CSS 422 - Disassembler Final Report

By Jeremy Tandjung, Angie Tserenjav, & Jun Zhen

Program Description

Our disassembler program is divided into three main parts, input output (I/O), opcode parsing, and effective address parsing. I/O handles getting the user's input and displaying output to the terminal. While opcode and EA parsing handles decoding the given assembled code. On a high level, our program will parse the memory from a starting and ending address. Then, as we go, we will decode the assembled code and output the appropriate instructions with its effective address. Figure 1 is a high level overview flowchart on our program's workflow.

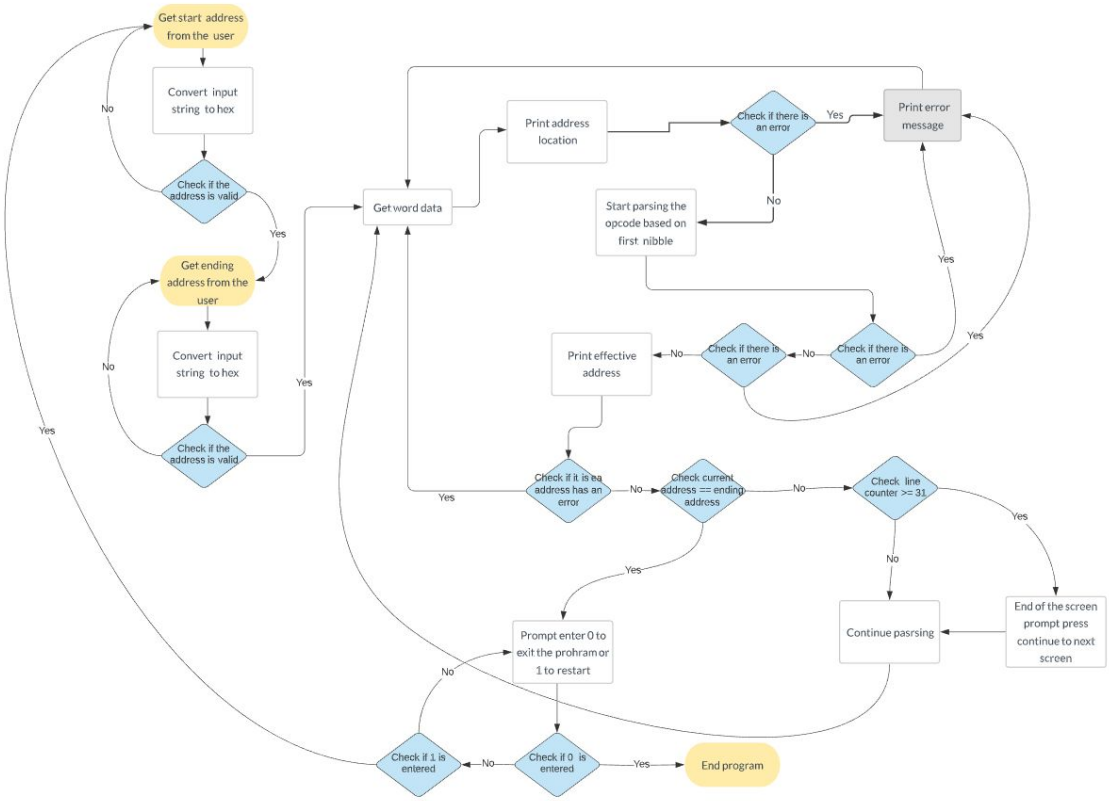


Figure1: Disassembler flowchart

Input output (I/O)

This section discusses our input output implementation, which includes input which deals with converting ascii values to hex values and output which does the opposite.

ASCII to HEX

We start the program by asking the user to input for both starting address and ending address. These starting and ending addresses are used as the range of addresses that the program is going to parse through. This might be simple at first glance; however, easy68k saves user inputs as ascii data to memory. Hence, we had to find a way to translate those ascii data to hex values in order to parse through the right addresses.

For getting user input, we used **Trap task #2**. Trap task #2 stores the ascii value of the input to (A1) and the length of the string to D1 [1]. For this program, we have set specific memory

locations to store the starting address and ending address which are $100 and $150 respectively. After Trap task #2 is executed, we move on to ASCII to hex converter subroutine.

Fortunately, since we have the length of the string in D1, we can use that as a checker if we are done converting the whole string or not. Conveniently, since ASCII values are one byte values, we can read and convert each byte iteratively.

