Disassembler Project Documentation

# CSS 422: Hardware and Computer Organization

# Spring 2021

# Awad Ahmed

Team Huskies:

Narapady Chhuoy, Divyashna Chandra, Hafsa Ali

## **Introduction**

Our program, the disassember’s purpose is to disassemble assembly instructions between a memory range given in by the user. Following the project requirements our project will parse the op-code word of the instruction and then decide how many additional words of memory need to be read in order to complete the instruction. The disassembler program prints out the complete instruction in ASCII- readable format.

## **Program Description**

### **Design Philosophy**

The program starts with a welcome page with instructions on usage. Basically, when the program is run, a question **“Would You Like To Use 68K Disassembler? (Y/N)”**  pops up, and users can input “Y/y” to start disassembling, or enter “N/n” to exit the program. Then users are required to enter the starting and ending memory in hexadecimal format without the dollar sign, and A-F (10th to 16th number of hex) should be in upper case. Otherwise, invalid message will be shown and let the user re-enter addresses. When a starting and ending address are valid, the program outputs 15 instructions to the simulator (disassemble) per page. The users have to press enter for the output of next pages. When done disassembling, the program goes back to the above question asking if users want to use it again. There is a main loop that controls the flow of the program. The main loop is keeping track of the current address and the opcode to be decoded until the ending address is reached. **The 3 main routines of the program are I/O routine, opcode routine and EA routine.**

### **I/O Routine**

The first part handles input from the user which includes the validation and conversion of inputs. The validation process involves 3 subroutines including checking the length of address (address must be 8 digit hexadecimal formats without dollar sign, for example, 000040CA), checking if the address is even, and checking if the letter(s) is upper case. Invalid input will be handled by I/O subroutine; a message about invalid address is shown, and users are to re-enter the address. Then both starting and ending addresses will be converted from ascii string to hexadecimal number for comparison; if the ending address is smaller than the starting address, it’s not valid.

After address validation, the current address will be converted back to ascii string and added to the buffer I/O subroutine. This subroutine is responsible for preparing string buffers and output ascii string in the buffer container to the screen. It received 3 parts of ascii string including the current address decoded instruction and decoded effective address based on opcode word.

**Main Disassembling Loop**

After addresses are successfully validated, the program then enters a main loop which includes the **opcode subroutine and EA subroutines** to disassemble the instructions one at a time . First the opcode is validated and the bit from the opcode word instruction is isolated and assigned to variables. To decode the opcode the program compares the isolated bits of the opcode to possible values to find a match. Then the effective address field is decoded based on opcode. At each section the result is converted to ascii-readable format and added to I/O buffer sub routine for printing. The loop continues until the starting address equals the ending address. The main loop also has a subroutine to handle when the page is full meaning that users have to press Enter to see the following pages of output.

**Opcode Routine**

Opcode routine starts with bit isolation. Bit isolation subroutine isolates the 16 bits of opcode word into 4 parts which are bit 15 to 12, bit 11 to 9, bit 8 to 6, bit 5 to 3, and bit 2 to 0. Then the first value of the opcode word (bit 15 to 12) is used to identify the instruction to be decoded. Bellow are starting value of the opcode and its corresponding groups:

**First value of opcode: Group**

* $0 Unsupported opcode
* $1 MOVE (Byte)
* $2 MOVE and MOVEA (Long)
* $3 MOVE and MOVEA (Word)
* $4 MOVEM, JSR, RTS, LEA, NOP, and NOT
* $5 ADDQ
* $6 BRA, Bcc (BLE, BEQ, BGT)
* $7 MOVEQ
* $8 OR
* $9 SUB
* $A Unsupported opcode
* $B Unsupported opcode
* $C AND
* $D ADD and ADDA
* $E LSR, LSL, ASR, ASL, ROR, and ROL
* $F Unsupported opcode

Once instruction is identified, it will be decoded and added ascii string to the buffer I/O subroutines. Next, the opcode word is sent to the EA routine to be decoded. Any unsupported opcode or the ones that can't be decoded will be output to a screen showing the memory address and the data of the unsupported opcode.

