**Disassembler Final Report**

Team Visual Basic

CSS 422 Winter 2018

Austin Abeyta, Dylan Desmond, and Henry Hong

March 16, 2018

**Table of Contents**

[**Project Description**](#_pww4g77mieil) **3**

[**Program Specification**](#_xe3wliypbwgy) **10**

[Using the Program](#_1pc9zo6yox2e) 10

[Code Standards](#_ur1fhcg7cgvz) 11

[**Test Plan**](#_g9eokqv9ca1z) **13**

[**Exceptions Report**](#_8egeowiadi1n) **15**

[**Task Breakdown**](#_1r80dpb9hvlu) **16**

[Schedule Breakdown](#_otn1gn2q6yr3) 16

[Work Splits](#_i9dmlhj6sb89) 19

# **Project Description**

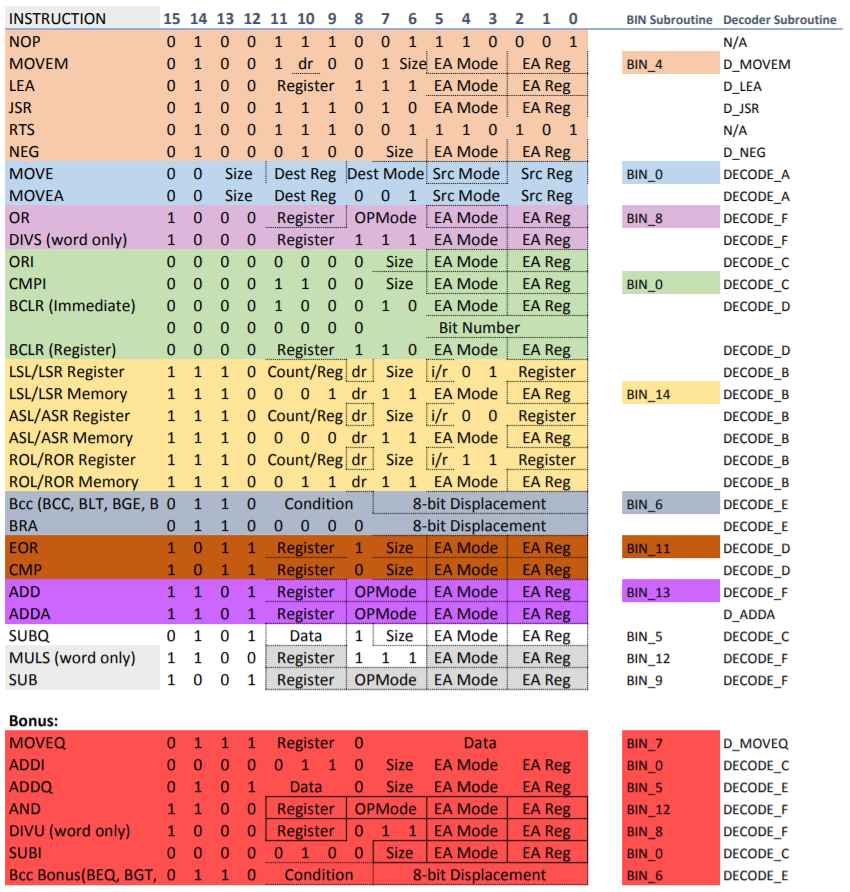
The disassembler is a program written in Motorola 68000 microprocessor assembly language. It takes in a machine code hex input and converts it back into readable assembly code. Our program is structured with the main driver file, named Abeyta\_Desmond\_Hong\_Disassembler.X68, handling printing a greeting message to the user. It then loops continuously, reading in one OPCODE worth of data at a time from the input data. It calls the OutputBuilder, which accumulates information into a string. The OutputBuilder calls upon the JumpTable file to determine what OPCODE it is and assemble it into the string. The JumpTable then calls the OP\_Decoder file to assemble the effective addressing information into the string. The main program continually loops, printing out one OPCODE at a time, until all OPCODES up until the specified end address have been decoded and printed.

