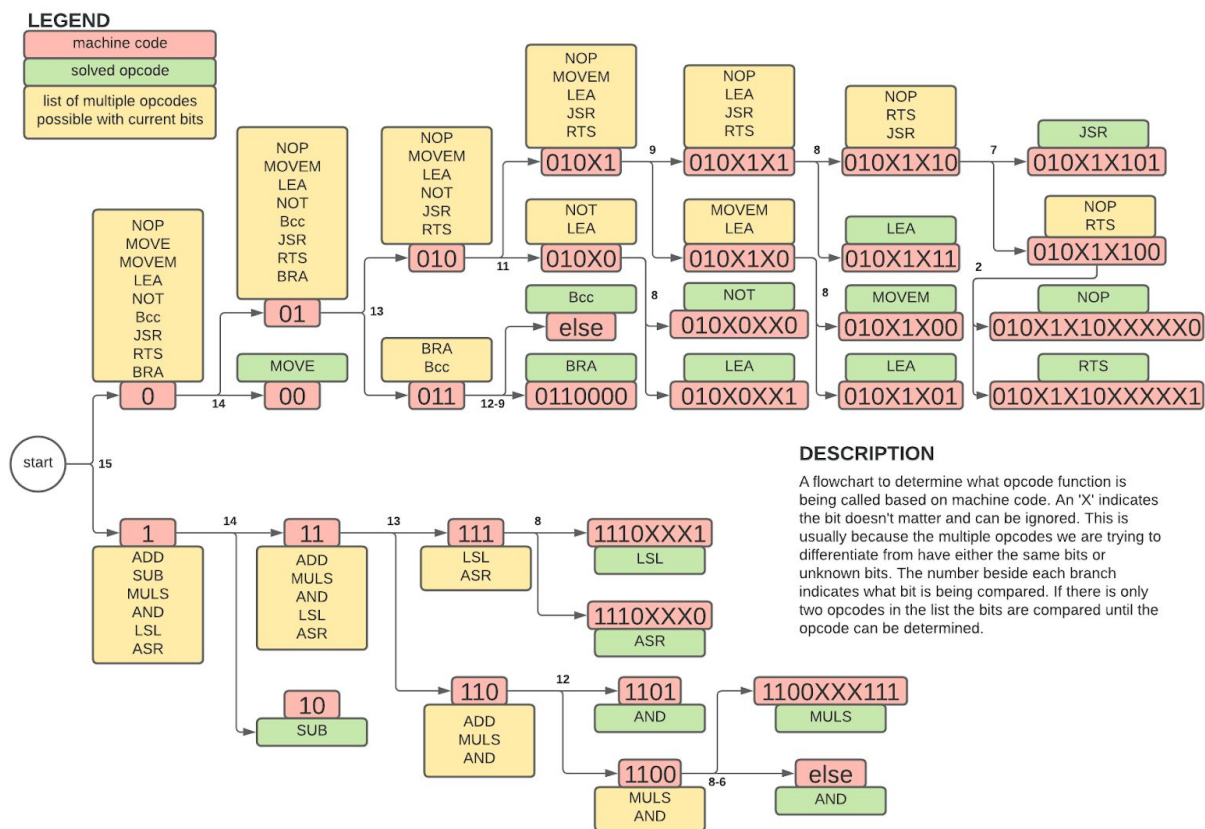



FIGURE 1: A flowchart used to identify which opcode is being used.

While it may seem counterintuitive to essentially decode twice (once to distinguish the opcode and once to ensure the bits actually resembled the opcode), it made it a



lot simpler in many ways. Since each opcode can vary widely with different operand forms, extensions, masks, etc., it is quite useful to identify which opcode we think we are dealing with as soon as possible. We can also take advantage of shifting this way and checking the carry condition flag, which is rather simple compared to having to check potentially over ten bits for every opcode.