**EA Routine**

The operands are decoded based on the opcode word. EA routine essentially uses the bits that are isolated in the opcode routine to identify 8 supporting operands. Basically, EA routine handles 3 types of EA including EA with both source and destination operands, EA with only destination operand, and EA without any operand such as NOP and RTS. EAs are decoded differently based on the instruction format of the 16 bits opcode word, but some instructions have the same or similar characteristics and patterns. For instance, branching instructions (BRA, BEQ, BLE, BGT) have the same EA pattern. Similarly, data movements such as MOVE and MOVEA share the same decoding process. ADD, SUB,OR, and AND also have a common EA decoding process. Lastly, bit shifting and rotating instructions such as LSL, LSR, ASR, ASL, ROR, and ROL have a similar decoding pattern with slight differences. After EA is decoded, the ASCII string of EA will be added to the buffer I/O subroutine and output to the screen

**Algorithm**

Our program algorithm is very simple and straightforward. We decode each part and add to the buffer variable one at a time in order of address, instruction, and operands. There is no printing until the opcode in the current address is fully disassembled. However there are two canned subroutines from another source that we used, that is, the I/O subroutine that converts hexadecimal numbers to ascii strings and the subroutine to handle when the output page is full. The original source code for the copied subroutines can be found at this [GitHub link](https://github.com/mitchjcarlson/EASy68K-Disassembler).

**Limitations**

* **I/O Limitations**
  + There were not many major limitations we are aware of in this routine, except for a small one, which is that the user will need to enter the address in 8-digit hexadecimal format, and no other format since the program will not take any other format and convert it to hexadecimal. This problem is taken care of, since as the user enters the address, the program checks if the address is valid but this is only true for the first round of disassembling.
* **EA Limitations**
  + The only limitation in this routine we encountered and is known at the moment is not being able to make the program run through the instructions of decoding the EA and handling the opcode after its first run through, when the user wants to use the program again after running it once, it won’t run as efficiently and will not produce the correct results.
* **OpCode Limitations**
  + One of the limitations we are aware of is that not being able to get the program to fully follow the instructions it needs to fulfil, instead of it running through instructions such as MOVEM.L (SP)+ or MOVEM.W A0-A4, -(SP), the program goes through instructions such as MOVEM.L (SP)+ MOVEM.W A0/A1/A2/A3/A4.

### **Flowchart of the program**

### **Specification**

The following list states what our program does

1. Program starts with a greeting message
2. Prompts users to enter “Y/y” to start the program or “N/n” to exit the program.
3. Prompts user to enter the starting address
4. Prompts user to enter the ending address
5. The program checks if the valid addresses are entered (valid size, boundaries and difference which needs to positive)
   1. If valid, then continue
   2. If not valid, then prompt user again
6. Valid address is added to the buffer.
7. Jump to Opcode and starting decoding opcode word
8. Opcode is added to the buffer
9. If need be then jump to EA to decode the addressing mode
10. Decoded EA is added to the buffer
11. Prepare buffer and print to the screen
12. Screen shows 15 instructions at a time
13. Press the “Enter” key to see the next output pages
14. The program goes back to (1) when done disassembling.

**Supported Opcodes**

|  |  |  |  |
| --- | --- | --- | --- |
| **NOP** | **CMP** | **BNE** | **BLT** |
| **MOVE** | **MOVEQ** | **MOVEM** | **MOVEA** |
| **ADD** | **ADDA** | **ADDQ** | **SUB** |
| **AND** | **OR** | **NOT** | **LEA** |
| **LSL** | **LSR** | **ASL** | **ASR** |
| **BGT** | **BLE** | **BEQ** | **BRA** |
| **ROL** | **ROR** | **JSR** | **RTS** |

**Supported Effective Addressing Modes**

* Data Register Direct
* Address Register Direct
* Address Register Indirect
* Immediate Data
* Address Register Indirect with Post incrementing
* Address Register Indirect with Pre decrementing
* Absolute Long Address
* Absolute Word Address

**Test Plan**

**How to test**

We use a separate TEST\_CODES.X68 file to test our program. At the bottom of the program we use the 68k directive “INCLUDE” to import our test file to our program for testing. The test file contains all supported opcode and EA in the program's specification. We also have some unsupported instructions to see if the program can handle that properly. The test file starts at memory location $4000 until $41CA. The users have to run the main program and enter preferred starting and ending addresses for the program to disassemble.