In the jump table, the information passed from the main disassembler file is taken, moved into another data register, and broken down so that only its first 4 bits remain. Then based on the value of those first four bits, the program jumps to the subroutine of possible OPCODEs with the same pattern of bits in that location. Now that it has jumped into one of those subroutines, the specific subroutine checks certain specific constant bits in order to figure out which of the possible OPCODEs it is. If the bits passed in don’t fit with the possible OPCODEs in that subroutine, the program branches to our DATA subroutine, otherwise it branches to the subroutine for the specific OPCODE we have found it to be. In each specific OPCODE subroutine, the subroutine accumulates OPCODE name in ASCII into a string located at A4 for printing, and depending on the OPCODE either adds and appends the size if it is constant, or jumps to the correct size finding subroutine. The size finding subroutines check the important bits to determine if it is a byte, word, long or data. If it is data it branches to the data subroutine, otherwise it jumps to the correct size subroutine where it appends the the size by pushing it into A4 and adds the size into A5 for incrementing after. From here it uses RTS to jump back to the main subroutine for that specific OPCODE, and depending on the OPCODE jumps into the correct subroutine in our decoder file for decoding and printing the rest of the OPCODE information.

Now that it is in the correct decoder subroutine, the opcode is broken down based on its structure in order to determine when applicable: the EA mode and register, source, destination, and any other important information. All of this information is accumulated into A4 and finally uses RTS to jump back from the decoder into the main Disassembler file where it is printed from A4.

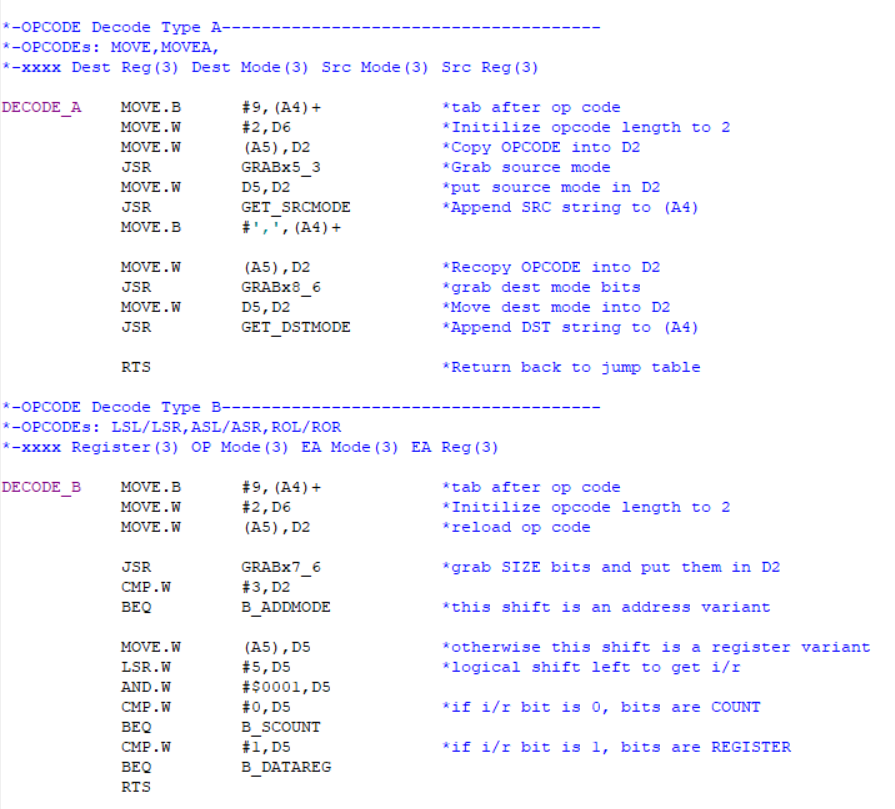
The table included below is a design document we created early on to get an idea of which opcodes we were going to add, and also how we could group them to re-use as much code as possible. By doing it this way, we were able to make subroutines in our jump table and decoder files that could handle multiple different opcodes. And because we were able to handle multiple different opcodes in each subroutine, we decided to do some extra opcodes that were not originally required by the spec. These extra opcodes include: **MOVEQ, ADDI, DIVU, SUBI, and the Bcc variants BEQ, BGT, BLS, BNE, BHI, BVS, BPL, BMI, and BLE.**