Memory & Register Used

For this implementation, below are the registers that we are going to use:

- **A1: string source holder**
  The location in memory where we stored the address (as per Trap task #2) in ascii values; hence, we use that as our source.
- **A2: final starting address holder**
  After converting the valid address, we move the starting address to A2.
- **A3: final ending address holder**
  After converting the valid address, we move the starting address to A3.
- **D0: temporary byte holder**
  When reading each byte, we first put it in D0.
- **D1: string length holder**
  As per Trap task #2, D2 holds the string length.
- **D2: starting/ending address toggle**
  In the validation implementation, we need to know whether we're checking a starting address or an ending address. If D2 is zero then we're checking the starting address, ending address otherwise.
- **D3: temporary address holder**
  As we parse through the ascii values, we pile up the converted values/letters to D3.

Conversion

First we shift the temporary address holder by one hexabit with LSL.L #4, D3 to make space for the converted value. Then for each iteration we move one byte in (A1), while also moving A1 by two hexabits with (A1)+, to D0.

After moving the byte to D0, we then check if it's a number or letter. The if else workflow for checking number or letters can be summarized in the pseudo code below:

```
if ( D0 < '0') // bad input
        goToBadInputHandler();
if ( D0 > '9') // might be a valid letter
        goToLetterHandler();
// if reach this line then D0 is guaranteed to be a number
// convert D0 to number
```

Converting the byte in D0 to a number can be done by subtracting the ASCII value of 0 from the value in D0. Fortunately, easy68k let's the user use the number with single quotes ('0') to quickly convert the number (or any character) to its ASCII value. Therefore, we can call SUB.B

#’0’, D0 to convert the value in D0 to the equivalent hex number value. Then we add the byte in D0 to D3, while also clearing D0.

Converting ASCII values to letters has a similar implementation to converting to integer values. Instead of subtracting them with ‘0’, we subtract them with the hex value of ‘7’.

Before looping back to the conversion, we have to subtract the value in D1, similar to an i-- in object oriented programming. If after the i-- is 0 then we’re done parsing through the address string, then we move on to the validation implementation.

HEX to ASCII

In some parts of the program, we want to print out the information we have in either memory or register. These cases include, printing the current address we’re looking at and printing immediate data value. Easy68k stores everything in memory in hex format, and when we print it out, it reads it as an ASCII value. Therefore, we need to convert the HEX value to the ASCII equivalent.

On a high level we are using a jump table [2] by shifting the address by the value we’re reading when we jump to the jump table. Our jump table will be stored in a subroutine and we will utilize JSR 0 (A4, D3), with A4 as the address of the jump table subroutine and D3 as the address shift. We will be looking at one nibble (four bits) at a time and converting them to the appropriate ASCII value.

Memory & Register Used

For this implementation, below are the registers that we are used:

- A4: Jump table subroutine holder
- D3: Nibble / jump shift holder
- $500: stores the current nibble in memory

Conversion

As we are using a jump table, technically we are not converting the HEX value. Instead, we are using that like a switch case and return the appropriate ASCII value. Before we jump, we have to do an unsigned multiplication by six with MULU #6, D3. After that, we then can jump to the jump table subroutine.

In the jump table, the program will jump straight to the appropriate line and print out the appropriate value. This implementation is similar to a switch case implementation in object-oriented programming languages.

Opcode Disassembling

The general strategy for disassembling all the supported opcodes is by reading the first nibble. By reading the first nibble, we can generally tell what kind of opcode it is. We then group the opcodes that have similar first nibble together. By separating the opcodes into groups we can write different subroutines that will parse based on the groups. Since we cannot compare with

bits in 68K, we have to continually bit shift the opcode left and right to get the exact bits we want. For the first nibble we do not have to shift left because they are already in the front. If they were in the back we would have to left-shift until they are in the front and then right shift until the only bits left are the nibble we want and the rest are filled with zeros.

Group Red - MOVE/MOVEA

The first nibble for MOVE and MOVEA is always 00 and then the size. Since the size is always between %01 - %11, we will know that the opcode is either MOVE or MOVEA if the first nibble is either $1, $2, or $3.

Now we have to tell if it is MOVE or MOVEA. Since MOVEA's destination mode is always going to be 1, we have to check Bit-11 to Bit-9.

- If it is equal to $1 then it is MOVEA.
- If it does not equal $1 then it is MOVE.

To tell which size the opcode is, we go just read the first nibble again.

- $1 equal to Byte size.
- $3 equal to Word size.
- $2 equal to Long size.

