Johnathon Hewit & Scott Blake

CSS 422 – Dr. Munehiro Fukuda

12 March 2020

Final Report

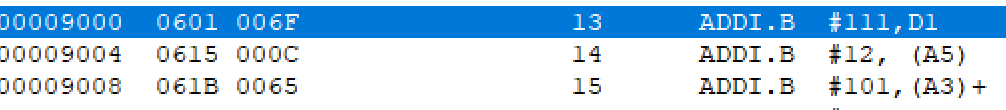
# Specification

This program converts memory images of instructions and data back to Assembly code instructions. What does that mean exactly? It means that in the snippet below, it will first take in a start (first line shown at far left) and ending address (last line shown at far left of the screenshot below) by input from the user (specified in the test program’s listing file).

In order to run the program, start EASy68K and load our program. Next, click the Execute button. This will generate a listing file and an s-record file and start the program. You will then need to load the desired test file in order to use Disassembler 3000. Click load, and select the Source file of your test program, then run Disassembler 3000 by hitting the play button. As mentioned before, the user will need to enter a valid starting address, and a valid ending address. To ensure these are valid, the program will check the input upon entry, and notify the user if they entered an invalid format or invalid address. Disassembler 3000 will accept an address up to 8 hexadecimal characters long, provided they entered all in uppercase if they are the characters A-F. Our program’s stack begins at $000A000, and will reject a starting address that begins at that address, or the origin address of $1000.

Next, the program will then begin reading instructions from the start address and not continue reading past the end address, printing out valid instructions from the test program. If the number of readable instructions amounts to 31, the output window will be full, and Disassembler 3000 will prompt the user to press Enter to continue decoding. If Disassembler 3000 encounters an invalid Op Code, the it will notify the user, and start the program from the beginning. If it reaches the ending address, the program will ask the user if they would like to continue decoding, where it will expect an entry of “y” or “Y” for yes, or “n” or “N” for no. If the user enters an invalid response, the program will prompt the user to re-enter their response. If yes is entered, Disassembler 3000 will start from the beginning. If no is entered, Disassembler 3000 will exit.

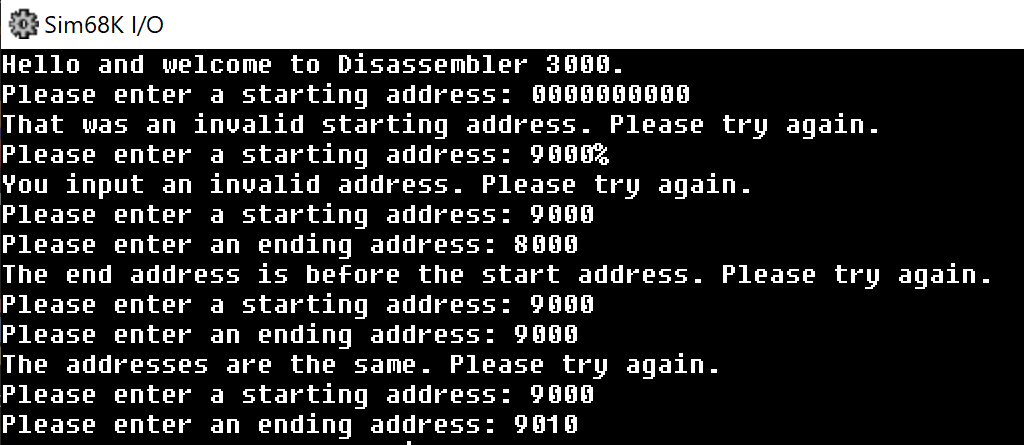
Upon successful read of an Op Code, Disassembler 3000 will output a similar response to what is shown in the screenshot below. The code to the right of the address in the highlighted instruction of the screenshot is read in and the proper output that should be read is on the left. The machine reads 0601 006F and will return ADDI.B #111, D1.



# Implementation

The implementation section is broken into four parts; User interface, Op Code determination and decoding, Effective Address Calculation, and a fourth section that we will call Print Functionality.

