Cloud Guys Disassembler Final Report

[Program Description 2](#_Toc58358988)

[Specification 7](#_Toc58358989)

[Test Plan 8](#_Toc58358990)

[Exception Report 10](#_Toc58358991)

[Team Assignments and Report 11](#_Toc58358992)

Program Description

*This program converts machine codes between two specific memory locations back to 68000 assembly language and outputs the disassembled code to the screen. The disassembler program consists of three parts, Input, Opcode, and EA (Effective Address).*

**I/O - Address Input**

Text, email

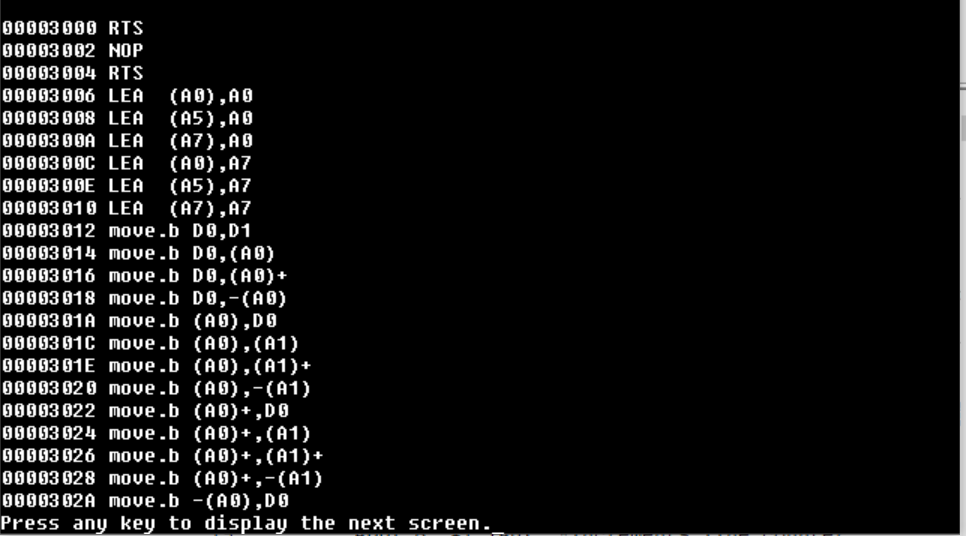
Description automatically generatedThe program starts with a welcome message printed to the console, “Welcome to the Cloud Guys Disassembler Program.” Then a message that gives the specification for input of addresses that states, “Addresses entered must be even 8 digit numbers within the specified range.” The specified ranges we decided on for our program are 00003000 and 0000FFFF. These are ranges in which a source file can be loaded into the program as well.

When the user enters an appropriate starting address the console prompts for an ending address that is within the appropriate starting address entered. Whenever a user enters an address that isn’t compliant with the input requirements the console prints out “Invalid Address” then prompts the user to enter an address again.

Text

Description automatically generated

**I/O - Instruction Output**

The IO instruction output portion of our code only prints the address of where the opcode is in memory. This address is the current address that the program loops through until it gets to the ending address. Each instruction is read in, decoded, and printed before reading in the next instruction. The opcode gets decoded by branching to the section it corresponds. The opcode gets printed out based on the first few bits of the instruction then depending on the EA modes that have been set the EA portion of the instructions gets printed out. How opcode and EA modes are printed is further explained below in the Opcode and EA sections. If the output from the file is more than 30 lines, then the program asks to input any character to print the next screen. Once all the output gets printed out the program asks the user if they would like to continue and they can input ‘Y’ to start again or ‘N’ to stop the program. If none of these options are inputted during this part, then the program asks if they would like to continue again until a valid character is inputted.

Text

Description automatically generated

Text

Description automatically generated

**Opcode**

After two memory address locations were obtained in the input part, the program starts looping from the starting address to the ending address and converts each machine code back to its assembly language. In this opcode part, a machine code is copied and divided into opcode and EA. Then, the program compares the opcode with a list of opcodes required for this project. If there is a match, the program branches down to a subroutine that prints out its assembly language and EA, and if not, the program branches down to a subroutine that returns an invalid code message. Before branching to an EA subroutine, the program also divides an EA code into four parts, source mode, source register, destination mode, destination register, and copies each of them into data register D4 to D7.

**EA**

In the EA part, the program checks the source and register mode and prints out proper EA right after its disassembled opcode. Since the EA of each opcode is structured differently, the program takes care of EA operation depending on the opcodes. EA of the following opcodes, such as MOVE, MOVEM, SUB, ADD, MULS, and AND contains two effective addresses, source, and destination; thus, the program calls a subroutine called “eaPrint” to print out the source and destination address. Source and destination address consist of two parts, mode, and register. EA has seven different fields, Dn, An, (An), (An)+, -(An), (xxx).W, (xxx).L, and #<xxx>. Each field has a different code stored in the mode field, and the register field contains a register number expressed as ‘n’. In eaPrint, the program checks the mode field and prints the letters out, and then prints out the register number in the decimal format right after the mode field. Whereas, the last three EAs, (xxx).W, (xxx).L, and #<xxx>, works differently from the other first five EAs. Those three EAs have the same mode field but different register field numbers. It’s because its address data is stored in a memory, and the eaPrint needs to read and copy the data from memory and print it out.