**Table 1:** Opcode command excel spreadsheet

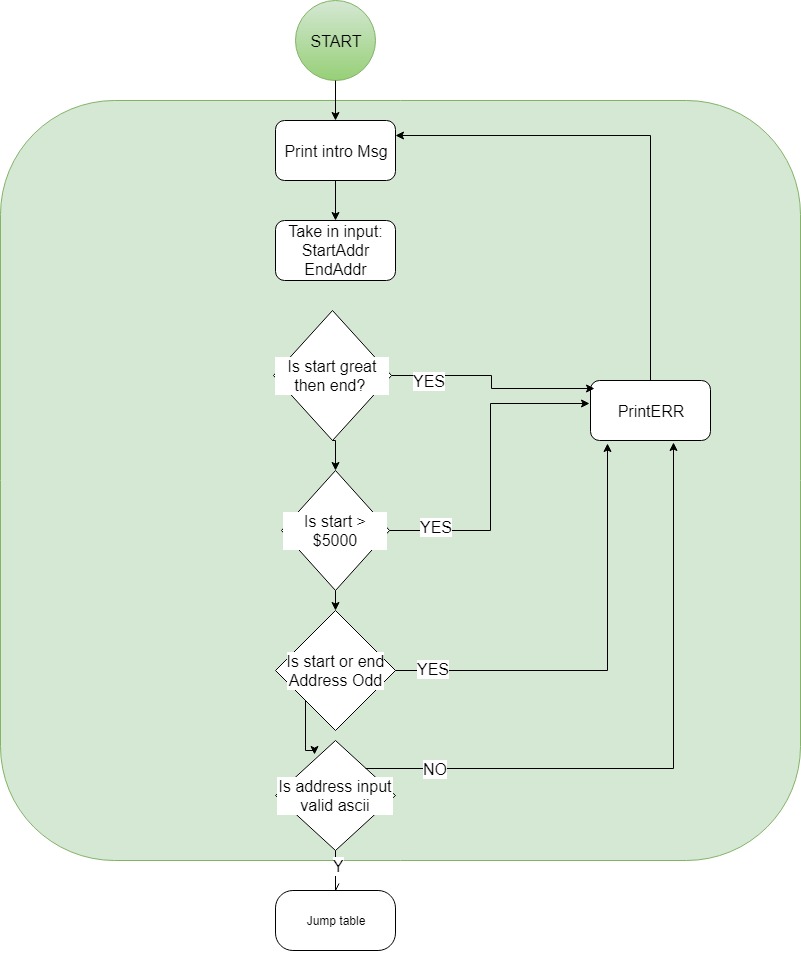


The figure included below is an example of code from our decoder file that we felt was both clever and difficult to do. We felt that this code was clever because it showcases the way that we structured our subroutines in the jump table and decoder files. Each subroutine is able to handle multiple different opcodes as long as they have the same structure. This gives us the option to then take what we have, and grow it to include many more opcodes than we currently do, as long as they fall into the structure type of one of the subroutines that we have already created. However, because of this the code ended up being quite difficult to write. Not only was it complex and conceptually difficult, but when bugs did happen it could be hard to trace the origin and fix them.

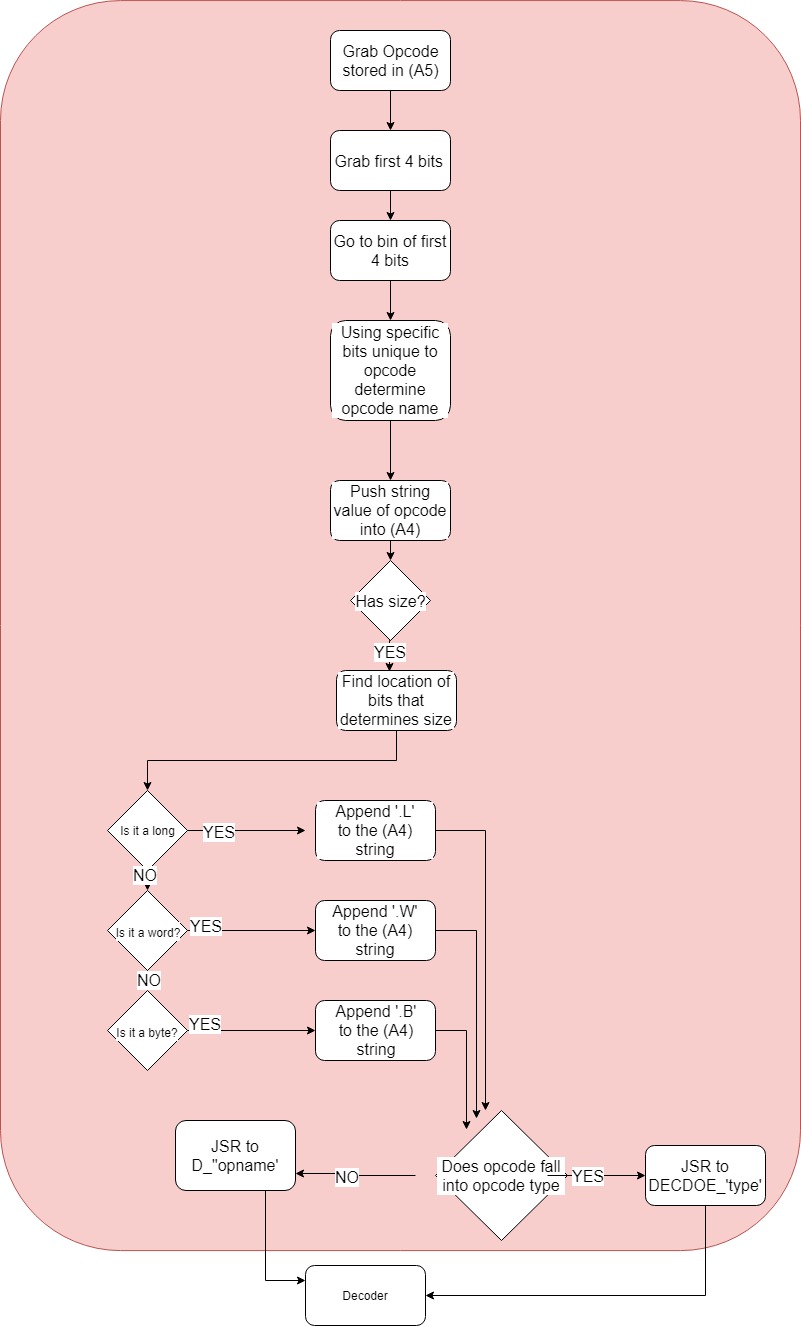
**Figure 1:** Code Snippet from “TeamVisualBasic\_OP\_Decoder.X68”