Also, we decided to load all of the ASCII characters into memory for printing individually, and then printing them at the end of each opcode. This decision was made for many reasons, the main one being that if we discover something is invalid late into the decoding process (for example the second operand's EA is an invalid form), we can overwrite what has already been loaded for printing with DATA (code for an invalid opcode). This method also makes the code easier to read, as we don't have to constantly call trap fifteen functions to print characters to the console.

Algorithms

Besides this idea of identifying the opcode before checking it, there aren't any advanced algorithms in play here. The only "canned" routines that are used in multiple places are an EA identification subroutine, and a hexadecimal to ASCII subroutine.

EA Identification

Since the mode and register bits are consistent throughout all the opcodes, we figured it would be easier to deal with printing them in a subroutine instead of having to print them out for each opcode that used them individually. The subroutine deals with six bits (three mode bits and three register bits), which it then uses to load the corresponding opcode for printing. This subroutine is called very frequently throughout our code, and can even be used for registers not in the effective address form by adding the corresponding mode bits before calling the subroutine.

This subroutine does *not* check if the addressing mode is valid for a given opcode, because that varies depending on the opcode. Thus, each opcode is responsible for determining whether the addressing mode is valid before calling the subroutine. This is rather redundant to do for each opcode, but since every call varies we thought it would be the easiest solution.

Hexadecimal to ASCII

Since we are unable to use trap task fifteen (display unsigned number converted to base), we had to implement logic for converting a raw hexadecimal number into ASCII for printing. We decided to do one word at a time, as that seemed to be the most useful size to load for printing at one time.

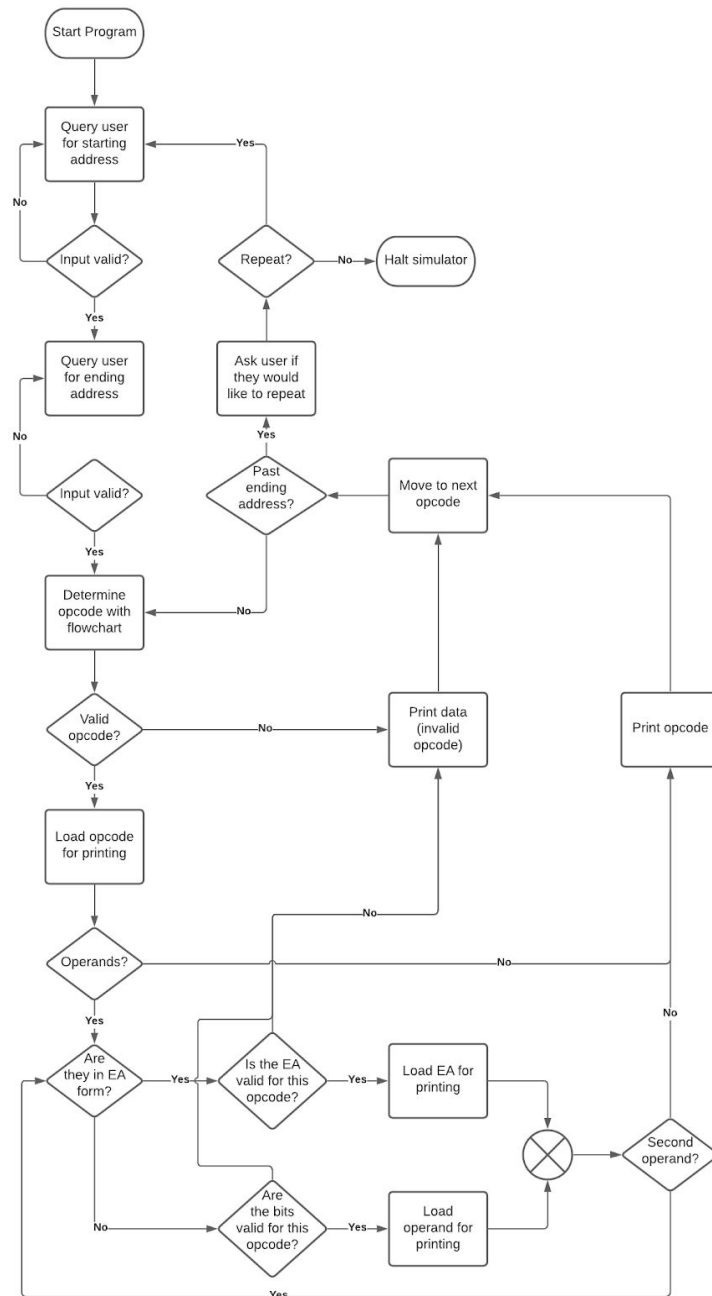


FIGURE 2: Flowchart representing general flow/logic of our program.

