CSS422 – Disassembler

Final Project  
By: Bartosz Dabkowski and Bethel Heo

Table of Contents

[Project Description 3](#_Toc421539134)

[Preliminary 3](#_Toc421539135)

[Getting Addresses 3](#_Toc421539136)

[Output 3](#_Toc421539137)

[Instructions 4](#_Toc421539138)

[Project Specifications 5](#_Toc421539139)

[Manual 5](#_Toc421539140)

[What program does 5](#_Toc421539141)

[How to use it 5](#_Toc421539142)

[Code Standards 6](#_Toc421539143)

[Test Plan 7](#_Toc421539144)

[Initial Testing 7](#_Toc421539145)

[Full testing 7](#_Toc421539146)

[Issues 8](#_Toc421539147)

[Exceptions Report 8](#_Toc421539148)

[MOVEM 8](#_Toc421539149)

[MULS.L 8](#_Toc421539150)

[Dec vs Hex 8](#_Toc421539151)

[Task Breakdown Report 9](#_Toc421539152)

# Project Description

## Preliminary

Starting the disassembler was the most difficult part of the entire project. In order to get a good start we decided to question outside sources to get a general idea of what the structure should look like. We asked a few sources ranging from CSS tutors to students that have already completed the course to get a feel of the overall structure of the project, things that were difficult, best practices, and things to watch out for. We created an initial flow chart that would be constantly updated and decided to pair program in the beginning to make sure we were both comfortable with the overall flow (flowchart is bundled in this zip). Once we were both confident in the framework we decided to split the project by instructions.

## Getting Addresses

The first thing we decided to implement was the input for the starting and ending address and converting them to valid addresses. We did this by taking in an input and storing it as a string in an address register. We would then go through each bit and determine if it was a valid hex valid. Next, we changed the values in the address register to hexadecimal by adding 30 if it was a number or 37 if it was a letter. Finally, we moved the hex value from the address register to a data register and shifted left by one to get the hex address in one data register. We did this 8 times and then assigned the value of the data register into A5, our designated starting address location. A5 would store our starting address and A6 would store our ending address. These would be constants throughout the whole program and the ending address would not be touched. The starting address would increment as we read in instructions to the next instruction we would need to work on until it passed the ending address in which point the program would stop.

Since this was the first thing we implemented there were some rough spots. We implemented this before we figured out bitmasking and so there was a lot of shifting left and right of data in order to isolate it to compare. Ex:

CONVERT\_DEC\_TO\_ASCII

MOVE.L D2,D1 moves D2 into D1

LSR.L #8,D1 isolate MSD

LSR.L #8,D1

LSR.L #8,D1

JSR NUMBER\_OR\_LETTER check if letter or number

It carried over to outputting the address as well before we figured out a better, more optimal way to isolate certain bits.

## Output

The format for the output is as follows:  
Memory location OPCode Operand  
Ex: 00010000 MOVE.L D0,D1

We initialized a memory location for our buffer stack that would print the output and stored it in A3. Anytime we had valid data we needed to output we would add it character by character into the buffer and then print it once that instruction was done. We would print each line of data before starting the buffer over. Ex:

MOVEL

MOVE.B #'M',(A3)+ add MOVE.L to buffer

MOVE.B #'O',(A3)+

MOVE.B #'V',(A3)+

MOVE.B #'E',(A3)+

MOVE.B #'.',(A3)+

MOVE.B #'L',(A3)+

MOVE.B #' ',(A3)+

MOVE.B #' ',(A3)+

After the line of data was done we would call a function PRINT\_VALID\_BUFFER that would output that line of data by printing each character in the stack.

Outputting the address required us to convert the HEX values into ASCII. As stated before we were able to accomplish this but used shifting to isolate the HEX digits instead of bitmasking and ended up with code like this:

JSR ISOLATE\_FIRST\_DIGIT isolate and buffer MSD digit

JSR CONVERT\_AND\_BUFFER  
 …

ISOLATE\_FIRST\_DIGIT

LSR.L #8,D2

LSR.L #8,D2

LSR.L #8,D2

LSR.L #4,D2

RTS

Luckily after talking with a few students in the class we were able to learn about bitmasking and so we implemented that strategy for the instructions.

## Instructions

After fully finishing a single instruction we had our framework and split the instructions evenly.   
To disassemble the instructions we first checked the OP codes to see which potential operation it could be. It would then branch to the specific operation and then from there figure out the operand. Once all of that had been buffered it would print it and then move on to the next instruction by incrementing the starting address and branching to START\_DISASSEMBLER. This time we were able to isolate certain bits and branch down the list by OP Code, instruction, and operand by using bitmasking. Ex:

OPCODE\_0000\_TEST

MOVE.W #OP\_0000\_BITMASK,D5 load 0000 bitmask to D5

EOR.W D2,D5 XOR instruction to D5

AND.W #OP\_AND\_BITMASK,D5 AND 1111 to XOR'd OP

CMP.W D5,D6 check if all 0s

BEQ OPCODE\_0000 if all 0s then 0000 op code

# Project Specifications

## Manual

### What program does

What this disassembler does is read the hex representation of instructions and data and convert it to readable 68k assembly language. It will take in an input from the user for the starting address and ending address of the instructions that they want disassembled in memory. It will then display the disassembled instructions 25 lines at a time, asking the user to continue after each 25 lines, until all of the instructions have been converted and displayed. Once all the instructions have been displayed the program will end.