## User Interface

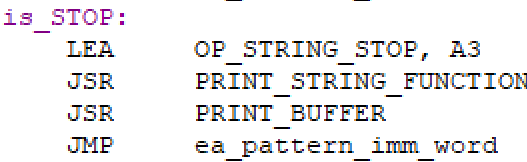
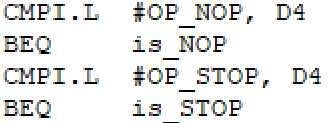
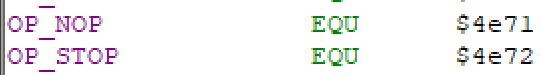
The user interface first displays the welcome message, then asks for the user to input a starting address. This address corresponds to the ORG address found in the listing file of the test file. The address that is entered is then tested for its length. If it is greater than or equal to 9 digits or if there was no input, the machine routes to an error handler that will give an error message informing the user they have entered an invalid start address. If a proper address is input, the machine will then move to an ASCII to HEX conversion function. This function checks the input one character at a time and converts it to hex by testing comparing it to known ascii to hex conversion values. If the user input a character outside of these values, they will be routed to the error handler and given an error message. Once this is completed, the machine moves forward to receive an end address. The functionality is essentially the same, with the exception of a few additional error handlers for ending addresses that are equal to the start, lower than the start, or invalid in any other way. Once valid input is received the machine will continue.

## Print Functionality

The print functions are separate functions created to make logic easier to implement. The majority of our printing is done by the creation of arrays. The idea came from our background in Java and C++ and we looked up how to implement in Easy68 in the easy68K Forum. The link to where we got the logic from is here: <http://easy68k.com/EASy68Kforum/viewtopic.php?t=283>. We call the array and have them organized in a way that a value that is parsed out from the full memory code will be used to roll through the array until the proper index is found. The other part of printing comes from loading the entire opcode name then calling a function to roll through and parse it out, loading it into the Decode Buffer (A1)+ using a helper function, as opposed to having each individual OpCode function do that.

## Op Code Decoding

The section starts by loading the number of window lines into a buffer that will decrease each time a new line prints to the screen. After moving through the print functionality to print the address, the instruction that is loaded from memory is then pieced apart to determine the name of the OpCode and all other pertinent information by performing a series of arithmetic ANDI calls and ROL or ROR to single out the piece of required information from the memory address. The rest of OpCode section is based entirely off an OpCode binary code to OpCode hex code conversion, essentially the hex code is compared to a library of pre converted codes called “OpCode Mask Library”. It will then move through 1 of 8 OpCode set checks. The first 7 checks have predetermined ANDI calls that help determine if the OpCode is in that section, the 8th is less predetermined and goes through a series of opcode checks that were unable to be grouped together with ease. Once the OpCode is determined, it then moves to its corresponding print function then to its corresponding EA pattern.



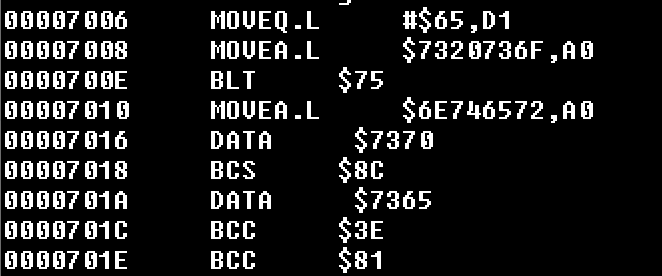
## Effective Address Calculation

The effective addressing section uses the fact that there are similar addressing patterns for certain OpCodes, which pattern is determined through the OpCode determination. There are some that have unique patterns which are also handled. The type of Effective addressing mode, size, register, displacement, condition, etc. was determined during the initial parsing of the data and loaded into a storage location. It will then go through a series of checks, based on the data placed into the storage, to determine which addressing functions need be called. It then calls upon the print functionality and prints to get the proper address display.

# Test Plan

Testing is relatively straight forward. We took each required and addition code and created a test file that tested each instance of it. We stepped through individual chunks of Op Codes with similar instructions and patterns before we moved on the next chunk. We also included op codes that are unsupported. Ultimately, we used the Test\_Disasm.X68 file provided by Professor Fukuda for many of the tests in the final version.