**Program FlowDiagram, schematic

Description automatically generated**

**Some algorithms**

**Converting char to hex digit**

This routine is useful for the reading in the address part of the program. It places a char from input and puts it in a register and converts it to a hex digit by subtracting $30 from the char value. Once the input is in hex then it can be used for comparing hex values like our starting and ending bounds.

**Converting hex digit to char**

This routine is useful for the reading in the address part of the program. It places a char from input and puts it in a register and converts it to a hex digit by adding $30 to the char value. Once hex value is in the char equivalent the values can be stored in a variable used for printing out.

**Breaking down the opcode**

Our program starts by reading in the opcode to d1, then moving it to d2. We then divide the opcode in d2 by $1000, to isolate its first nibble. We swap d1 (to create storage space) then move the first nibble up to d1 for reference later.

Our program then breaks down the last three nibbles in the opcode by using andi.w #0007 to isolate its last 3 bits the shifting. The program has a counter to tell which data register to store it in for use later. The first three go to d5, for source register; the second three d4, for source mode; the third go to d6, for destination mode; and finally d7, for destination register.

Our program then checks the opcodes first nibble in d1 and either branches to the opcodes' correct code block, to print the opcode type or prints out “invalid opcode.” We had to write the code in this order so everyone could use the next part of our program, which prints the source and destination registers.

**Printing source and destination registers**

Most of the opcodes in our program use eaPrint to print the source and destination registers. eaPrint would not function correctly if our program did not break down the opcode the way it does. eaPrint first checks to see what is in d4, or the source mode, and prints it out, It then checks d5, the source register, prints that out. It then prints out a comma and checks d6, the destination mode, and prints it out. Finally, the program checks d7, destination source, prints it out, and branches back up to read in the next opcode.

Specification

**Program**

* Build and run on Easy68k
* Require two source file (.x68) for the program and its test cases

**I/O**

* Print out welcome messages and program instructions
* Take memory address range between $2000 to $FFFF
* Ask an user to insert starting and ending address
* Check if an input address is within the range. If not, print out error message and ask for a new input address
* Ask an user to repeat a program again
* Repeat the program until a user selects to quit

**Opcode**

* Disassemble BCC, LSL, ASR, BLT, BGE, BEQ, SUB, ADD, AND, MULS, MOVE, BRA, NOP, NOT, RTS, JSR, LEA, MOVEM
* Handle invalid opcodes and print out a message for invalid
* Break a machine code into opcode, source mode, source register, destination mode, and destination register
* Check opcode and branch to a subroutine which the opcode is matched. If it’s a invalid opcode, branch to a subroutine to print out a invalid code
* Print out the opcode
* Check and print the size of opcode if the opcode has one

**EA**

* Disassemble Dn, An, (An), (An)+, -(An), Immediate, Absolute word address, and Absolute long address
* Check the mode and branch to a subroutine which the mode is matched.
* Prints out EA

**Test codes**

* Starting memory address of a test file is $2000
* Contains all possible opcode/EA combinations for required opcode and invalid cases

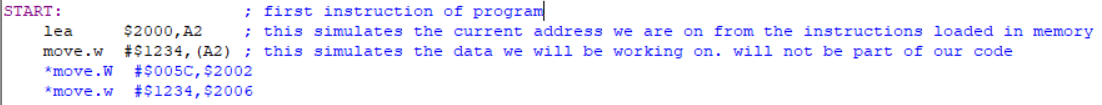
Test Plan

The program was tested in two phases. During a development phase, each part of codes are individually tested by a member who wrote the codes. After completing each task, we combined the program codes into one source file and created a test file which contains test cases for all required opcodes and effective address combinations including invalid cases. We also compiled a list of invalid opcodes to test as shown below to the right.

|  |  |
| --- | --- |
| **I/O**  **Address Input**  Graphical user interface, table  Description automatically generatedThe address input part of the program was tested by entering addresses that were not compliant with the input requirements and addresses that were. Types of inputs that were tested were addresses that were less than eight characters, less than the starting boundary, less than the new starting address set, and greater than the ending boundary.  **Instruction Output**  When testing the instruction output part of the program we individually started off testing our output by loading a sample instruction into memory and have it go through the decoding opcode part of the program to see if the console prints out the correct opcode. Later on we added the EA printing portion of the program and tested it with one instruction loaded into memory. |  |
|  |  |

**Opcode**

Each opcode was tested individually first by loading them into memory and making sure it printed the correct result. We mostly tested them in a very early version of our program by loading the hex value into memory, then stepping through the code to make sure the program broke them down correctly. We did it this way so we wouldn't have to step through the IO part of our program. This is shown below.