Group Orange - MOVEM/LEA/JSR/RTS

Every opcode's first nibble in Group Orange is always 0100. If an opcode's first nibble is equal to $4 then they belong to this group. The following are the conditions to determine which opcode it is.

- MOVEM's Bit-10 to Bit-7 is always $1, if the Byte data is equal to that then the opcode is MOVEM. We tell what the size is by reading Bit-6.
- LEA's Bit-8 to Bit-6 is always $7, if the Byte data is equal to that then the opcode is LEA.
- JSR's Bit-11 to Bit-6 is always $3A, if the Byte data is equal to that then the opcode is JSR.
- RTS is always $4E73, if the whole Word data is equal to that then the opcode is RTS.

Group Yellow - BCC/BGT/BLE

Every opcode's first nibble in Group Yellow is always 0110. If an opcode's first nibble is equal to 6 then they belong to this group. Within this group, the opcodes must be one of the branch condition opcodes. We will check the second nibble to determine the exact opcode.

- BCC's second nibble is always 0100, if the Word data is equal to $4 then the opcode is BCC.
- BGT's second nibble is always 1110, if the Word data is equal to $E then the opcode is BGT.
- BLE's second nibble is always 1111, if the Word data is equal to $F then the opcode is BLE.

Group Green - OR/SUB/CMP/AND/ADD

Every opcode's first nibble in Group Green is different, but they all have similar structure to their bits. Bit-7 to Bit-6 are always for the size.

- OR's first nibble is always 1000, if the Word data is equal to $8 then the opcode is OR.
- SUB's first nibble is always 1001, if the Word data is equal to $9 then the opcode is SUB.
- CMP's first nibble is always 1011, if the Word data is equal to $B then the opcode is CMP.
- AND's first nibble is always 1100, if the Word data is equal to $C then the opcode is AND.
- OR's first nibble is always 1101, if the Word data is equal to $D then the opcode is ADD

Since it only ranges from 0 to 2, if it is 3 then it must be an unsupported opcode.

Group Blue - LSL/ASR

Every opcode's first nibble in Group Blue is 1110, if the Word data is equal to $E then the opcode belongs in this group.

- LSL's Bit-4 to Bit-3 is always 01 and Bit-8 is 1. If both conditions are true then the opcode is LSL.
- ASR's Bit-4 to Bit-3 is always 00 and Bit-8 is 0. If both conditions are true then the opcode is ASR.

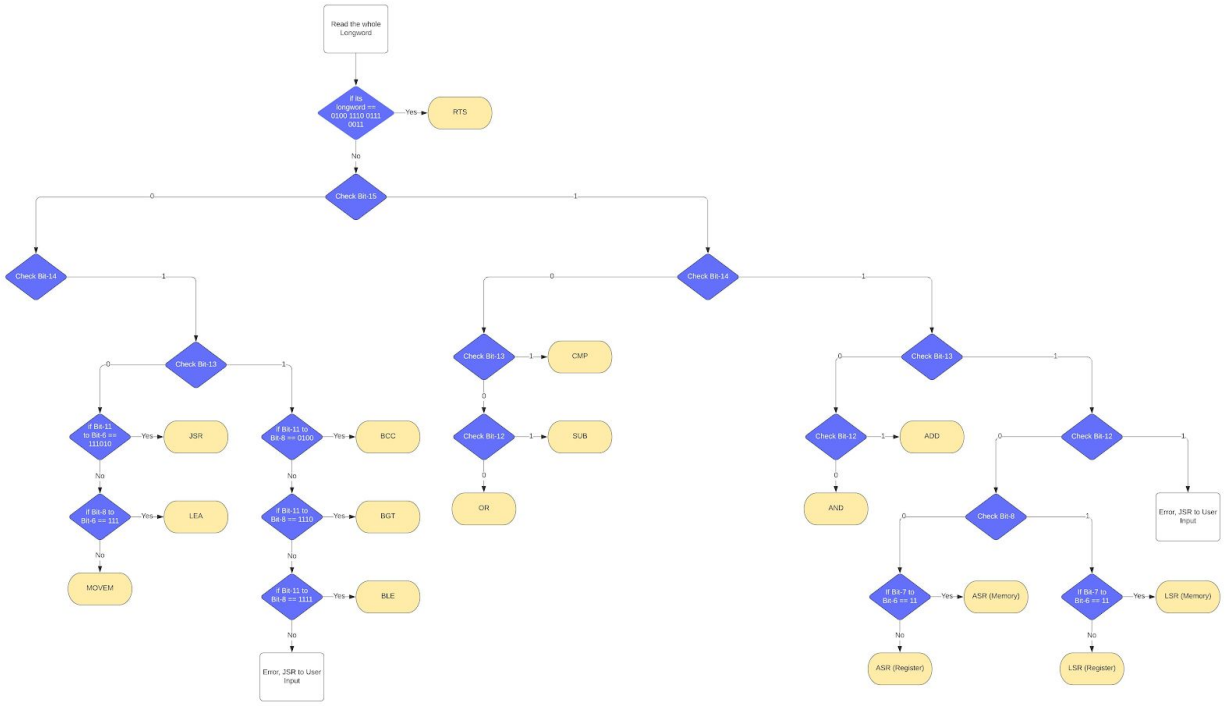
To determine if it is memory shift or register shift, we can tell by Bit-8 to Bit-7 which is the size bit. Size bits for register shift cannot be over %01, if it is %11 then it is memory shift.

