Disassembler M68K Project Documentation

By The Cool Group

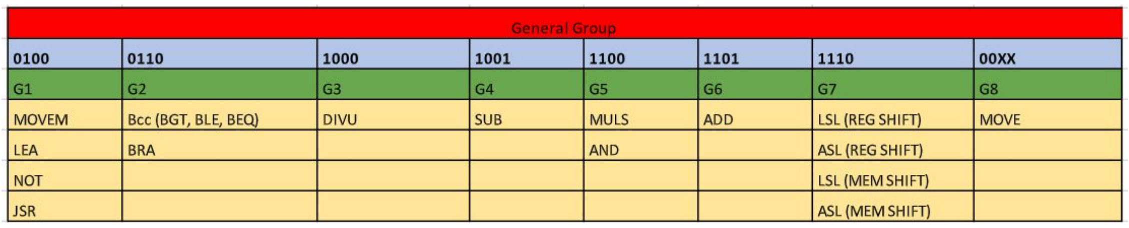
[Micah Rice, Zachary Liong, Sesario Imanputra, Marci Ma]

1.) Program description

When we first started this project, we created an excel sheet of all the various requirements for our project. We use both the quick reference of OPCODES, as well as the programmer's manual to watch what values are important and location of each bit.

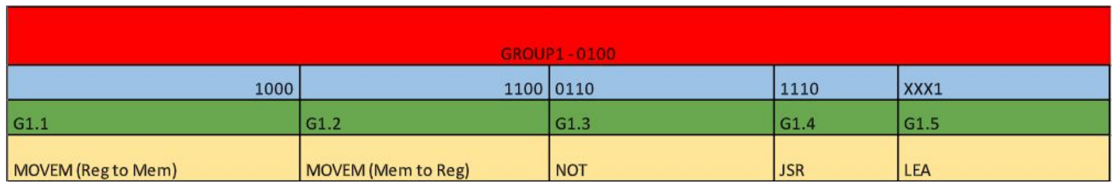
For example, we would first find the first four binary bits of the OPCODE that were required, and generalize them into groups. We would also check the next four bits to categorize them to specific OPCODES. See Figure 1.1 of the most generalized groups, and Figure 1.2 for an example of broken down groups of OPCODES.

Figure 1.1 - OPCODE Generalized Groups



General Group							
0100	**0110**	**1000**	**1001**	**1100**	**1101**	**1110**	**00XX**
G1	G2	G3	G4	G5	G6	G7	G8
MOVEM	Bcc (BGT, BLE, BEQ)	DIVU	SUB	MULS	ADD	LSL (REG SHIFT)	MOVE
LEA	BRA			AND		ASL (REG SHIFT)	
NOT						LSL (MEM SHIFT)	
JSR						ASL (MEM SHIFT)	

Figure 1.2 - OPCODE Specific Categorized Group

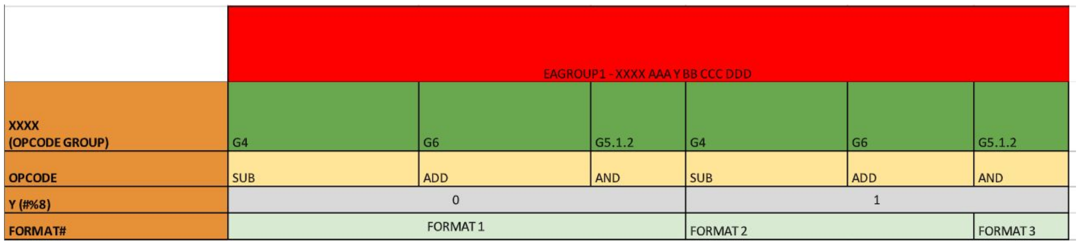


GROUP1 - 0100				
1000	1100	0110	1110	XXX1
G1.1	G1.2	G1.3	G1.4	G1.5
MOVEM (Reg to Mem)	MOVEM (Mem to Reg)	NOT	JSR	LEA

There are some exceptions to the rules, like RTS and NOP. These instructions take no operands and have unique hexadecimal values, 4E75 and 4E71 respectively.