**EA**

The ea in our program was tested mostly by testing eaPrint. We tested to make sure every addressing mode was stored correctly in data registers four and six. We tested that data registers five and seven to make sure the correct register numbers were printed. We did all this by testing every combination in a very similar way we tested the opcodes above.

**The Entire Program**

We tested the entire program in two phases like mentioned above. After testing individually in the first phase, we created a test file with all the correct combinations of opcodes. We then later tested the program with invalid opcodes that were obtained from the Motorola 68000 CPU Opcodes table.

Exception Report

**Movem**

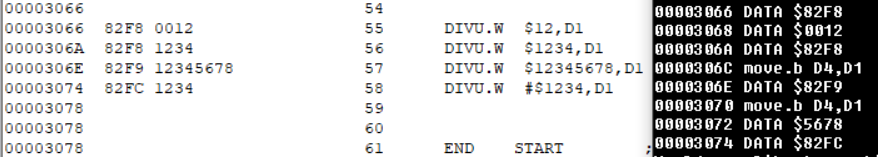
We came up with an algorithm that we really liked for movem, but we didn't have enough time to implement it. There were just too many combinations of addresses. We found the first word after the movem opcode was for the memory locations, either An’s or Dn’s. A0 was 0100, A1 was 0200… D0 was 0010, D1 was 0020... The combination of them was just the hex addition of the code. With so many combinations and not enough time, we decided it would be best to just print out the word after the movem opcode and just make sure the program did not crash.

**Opcode**

* There was an issue using the JMP table for the opcode part. The problem occurred when there was a branch (BRA) or subroutine (JSR) right before a JMP table, and it ended up skipping the branch or subroutine. As discussed in class,  the problem seemed to be caused when storing PC value. Tomomi liked to use the JMP table because the code is more organized than using compare and branch methods (CMP & BEQ). However, the solution she could come up with was to add any assembly code after the branch or subroutine. Since she did not like adding an unused line, she decided to use compare and branch methods instead.

**Invalid opcodes**

Our program can't have any invalid opcodes tested if they have any additional addresses or data that needs to be read in after them in memory. For instance, the DIVU cases below are these cases. The first code will be fine. It will read in divu, throw an error, then read in 0012 and throw an error for it as well. The second line will read in divu, throw an error, then read in 1234 and think it's a move. The third line will read in 1234, then 5678 as two opcodes.



Also, if an opcode is converted to an invalid opcode, our program will not be able to recognize it. For instance, SUB #$12, D1 is converted to SUBI. Since SUBI is an invalid opcode, our program will not decode SUB #$12, D1, it will print an invalid opcode error. The same would be true for SUB D1,A1 since that is converted to SUBA. ADD also has this behavior.

Team Assignments and Report

Throughout the quarter, we all attended weekly meetings and communicated well to plan and keep track of the progress. We spent the first two weeks on analyzing the project details and planning how we could approach the project. Then, we assigned each member to work on 3 to 4 opcodes.  Charlie worked on the input handling for the starting and ending address, and her assigned opcodes.  Sean worked on his assigned opcodes and structured a pseudo code for the entire program including breaking down instructions that have registers and EA in them. Bisrat worked on his assigned opcodes and EA mode handling for the ones that don’t use the EA print. Tomomi worked on her assigned opcodes and added to the EA print functionality.

Each member analyzed and implemented their assigned opcodes in their local file, and after passing all test cases, the code was added to the file. We used this file that Sean created to be able to test our individual opcodes against and to test our opcodes with others.

This project was the definition of Agile programming. We had multiple design ideas, and multiple ways we split up the work throughout the quarter. It seemed like as soon as we made the decision to go in one direction, we found a better direction almost immediately. The reason we felt like we were successful in this project was because of how flexible and patient everyone on this team was. We all worked very well together and we met at least once a week on every Saturday at 10 am. Some of us really enjoyed working with 68k assembly and are disappointed that we probably will not be using it again. Although it was very stressful, overall we felt this assignment was very rewarding in the end.

|  |  |  |
| --- | --- | --- |
| Member | Codes (%) | Task |
| Bisrat | 25 | * Implements both opcodes and EAs for NOP, NOT, RTS, JSR, LEA, MOVEM, MULS.L |
| Charlie | 25 | * Implemented the input handling for getting the starting and ending addresses. * Decoded Bcc, BLT, BGE, BEQ, BRA * Did some decoding of SUB |
| Sean | 25 | * Decoded Muls.w, and, move.b/w/l, sub, * EA work for EA print * wrote code to break down the opcode and load info about the code into Dn’s 4-7. |
| Tomomi | 25 | * Opcode and EAs for ASR, LSL, ADD * some EA work with EA print. |