Memory & Register Used

For Opcode implementation, below are the registers that we are used:

- **A2: final starting address holder**
  After converting the valid address, we move the starting address to A2.
- **D6: final bits after shifting holder**
  After shifting hex bits, it will be stored in here to be used to compare with conditions.
- **D7: Current Word data holder**
  After reading the word data in memory, store the current word data here.

Flowchart



Effective Address Disassembling

Our approach to disassembling effect addresses is very similar to how we did the opcodes. We essentially are building upon the groundwork we laid out from parsing the opcodes. After the program is done parsing the current opcode and we know what the opcode is, we then can write individual subroutines to get their effect address. While the majority of the opcodes' effective addressing structure are different, they all still share one similar trait. That being Bit-5 to Bit-3 is the addressing mode and Bit-2 to Bit-0 is the addressing register. With this information, we wrote a subroutine that extracted that information out and was able to reuse it consistently with every opcode.

Group Red - MOVE/MOVEA

- MOVE - The destination address is located Bit-11 to Bit-6 and the source address is at Bit-5 to Bit-0.
- MOVEA - The destination mode is always going to be #1. Therefore, we only need to look at the destination register which is located at Bit-11 to Bit-9 and the source address is at Bit-5 to Bit-0.

Group Orange - MOVEM/LEA/JSR/RTS

- MOVEM - First we read the Bit-6 to know if it is a register to memory or memory to register. Then we move the current address to the next word which contains the register masking. We then read the Word data bit by bit to determine which registers.
- LEA - The source mode is always going to be #7. Therefore, we only need to look at the source register which is located at Bit-11 to Bit-9 and the destination address is at Bit-5 to Bit-0.
- JSR - The destination address is at Bit-5 to Bit-0.
- RTS - No effective address.

Group Yellow - BCC/BGT/BLE

- BCC/BGT/BLE - First we get the displacement by reading the next Word data. Then we add #2 and the current address to the displacement to get the destination address.

Group Green - OR/SUB/CMP/AND/ADD

- OR/SUB/CMP/AND/ADD - We first read Bit-7 to Bit-6 to determine if its effect address to register or vice versa. After that the destination address is at Bit-5 to Bit-0.

Group Blue - LSL/ASR

- LSL/ASR - First read the size to determine if this is a memory shift or register shift. If this is a memory shift then the destination address is located at Bit-5 to Bit-0. If this is a register shift then the register number is located at Bit-11 to Bit-9 and the register mode is located at Bit-2 to Bit-0.

Memory & Register Used

For EA implementation, below are the registers that we are used:

- **A2: final starting address holder**
  After converting the valid address, we move the starting address to A2.
- **D5: calculating Bcc effect address.**
  We add the current address at A2 to the displacement to get the destination address.
- **D6: final bits after shifting holder**
  After shifting hex bits, it will be stored in here to be used to compare with conditions.
- **D7: Current Word data holder**
  After reading the word data in memory, store the current word data here.

Specification

This section talks about what the overall program does.

Getting Starting & Ending Addresses

This program has to be given a range of addresses to parse through. In order to do that, the program prompts the user to enter valid addresses for both starting addresses and ending

addresses. The program only accepts one to eight bit addresses which includes integers (0 - 9) and capital hex letters (A - F).

Another constraint for the addresses are addresses must be even addresses and the ending address must be greater than the starting address.