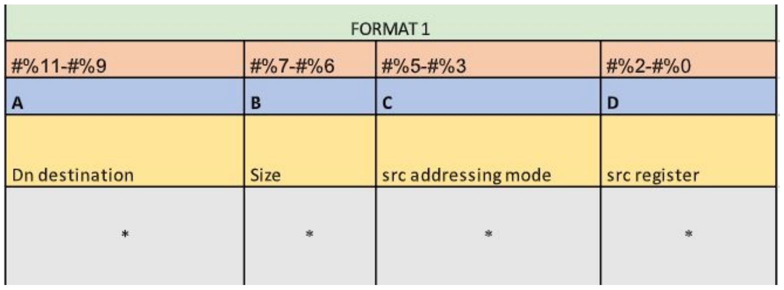
We would also make an excel table for EAGROUPS, generalizing them based on their bits, and the binary bits locations (called formats) to look out for. It is worth mentioning that in each group, there would be the appropriate OPCODES that the EA would use. See Figure 2.1 and 2.2 for an example of EAGROUP and format.

Figure 2.1 - EAGROUP 1



	EAGROUP1 - XXXX AAA Y BB CCC DDD					
XXXX (OPCODE GROUP)	G4	G6	G5.1.2	G4	G6	G5.1.2
OPCODE	SUB	ADD	AND	SUB	ADD	AND
Y (#%8)	0			1		
FORMAT#	FORMAT 1			FORMAT 2		FORMAT 3

Figure 2.2 - EAGROUP 1's format



FORMAT 1			
#%11-#%9	#%7-#%6	#%5-#%3	#%2-#%0
A	B	C	D
Dn destination	Size	src addressing mode	src register
*	*	*	*

Then, we created a flowchart that would give us an understanding of how the disassembler would try to break down the instructions and output it.

In the main disassembler program, we ask the user to input the values of the start and end memory address that the user wants to disassemble, while validating the addresses. Then, we would break down the OPCODE instruction into hexadecimal values and store it.

Then the next stage would be the OPCODE disassembly, where we would break the hexadecimal into binary and have our own version of the jump table to appropriately go to the branches that the instruction was referring to. Also, near the end of each group OPCODE stage, we have a string buffer store the name of the instruction, as well as call the EAGROUP to store them into the buffer. The order stored into the buffer is destination, source, instruction, and the PC code.It calls both the output addresses and EAGROUP stages.

Next, we go onto the last stage: EAGROUP. We call the last stage of OPCODE to ask for the binary bits and take the nth bits we need to figure which registers are source and destination. Of course, there are some exceptions to this generalized group like ASL Reg Shift, where the 5th bit is I/R that would identify if the source is register or immediate data.

Finally, we took inspirations from the addendum in canvas and some disassemblers from GitHub (located at end of description, Figure 3), which led to the use of the string buffer. Originally, we plan to display opcodes using trap task 13. However, we found this insufficient as invalid cases will often disrupt the flow of the output (i.e. displaying an opcode with incorrect EA). In terms of handling opcodes, the addendum recommended the use of a jump table. However, since we were not confident to implement it ourselves, we made a comparison table instead. The comparison table behaves similarly to a jump table with the exclusive use of the comparison opcode rather than the jump to subroutine opcode. We favored this implementation as we completely understood how the program behaves, which helps with debugging during the later stages of development.

Github Reference: https://github.com/himanshumehru/68K-Disassembler/blob/master/TeamBits_Disassembler.X68

Also, there were several known limitations that were encountered during our program writing. SUBX and ADDX were a problem, since our program was having difficulties differentiating from SUB and ADD respectively. Another limitation is when handling symbols with immediate values. Thus, all our immediate values are always displayed as the hex equivalent. This is because there is no difference in the machine language for immediate values given a hex, binary, or decimal value. The last limitation of our program is our choice to display extra zeroes for a given value that does not exactly take up all the size of the given opcodes that use immediate values for their effective addressing. For example, #3 will be displayed as #$03 for byte size, #$003 for words, etc. We chose to do this so that all values using immediate values will be indistinguishable depending on their size.