**Details**

During the development of the program we incrementally tested in a 3 part phase which includes, identifying bugs in the program, testing individual components of the program to meet the specifications and validating the functionality of the entire program. We did a rigorous review of the program at each phase to find any issues. When we identified them we recorded them in a log of bugs that needed to be fixed before proceeding to the next phase. Small bugs which we identified were fixed incrementally with the appropriate modifications to the program. To ensure the entire program would work as expected we tested individual components of the program including the I/O routine, opcode routine, EA routine and various other subroutines. We tested the I/O by entering a combination of possible user input such as incorrectly formatted input, out of bounds address and unrecognized characters to ensure the program would appropriately handle/validate both valid and invalid user input. For the opcode routine we tested that the program could correctly identify corresponding instructions based on the isolated bit and first value of the opcode and handle unsupported opcodes. Also the opcode routine was tested to ensure that once a match is found it is able to be added to the buffer to be displayed. Also that it can successfully send the opcode to the EA routine for further decoding. For the EA routine we focused on ensuring that the program could take the opcode and based on it figure out the source and destination. Also if the EA is invalid it is able to handle this scenario. Since the EA can be formatted differently we ensured that the program could recognize the pattern and output the correct corresponding instruction. Once the EA is decoded we checked that the program was sending the ascii is added to the buffer for printing. During this testing phase we also considered subroutines and helper functions such as the helper function for shifting, rotation and bit isolation subroutines to validate their functions. Once we completed testing the different components of the program we tested the program overall to ensure there were no major bugs, the program functions as expected and that the logic and flow was correct across the program. To do this we developed a separate testing file which contains a number of possible assembly instructions. This includes both all the required as well as supported and unsupported opcodes and EA as part of the instructions. We loaded this testing to the program and provided a starting and ending address to start the decoding process. We checked that the program was correctly decoding the instructions and that there were no errors in the process. We also checked that the program decodes the instructions one at a time and displays them in the correct format. We have included the testing file as part of the project submission.

## **Exception Report**

There were a couple of issues we encountered in our program which are working now, but we thought of just mentioning. First one is when the program finishes disassembling for round one and the users want to continue using it again it would work as expected, but we got it to work by looking closely at the code and finding where the problem was, it was not clearing out D6 when instructed and rather put some other random number which caused the other runs to generate the wrong result. Another issue we couldn’t fix is related to MOVEM. Actually, it’s not a technical issue, but a formatting issue. For example, if the opcode word is corresponding to MOVEM.L (SP)+, D0-D4 or MOVEM.W A0-A4, -(SP), our program will instead print MOVEM.L (SP)+, D0/D1/D2/D3/D4, MOVEM.W A0/A1/A2/A3/A4, -(SP) respectively.

## **Team Assignments and Report**

Our group’s task distribution is intertwined meaning that we all worked on 3 main routines together. We had our own revised version of the program each time we had a group meeting. We compared our code or solution, and then picked the most optimal one. We continued the same process until we had the final version of the program.

|  |  |  |
| --- | --- | --- |
| Assignment | Team Member and Role | Description |
| I/O Routine | Narapady: 40%  Divyashna: 30%  Hafsa: 30% | Handled all input and output functions of the program. Checked for valid addresses and converted from ASCII to hex and vice versa. |
| Opcode Routine | Narapady: 40%  Divyashna: 30%  Hafsa: 30% | Decoded the Opcode, also developed main opcode decoding functions and passed to EA person. |
| EA Routine | Narapady: 40%  Divyashna: 30%  Hafsa: 30% | Handled the EA coding, and dealt with debugging and opcode implementation. |