Disassembling Code

This disassembler program will prompt the user to get a range of addresses. This range of addresses will be the range that the program will parse through. For each address it parses, it will disassemble the code into the appropriate instructions. The supported opcode instructions are:

- MOVE
- MOVEA
- MOVEM
- ADD
- SUB
- LEA
- AND
- OR
- LSL
- ASR
- CMP
- Bcc (BCC, BGT, BLE)
- JSR
- RTS

When parsing, we will display the address where it was stored, the converted instructions, both address registers and data registers if applicable, and the immediate data if applicable. If we stumbled upon unsupported instructions, we will just display “Invalid Opcode”.

Besides the opcode, this program also decodes the effective addresses of the code. Supported effective address modes are:

- Dn; Data Register Direct
- An; Address Register Direct
- (An); Address Register Indirect
- (An)+; Address Register Indirect with post increment
- -(An); Address Register Indirect with pre increment
- #<data>; Immediate Data
- (xxx).W; Absolute Word Address
- (xxx).L; Absolute Long Address

Printing One Screen at a Time

Since the terminal in easy68k sim is limited to only 30 lines at a time, we implemented an output stopper for every 31 lines (30 lines of output + one line for prompt). After printing 30 lines, the program will prompt the user to press enter to continue printing output if needed.

Restarting Program

After parsing through the given addresses, the program will prompt the user to either restart or end the program. The user can enter ‘1’ to restart or ‘0’ to end the program. If neither is given then the program will keep prompting.

Test Plan

For testing, we implemented an agile testing methodology by testing features as we implemented them. Since sometimes we implemented a feature individually, we believe that it'd be better if we test them in an isolated environment before combining it to the 'master' source code. This works very well in our case since most features are independent from one another. For example, one of our members implemented the ASCII to HEX conversion and tested it separately from opcode parsing.

After testing features separately, we then combined them in one master source code where we then again test the workflow between these implemented features. We wrote another test file to test the overall program with both supported and unsupported instructions.

The features that were tested included:

- Getting user input
- Validating user input
- Opcode disassembling
- Effective address disassembling
- ASCII to Hex conversion
- Hex to ASCII conversion

For testing opcodes and effective addresses. Each time we were able to parse an opcode, we would test with different variations of the same instruction. For example if we were testing MOVE, our test file would include data register to data register, data to address, immediate to register, memory to register and etc. We would test each variation one at a time and if all variations are successful, we then combine them into one large file to test. We also use this methodology when testing different kinds of opcode. Once an opcode is considered finished, we then test the newly finished opcode with the previous finished opcode. By doing this we can see if the program is actually disassembling correctly or not. After every opcode is considered complete, we move on testing with bad input data. We want to make sure that the bad data would not affect the good input data. Once the program was able to discard bad input data and only disassemble good input data, the program was considered finished.

Exception Report

- For getting user input, we decided to just support capital hex letters (A - F) and not lower case letters (a - f).
- For bad input data, since we are reading the memory word by word, if the bad data is two words, it would read it and discard it as bad data and then read the next word as new opcode. That 2nd word data would be parsed because we would have no way of telling how many words belong to the same instruction. We have asked the professor and she states that bad inputs are only 1 word long. If that is the case, everything in theory will run perfectly.
- For Easy68K bugs such as MOVE sometimes get translated to MOVEQ, if that is the case then it will be flagged as bad data.

- However we made an expectation for arithmetic instructions such as ADD and SUB. These instructions sometimes get translated to ADDI or SUBI. If that is the case, we simply catch it and print it as ADD or SUB. So in our program ADDI is the same as ADD.

Team Assignments & Report

In our final project, we divided the work in between I/O person, Opcode person and EA person. However, when we started implementing our part we helped each other and decided to do that through the project. Jeremy's contribution was managing the project, setting up the group github repository, organizing the files, I/O part, converting the ASCII to Hex, Hex to ASCII, and printing the output screen by screen. Angie participated in part of the conversions, I/O parts, EA parts, created the test files for the implementation, creating the flowcharts and prompting the user restart or exit at the end of the program. Jun handled parsing the 14 of the Opcode, and part of EA. Based on the effort each group members put in this project, we break down each team member's participation in percentage.

- Jun Zhen - 40%
- Jeremy Tandjung - 30%
- Angie Tserenjav - 30%

We provided more detail of our contribution to this project in an individual report.

References

1. http://www.easy68k.com/QuickStart/TrapTasks.htm
2. https://canvas.uw.edu/courses/1387438/pages/addendum?module_item_id=10501791