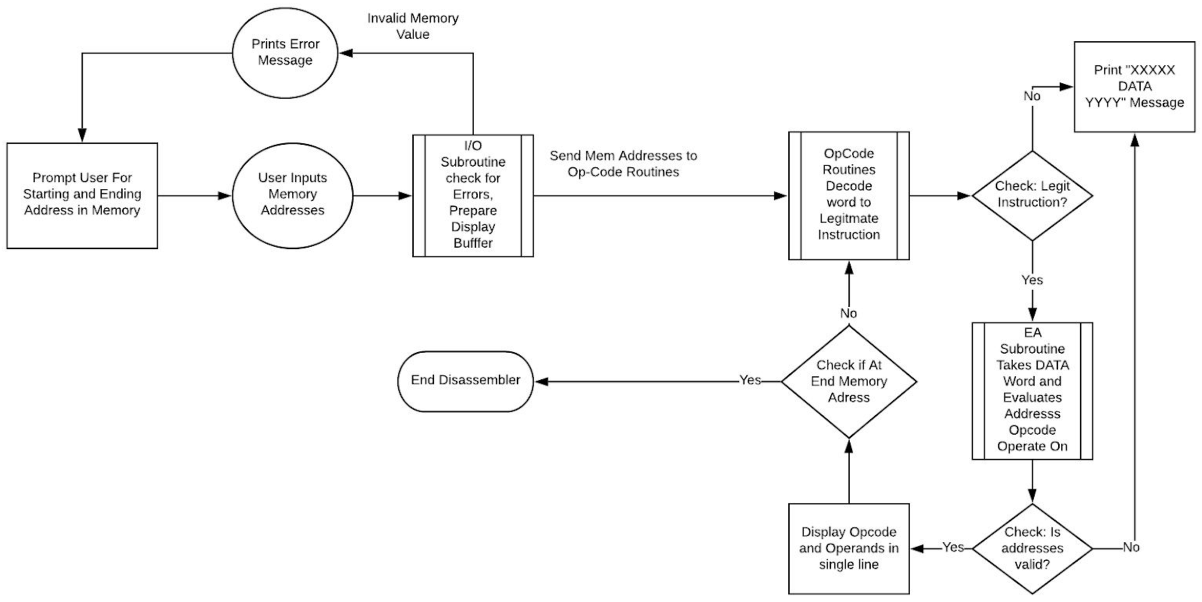

Invalid Memory Value
Prints Error Message
Prompt User For Starting and Ending Address in Memory
User Inputs Memory Addresses
I/O Subroutine check for Errors, Prepare Display Buffer
Send Mem Addresses to Op-Code Routines
OpCode Routines Decode word to Legitmate Instruction
Check: Legit Instruction?
No
Print "XXXXX DATA YYYY" Message
Yes
EA Subroutine Takes DATA Word and Evaluates Address Opcode Operate On
Check: Is addresses valid?
No
Yes
Display Opcode and Operands in single line
Check if At End Memory Adress
No
Yes
End Disassembler

Figure 3 - Simplified version of Program Flowchart. Click the link for a more detailed version: [https://app.lucidchart.com/documents/edit/7bac2a00-adfb-4e82-b551-91b024d4c414/0_0?shared=true]

2) Specification

1. The program starts with a display message.
2. The program asks the user for a starting address and an ending address.
3. The program checks if all the input addresses are valid or not.
4. If the address is not valid, ask for the address again.

5. Also check if we reach the end of the address. If not, continue, if so, end of the program.
6. The program jumps to the OPCODE table and starts the conversion.
7. Save the OPCODE to the string buffer and call the corresponding EAGROUP.
8. The program jumps to the EAGROUP and decodes the addressing mode.
9. I/O print out all the OPCODE and EAGROUP in order.
10. Program asks user if they wish to evaluate a new memory address, if not:
11. Program ends with no crashes.