# Challenges

In general, the entire project proved to be very challenging, however, there were portions that we realized were issues once we were deep in throes of our program. Given our approach for checking for Op Codes using masks, we had a tedious effort involved in checking for each Op Code, even if it was unsupported. This led to issues in determining interspersed data and outputting it as DATA $WXYZ. As it stands, our program will output the DATA message correctly, but will also result in false positives when an Op Code should be defined as unsupported or invalid. For example, the output window on the left refers to the input on the right. Line 7004-7022 are interspersed DATA, but the program reads them as valid OpCode values. It will sometimes populate DATA when the OpCode cannot be deciphered at all, however, as seen in line 7016 in the screenshot above. We discovered very late in the testing and implementation that we had not fully covered ADDQ or SUBQ Op Codes, as a result, were unable to implement them before the final submission.

# Team Assignments

The majority of the work was done collaboratively. There were, however, two areas that were primarily implemented by one individual and not the other. Those are User Interface/Print Functionality and OpCode conversions. Below are the reports that were submitted, as well as a final statement from both parties.

## Progress Report #1

Progress

The project is starting to be pieced together. The UI is nearly completed, and the opcode is starting to be implemented

Johnathon designed the UI and began handling the addressing concerns of input

Blake created the constant variable library and created the ASCII to HEX functionality. Is in the process of testing it.

Plan

This weekend both parties will meet at Johnathon’s house to create the opcode table, begin plans for addressing, and create functions for opcode

Johnathon will continue working on addressing and finalize UI features such as print statements

Blake will begin creation of the opcode and creating the opcode table

## Progress Report #2

Progress

The project is starting to be pieced together. The UI is nearly completed, and the opcode is starting to be implemented

Johnathon designed the UI and began handling the addressing concerns of input

Blake created the constant variable library and created the ASCII to HEX functionality. Is in the process of testing it.

Plan

This weekend both parties will meet at Johnathon’s house to create the opcode table, begin plans for addressing, and create functions for opcode

Johnathon will continue working on addressing and finalize UI features such as print statements

Blake will begin creation of the opcode and creating the opcode table

## Progress Report #3

Progress

This past weekend both parties met at Johnathon’s home and worked on the program together.

Johnathon finished what was left of the UI, adding a Pause/Continue feature to allow for input to not be constantly required. He also finished what was left of some OpCode instructions

Blake finished creation of the OpCode Tables and instructions for Code values. He then began working on the EA table and address types.

Plan

During the next leg, both parties will work on the same section of code. Primarily EA, which is hoped to be done by Saturday where the hope is to begin the debugging process.

Johnathon will be finalizing all remain opcode instructions and begin working on any EA types that have not been completed

Blake will continue working on EA Types and start creating test files.

## Progress Report #4

Progress

This past weekend both parties met at Johnathon’s home and worked on the program together, putting together a nearly fully functional dissassembler.

Johnathon finished the Effective Addressing types and integrated it to the print functions.

Blake finished integrating the OpCode to their corresponding Effective Addressing patterns, and the method of parsing to get determine pertinent information.

Together they created a test file that tested the functionality of the entire code as it was currently written.

Plan

During the final leg of the project both parties will be working on debugging, any last-minute changes, commenting and creation of the final report.

## Unlisted Progress Report

Pair programming was used for a majority of the Effective addressing section of the code, with neither individual taking a majority of the work or any one section in particular. They both worked on the same section and would trade off to look at each other’s work, bringing new perspective when roadblocks were hit.

## Final Words

### Scott Blake

My main solo contribution was converting the values of the M68KOpcodes-v2.3 page to hexadecimal. A majority of this was done through online converters, but some were verified through methods taught in this class. I realized after looking the test file that the code in memory would create the output shown on the right of it, I was basically able to work backwards to convert and realize what pieces to parse out to get the next set of information. I also defined all of the OpCode constant variables.

### Johnathan Hewit

My main solo contribution was establishing the UI portion of the program, including input validation checking, converting from ASCII to Hexadecimal, forming the error messages, and printing messages to the console window. I also came up with the logic for handling the output buffer, including the output of converted instructions for the Op Code, Operands, and Address information, as well as the output of error messages and user prompts.