Specification

Opcodes

Our program takes machine code for the Motorola 68000 processor and converts it into the assembly language for Motorola 68000.

There are currently fifteen instructions supported by our disassembler listed below:

- | | | |
|-------|--------|---------|
| - NOP | - MOVE | - MOVEM |
| - ADD | - SUB | - MULS |
| - LEA | - AND | - NOT |
| - LSL | - ASR | - Bcc* |
| - JSR | - RTS | - BRA |

*The only conditions supported by Bcc are LT, GE, and EQ

*MULS.L not supported

Addressing Modes

The addressing modes that are supported by our disassembler are listed below:

- | | |
|--|---|
| - Data register direct | - Address register direct |
| - Address register indirect | - Address register indirect with post-increment |
| - Address register indirect with pre-decrement | - Absolute word addressing |
| - Absolute long addressing | - Immediate data |

The addressing modes are checked to ensure they are valid before the opcode is recognized. For example, if the destination of a MOVE instruction is an address register, the opcode will be recognized as invalid as that addressing mode is not supported by the MOVE instruction.

Valid Memory Locations

The instructions that are decoded are decoded sequentially incrementing up from a location in memory specified by the user. Valid memory locations include any address between \$00003000 (everything below is reserved for the program) and \$00FFFFFF (the last memory location).

An Important Note Regarding Auto-Conversion

When compiling 68K code, there are several auto-conversions the compiler will make that can change the opcode. For example, a call for ADD using immediate data as the source will automatically use ADDI's opcode (potentially ADDQ depending on the size of the immediate data). Since alternative opcodes like ADDI and ADDQ cannot be properly decoded by our program, we allow opcodes to support addressing modes that would normally be taken care of by auto-conversion.

For example, ADDI's opcode will return as invalid (as ADDI is not supported by our disassembler). If you want to use ADD with immediate data you can use ADD's opcode directly and use the addressing mode for immediate data. While this is not ideal, it allows us to support addressing modes intended to be supported without implementing additional opcodes.

Test Plan

Our test file can be found as **Terminal_Test.X68**.

Our program is tested by first executing the Terminal_Test.X68 file, with the Terminal_Test.S68 file being built as a result. Then, the Terminal.X68 file is executed and the test data from Terminal_Test.S68 is inserted into memory. The disassembler first prompts the user for a starting and ending address, given a list of requirements and constraints for input. Once given the addresses, the disassembler prints the first ten lines of the disassembled output, printing ten more if the user presses Enter. Once all instructions have been decompiled, the disassembler prompts the user if they would like to disassemble again, asking for a "Y/N" input.

The test file contains a list of opcodes and their respective operands, with no invalid instructions being included in the test file, as the file would not be able to execute and build the .S68 file for testing. However, if the instructions were manually entered into memory, our code has various checks for confirming the validity of the instructions, and should be able to recognize when an instruction's bit format is not valid for the opcodes supported by our disassembler. If an opcode is supported by the disassembler, the output should be identical to that of the testing file, omitting comments and label names. If an opcode is not supported, the disassembler outputs "DATA" and the corresponding hex data related to the instruction, as required.

In terms of coding standards, our team is aiming for readability over unnecessary complexity, implementing subroutines for common operations, and reusing code where possible. Many of the opcodes differ in miniscule ways or include various exceptions in relation to one another, and as such the code we implemented was done so on a case-by-case basis for the majority of the opcodes. However, some opcodes are very similar in process, such as LSL and ASR, and ADD and SUB, and as such the team reused portions of code where possible. The code is not as efficient as it possibly could be, however, this was not important to our team in developing the disassembler, as we valued completion and readability above all else.