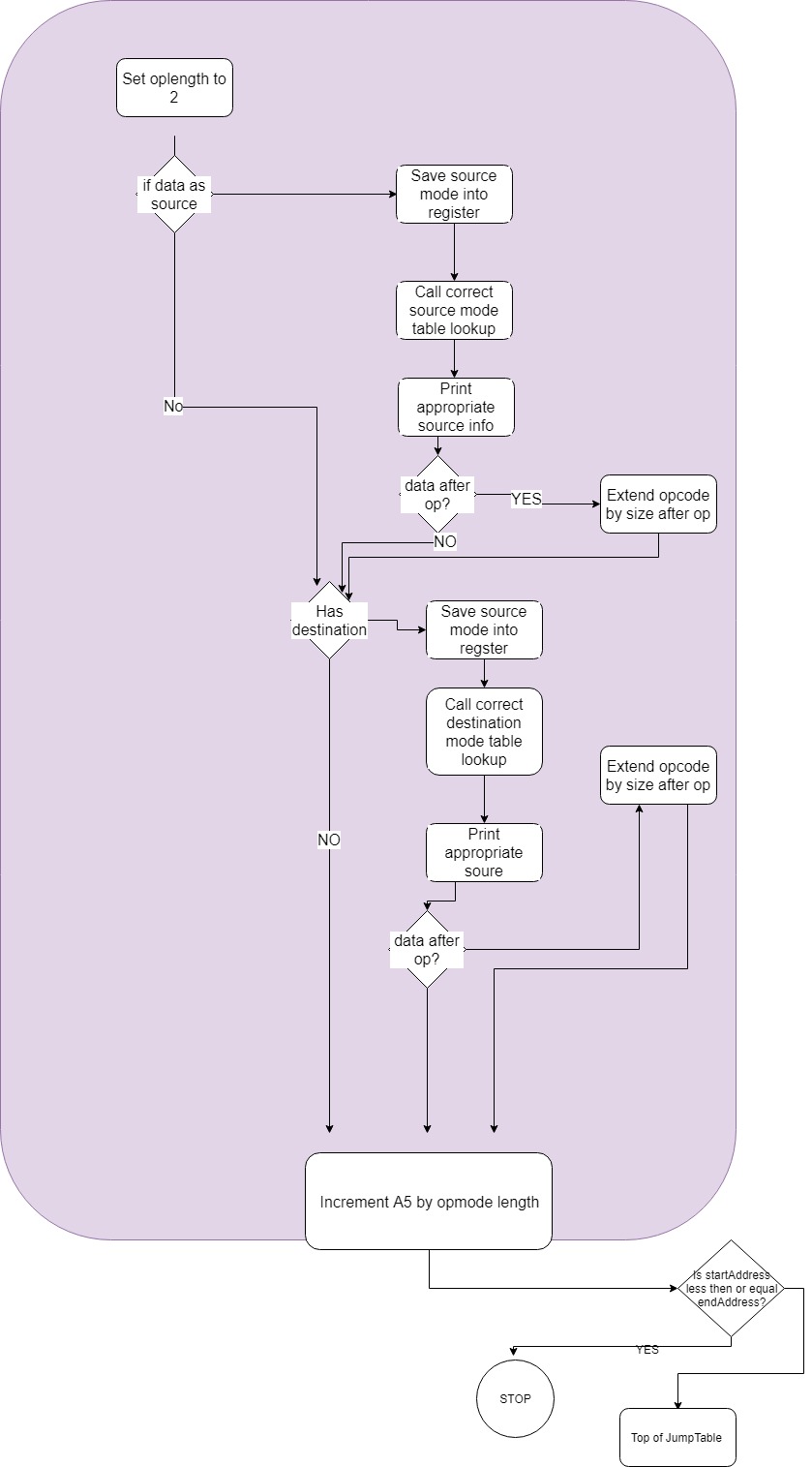
**Figure 2-a:** Flowchart (input)