Supported Opcodes and EA

Required Opcodes	Required Effective Addressing Modes
NOP MOVE MOVEM ADD SUB MULS DIVU LEA AND NOT LSL ASL Bcc (BGT, BLE, BEQ) JSR RTS BRA	Data Register Direct Dn Address Register Direct An Address Register Indirect (An) Immediate Data #<data> Address Register Indirect with Post incrementing (An)+ Address Register Indirect with Pre decrementing -(An) Absolute Long Address Abs.W Absolute Word Address Abs.L

	Size	EA
NOP	No Size	No EA
MOVE	All(Byte, Word, Long)	Destination: Dn, (An), (An)+, -(An), Abs.W, Abs.L Source: All
MOVEM (reg to mem)	Word, Long	(An), -(An), Abs.W, Abs.L
MOVEM (mem to reg)	Word, Long	(An), (An)+, Abs.W, Abs.L

ADD	All(Byte, Word, Long)	Destination: (An), (An)+, -(An), Abs.W, Abs.L Source: All
SUB	All(Byte, Word, Long)	Destination: (An), (An)+, -(An), Abs.W, Abs.L Source: All
MULS	Word	Dn, (An), #<data>, (An)+, -(An), Abs.W, Abs.L
DIVU	Word	Dn, (An), #<data>, (An)+, -(An), Abs.W, Abs.L
LEA	No Size	(An), Abs.W, Abs.L
AND	All(Byte, Word, Long)	Destination: (An), (An)+, -(An), Abs.W, Abs.L Source: Dn, (An), #<data>, (An)+, -(An), Abs.W, Abs.L
NOT	All(Byte, Word, Long)	Dn, (An), (An)+, -(An), Abs.W, Abs.L
LSL	All(Byte, Word, Long)	(An), (An)+, -(An), Abs.W, Abs.L
ASL	All(Byte, Word, Long)	(An), (An)+, -(An), Abs.W, Abs.L
Bcc	Byte, Word	No EA
JSR	No Size	(An), Abs.W, Abs.L
RTS	No Size	No EA
BRA	Byte, Word	All

3) **Test Plan**:

Test Plan:

We tested our program by filling out a massive list of instructions organized at $1000 in memory, and ran our program from $4000 examining each of those instructions. We examined in-scope and out of scope instructions to make sure they were handled correctly, either outputting invalid or the correct opcodes & operands through visual inspection for each line.

Test File:

Below, is our latest test file for our program, although we altered this at various times during development.

```
*-----------------------------------------------------------
* Title      :  Test file
* Written by :  Sesario Imanputra, Zachary Liong, Micha Rice, Marci Ma
* Date       :  8/17/2020
* Description: Test file
*-----------------------------------------------------------
       ORG    $4000
****************************************************
START:
pro
             NOP

             ;OPCODEGROUP8 TEST
             MOVE.B  D2,D3
             ;MOVEA #$00000047,A0 ;should be data

             ;OPCODEGROUP1.1 TEST
             MOVEM.W D0,-(A5)
             MOVEM.W D1,-(A5)
             MOVEM.W D2,-(A5)
             MOVEM.W D7,-(A5)

             MOVEM.W A0,-(A5)
             MOVEM.W A1,-(A5)
             MOVEM.W A2,-(A5)
             MOVEM.W A7,-(A5)
             MOVEM.W D0-D7/A0-A7,-(A5)
             MOVEM.W D1-D4/A2-A5,-(A5)

             MOVEM.W (A5)+,D0
             MOVEM.W (A5)+,D1
             MOVEM.W (A5)+,D2
             MOVEM.W (A5)+,D7
             MOVEM.W (A5)+,A0
             MOVEM.W (A5)+,A1
             MOVEM.W (A5)+,A2
             MOVEM.W (A5)+,A7
             MOVEM.W (A5)+,D0-D7/A0-A7 ;D7 not printing
             MOVEM.W (A5)+,D1-D4/A2-A5

             ;JMP   pa     ;should be data (not working)

             ;OPCODEGROUP6 TEST
             ADD.B   D2,D3
             ADD.B   D0,$00001012
             ;ADDX   -(A2),-(A3) ;should be data (not working)
             ;ADDA   D3,A3       ;should be data

             ;OPCODEGROUP4 TEST
             SUB.B  D2,D3
             SUB.B   D0,$00001012
             ;SUBX   -(A2),-(A3) ;should be data (not working)
             ;SUBA   D3,A3       ;should be data
             ;OPCODEGROUP5 TEST
             MULS  D2,D3
             ;MULU  D2,D3        ;should be data (not working)
             ;OPCODEGROUP3 TEST
             DIVU  D2,D3
             ;DIVS  D2,D3        ;should be data
             ;OPCODEGROUP1.5 TEST
             LEA   $2000,A2
             ;OPCODEGROUP5 TEST
             AND.B   #3,D3
             AND.B   D0,$00001012
             ;EXG   D2,D3
             ;OPCODEGROUP1.3 TEST
             NOT.B   D3
             NOT.W   (A3)+
             NOT.L   -(A3)
             NOT.W   $1234
             NOT.L   $12345678
             ;MOVE  D3,SR
             ;OPCODEGROUP7.1.1 TEST  REG SHIFT
             ;LSL.B   #2,D3
             LSL.L   D0,D5
             ;OPCODEGROUP7.1.2 TEST   REG SHIFT
             ASL.B   #2,D3
             ;OPCODEGROUP7.3.2 TEST     MEM SHIFT
             LSL   (A0)+
             ;OPCODEGROUP7.3.1 TEST     MEM SHIFT
             ASL   (A0)+
             ;OPCODEGROUP2.3 TEST
             BGT   pa
pa           ;OPCODEGROUP2.4 TEST
             BLE   pi
pi           ;OPCODEGROUP2.2 TEST
             BEQ   pu
pu           ;OPCODEGROUP2.1 TEST
             BRA.W  po
po            RTS
pe           ;OPCODEGROUP1.4 TEST
             JSR   (A1)
              *taken from absolute value testing given in canvas
              MOVEA.W #$9100, A0 *A0<-- FFFF9100
              MOVEA.W #$1000, A1
              LEA   $9100, A0  *A0<-- 00009100
             MOVEA.W #$4213, A0 *A0<-- 00004213
              MOVEA.L #$9100, A0 *A0<--- 00009100
              MOVEA.L #$00004213, A0 *A0<-- 00004213
              MOVE.L #$00008000, D0
              MOVEA.W D0, A0

                   SIMHALT             ; halt simulator
       END    START        ; last line of source
```

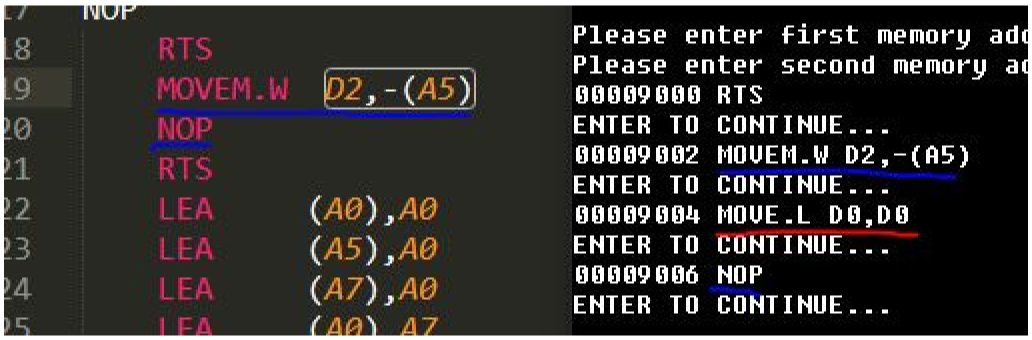
Coding Standards:

In our group we strove to write code that is commented at each significant point of divergence usually above the line which was significant, and especially we commented code that was not currently working, for this usually on the same line as the code, or for explanation. We used a ';' to comment typically. Other than our commenting conventions, we could not deviate much from the format restrictions of M68k Assembly programming. Our opcodes in our disassembler typically start indented 16 spaces from the left, with the operands usually organized 24 spaces from the left.