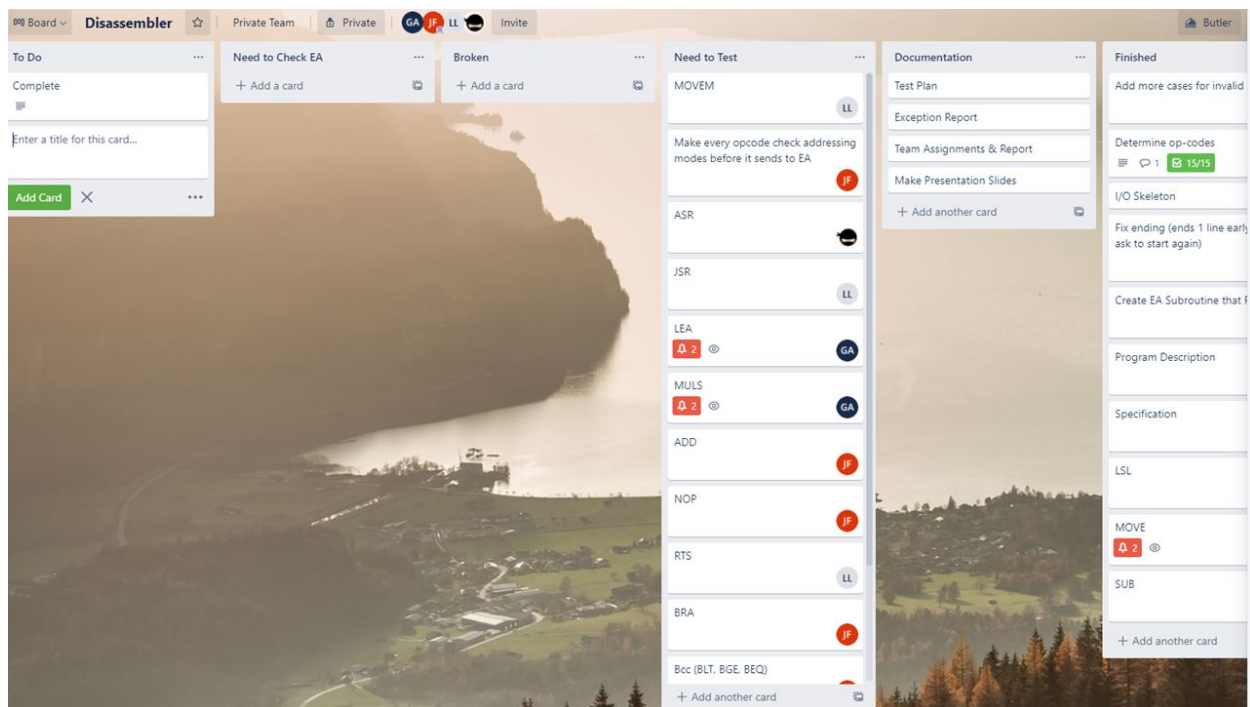
Exception Report

There is nothing in our program that deviates from what is expected. All opcodes behave as we intend, and all bugs we know of have been resolved.

Team Assignments & Report

Programmer	Task
Jayden	NOP
Gabriel	MOVE
Jayden + Luo	MOVEM
Jayden	ADD
Jayden	SUB
Gabriel	MULS
Gabriel	LEA
Sean	AND
Sean	NOT
Sean	LSL
Sean	ASR
Jayden	Bcc (BLT, BGE, BEQ)
Luo	JSR
Luo	RTS
Jayden + Gabriel	BRA
Jayden	EA Decoding
Jayden + Gabriel + Sean + Luo	EA Validity Checks
Jayden + Luo	I/O
Sean	Testing
Jayden + Sean	Documentation
Gabriel	Presentation Slides

Our Trello Board (before all was finished):



We believe all team members contributed equally in the project. Not all opcodes are created equal, and in the end we all ended up contributing the same amount of effort to the project.