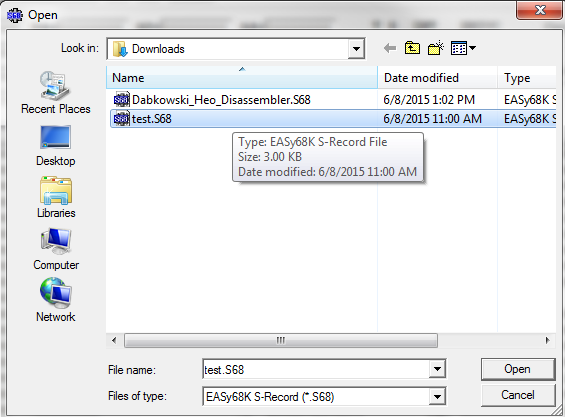
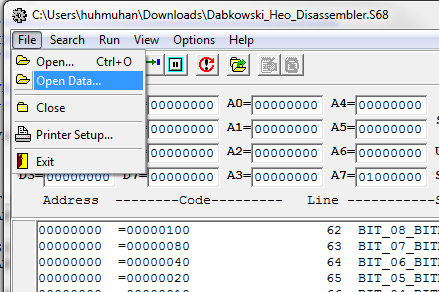
These are the supported instructions:  
MOVE, MOVEA, MOVEM, ADD, ADDA, SUB, SUBQ, MULS, DIVS, LEA, OR, ORI, NEG, EOR, LSR, LSL, ASR, ASL, ROL, ROR, BCLR, CMP, CMPI, BCC (BCS, BGE, BLT, BVC), BRA, JSR, RTS

And these are the supported effective addressing modes:  
Dn, An, (An), (An)+, -(An), #, (xxx).L, (xxx).W

Anytime the program finds an operation that is not supported it will call a function INVALID\_DATA that prints a message saying it is invalid and the invalid HEX operation.

### How to use it

Open the file, DabkowskI\_Heo\_Disassembler.X68, in EASy68k. Assemble the code by hitting F9. Click ‘Execute’ on the next pop up message. In the next window click File > Open Data. This is where the user will load the code they want to disassemble. Included with this documentation is a file called text.S68. The user can use this file to test the disassembler or use your own .S68 file. If the user uses their own test file it must be located outside of memory location $1000 - $262C, as that is where the disassembler code is. Double click this file to load it into memory.



Once the test file is loaded press F9 again to launch the disassembler. The user will be greeted with a welcome message and to input the starting address and ending address of their test code. Once the starting address and ending address are entered the disassembler will begin to display the converted instructions. If the test code is larger than 25 lines then it will prompt the user to ‘press enter to continue’. This will happen every 25 lines. Once all of the instructions have been converted and displayed the user will receive an ending message and the program will stop.

## Code Standards

* All instructions, data registers, address registers, and functions are in uppercase.
* Each function has a header with the title of the function in uppercase that clearly distinguishes it from the rest of the code.
* Comments are in lowercase.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EOR \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EOR

MOVE.B #'E',(A3)+ add EOR to buffer

MOVE.B #'O',(A3)+

MOVE.B #'R',(A3)+

* The main functions that structure the code must have pre and post conditions that describe what each data register and address register are used for and what happens after it runs.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* BUFFER\_ADDRESS \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* PRE: A3 VALID BUFFER

\* A5 CURRENT ADDRESS

\*

\* D2 ADDRESS TO MINIPULATE

\* POST: Prints the current address and then increments by 1

\*------------------------------------------------------------------

* Double tab for the start of the instruction, no spaces between the operands.
* EQUs at the top of the program, and all DCs at the bottom.
* Overall flow/structure of the code (flowchart is bundled with zip):
  + Get starting and ending addresses
  + Start disassembler
  + Print memory location
  + Read in instruction
  + Decode and buffer OP code
  + Decode and buffer operand
  + Print buffer to display
  + Loop back to start / Check if ending address
  + If ending address end program

# Test Plan

## Initial Testing

The first thing we tested was getting our starting and ending addresses from the user correctly inputted into the address registers. We tested this by running the code, inputting addresses, checking to see what ended up in memory for A5 (starting address), and A6 (ending address), and making sure they matched up with what we initially entered. When there was incorrect information we would first do an eye test and walk through the code in our heads to figure out any holes and problems. If that didn’t work then we would walk through the program using the step over and trace into options to go line by line, watching each register and memory location that is affected. We would watch the changes and compare the expectations to reality to figure out at which step in the instruction is causing the error. This proved to be an effective debugging method that we would use throughout the rest of the project.