We separated our code into different chunks by the branch labels, and identified issues to each other by the name of the label, or the specific line of the code if we were working together on a call with shared screens, since some of us have editors which display lines.

4) **Exception report**:

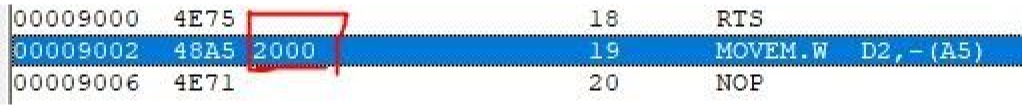
We were able to accomplish all the goals we set out to do at the beginning of our project. However, there is one unexpected deviation from what is standard.



```
    NOP
        RTS
        MOVEM.W   D2,-(A5)
        NOP
        RTS
        LEA       (A0),A0
        LEA       (A5),A0
        LEA       (A7),A0
        LEA       (A0),A7
```

```
Please enter first memory ad
Please enter second memory a
00009000 RTS
ENTER TO CONTINUE...
00009002 MOVEM.W D2,-(A5)
ENTER TO CONTINUE...
00009004 MOVE.L D0,D0
ENTER TO CONTINUE...
00009006 NOP
ENTER TO CONTINUE...
```

Fig. 4 - (left) Test file opcodes, and (right) the disassembler output



```
00009000   4E75                          18        RTS
00009002   48A5 2000                     19        MOVEM.W   D2,-(A5)
00009006   4E71                          20        NOP
```

Fig.5 The hex word in memory being decoded as MOVE.L D0,D0 is outlined in red

In our own Test file, MOVEM.W D2, -(A5) generates an unexpected interpretation of an opcode action that does nothing: MOVE.L D0,D0. This is because the next hex word after the last valid instruction is identical to the unexpected instruction in memory.
Our disassembler does correctly decode the MOVEM.W D2,-(A5) instruction, and this additional throwaway action does not perform an actual task, nor does it cause any error with subsequent instructions being examined in memory.

Other than this, we have identified no particular exceptions in our program. All in-scope operands are decoded correctly as far as we were able to test both using our testing file and the provided one. We also identified out-of-scope operands and EA as invalid correctly, per our testing. We expect our final program to run without crashes, and put out all the correct data, except in the very particular situation described above where an extra “throwaway” opcode might be decoded in between two correctly evaluated instructions.

5) **Team assignments and report**:

Our original Team Organization plan was to approach the work not by dividing the different functions of the code, but to approach each problem sequentially as a group. Over the course of our project work, we had to make some adjustments due to emerging circumstances or the nature of the operation. The most major adjustment was losing one of our group members after the midterm, and gaining a new member. As this new member joined during the development process, we used one meeting as a team to onboard and create a github for Marci Ma, our new member and familiarize her with the work we had done so far, and what remained.

As the quarter developed, a work balance did seem to develop in our team. About 50% of the work of coding our project can be attributed to the herculean efforts of Sesario Imanputra, who was able to take on extra effort to alleviate the workload as Sesario is taking one class and a capstone this quarter.The other group members, Marci Ma, Zach Liong, and Micah Rice were registered for three classes this quarter, to say nothing of the consistent and continuing background effects of the global pandemic which at various times impacted all group members. This enabled each group member to contribute to the project according to their level of availability.

Percent Coding Contribution:
At the end of the day, Sesario contributed ~40% of the total coding for Disassembler Project, and was voted MVP by the rest of the group. For the rest of the group, the contribution was about even, making the breakdown Marci Ma ~20%, Micah Rice ~20% and Zach Liong ~20%

We used GitHub to organize our work, with a master branch, and one branch for each student. Our development model was Test Driven Development, so each time we finished a branch of our code we tested it by updating our test file to include the new opcodes or EAs as the case may be.

Including both elements of the coding and the elements not directly related to coding, much of the work was done on synchronous discord calls or through discord chat asynchronously.
We created and communicated extensively on a group discord server, with separate #channels to communicate about different topics or to share files or resources. We worked together on group calls each Monday and Friday for several hours at a time and shared our screen to go over the code together, or determine the logical processes of our project, and to compile reports.