At first, for the instructions we lightly tested the functions that we individually worked on; later on, a majority of our testing took place after we finished coding all of the instructions. In the beginning we would test the instruction by writing some instructions at the bottom of the code past our disassembler that would never run. We would put in instructions that we were currently trying to convert and use the address locations in our disassembler that those instructions were in. In this stage of testing we would use just a few options to make sure the OP codes were working and a few of the operands. In most cases if the operands worked for one, then it would work for the others.

## Full testing

Once all the operations were decoded we spent a significant amount of time testing the instructions for our disassembler. We did this by building a test file that tests all of the instructions thoroughly and we were able to see what needed to be corrected. Initially we had every possible combination of valid data registers, address registers, immediate data, and absolute addressing allowed for the instruction. Ex:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EOR \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EOR.B D0,D1

EOR.B D2,(A0)

EOR.B D3,(A1)+

EOR.B D4,-(A2)

EOR.B D5,$1000

EOR.B D6,$10000000

EOR.W D0,D1

EOR.W D2,(A0)

EOR.W D3,(A1)+

EOR.W D4,-(A2)

EOR.W D5,$1000

EOR.W D6,$10000000

EOR.L D0,D1

EOR.L D2,(A0)

EOR.L D3,(A1)+

EOR.L D4,-(A2)

EOR.L D5,$1000

EOR.L D6,$10000000

We loaded this test file into our disassembler and ran it. From there we would find the bugs and work on fixing it together.

## Issues

We found out however, that Easy68K explicitly changes the instruction, when it’s typed in incorrectly. To clarify, when “ADD.B #1, D2” is entered, Easy68K changes the instruction to “ADDI.B #1,D2”. Moreover, there are many other examples of this happened with other instructions. These changes ended up confusing us and made us think that we had incorrectly programmed some of our functions, when in fact, we had not.

In order to test the variability in every single instruction we created a test file that had every instruction that we needed to support. We ended up creating 500+ lines of code that thoroughly tested the disassembler. We would go through 25 lines at a time comparing the output of our program to that of the test file. If something didn’t match up we knew that we had to go back and debug our code. This technique was very effective at finding problems with our code and correcting them. Moreover, since we repeated the use of most of our functions we knew exactly where the problem was. The test file is included with this documentation.

# Exceptions Report

## MOVEM

We were not able to fully implement MOVEM. We couldn’t figure out the algorithm to decode MOVEM correctly and so we left the work that was accomplished and just commented it out.

## MULS.L

MULS.L is not supported as we are working on a 68k disassembler and MULS.L applies to MC68020 and above.

## Dec vs Hex

As a design decision we decided to print the decimal values instead of hex when dealing with immediate data and absolute addressing. This does not go against the project specifications but we felt we should include this in our documentation. Branching also correctly displays the address location to where the system will be branching instead of just the displacement and this address is also displayed in decimal.

# Task Breakdown Report

|  |  |  |  |
| --- | --- | --- | --- |
| Task | Implementer | Time Required | {IO, IS, EA} |
| Project Schedule | Both | 1 hour |  |
| TeamProgress Reports | Both | 3 hours |  |
| Flow Chart initial | Bethel | 45 mins |  |
| User Input | Both | 2 hours | IO |
| Starting/Ending addresses | Both | 6 hours | IO |
| Print address | Both | 3 Hours | IO, IS, EA |
| ASCII conversion | Both | 2 Hours | IO, IS, EA |
| MOVEB | Both | 5 Hours | IO, IS, EA |
| JSR, RTS, LEA, BRA | Bethel | 4 hours | IS, EA |
| ROd\_REG, ASd\_REG, LSd\_REG, ADD | Bartosz | 4 hours | IS, EA |
| SUB, SUBQ, ADDA,CMP, EOR | Bartosz | 4 hours | IS, EA |
| NEG, BCC, DIVS, OR, MULS, MOVEA.L, MOVE.L | Bethel | 6 hours | IS, EA |
| ROd\_MEM, ASd\_MEM, LSd\_MEM | Bartosz | 2 hours | IS, EA |
| MOVEM | Bethel | 3 hours | IS, EA |
| MOVEA.W, MOVE.W | Bartosz | 2 hours | IS, EA |
| CMPI | Bethel | 1 hour | IS, EA |
| ORI, BCLR | Bartosz | 1 hours | IS, EA |
| Testing | Both | 8 hours |  |
| Flow chart final | Bartosz | 2 hours |  |
| Documentation | Both | 10 hours |  |

|  |  |  |
| --- | --- | --- |
| Contributor | Total Hours | Percentage |
| Bartosz Dabkowski | 54.75 | 50% |
| Bethel Heo | 54.75 | 50% |
|  | Total Man Hours: 109.5 |  |