****

**Figure 2-b:** Flowchart (Jump Table)

****

**Figure 2-c:** Flowchart (OP Decode)



# Program Specification

## Using the Program

The disassembler can be used in the following way:

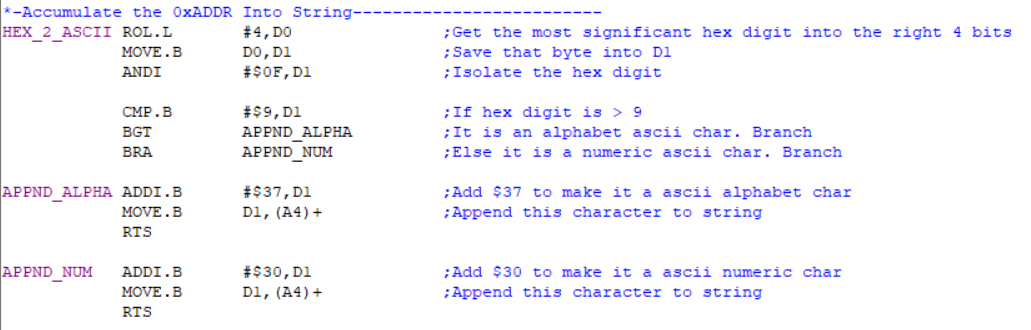
First, open up the file that will be used for testing and run it. After you have run the testing file open up the main disassembler file titled “Abeyta\_Desmond\_Hong\_Disassembler”. Once in that file click to run it. When it opens before executing, click on “file” and “open data”. Once there, select the .S68 file for the testing file as this is the file broken down into binary. Then execute the file. It will prompt you for a starting address. For testing we have set the starting address to be 7F00. After that you will be prompted for an ending address. This address must be larger than the starting address, and both have to be non-negative and even addresses. If the given address does not follow these conventions an error will be thrown and you will be asked for a new one. Otherwise the program will execute and print out the opcodes included in the test file up through the end address.

Our program takes opcodes and data that have been broken down into binary, and starting at the given address, parses through it figuring out what sections of bits are an opcode, and what sections of bits are random data. When bits are found that seem to be random data, the program prints out DATA followed by the hex value of the bits we have found to be DATA. Otherwise, when a section of bits is found that looks like it could be an opcode, we look through those bits to determine if it is a valid version of one of the supported opcodes, and then if it is, we figure out which one. Once the opcode has been determined, we print it out along with its size and extra information when applicable. After it has been printed, we continue parsing through until all of the opcodes up to the end address have been printed. As it will only print a page at a time, more pages of printed opcodes can be seen by hitting the enter button to move to the next page.

## Code Standards

To make the code easy to read, while writing our program we kept to pretty strict coding standards. We included function headers at the beginning of each important subroutine to describe the opcodes used in that function, commented almost every line except when we felt the code was pretty self explanatory, and kept to a method of tabbing so that each program looked the same and was easy to read. Furthermore, in order to make it easier to understand while we were coding we also had guidelines for the function naming in certain key areas. In the jump table we names all of the subroutine bins with “BIN\_” followed by the decimal value for their first 4 bits (“1-14”). In the decoder we named each subroutine with “DECODE\_” and the letter we associated to a certain opcode type for those that were similar, or “D\_” followed by the opcode in all caps if it was a unique structure that we were decoding.

**Figure 3:** Code Snippet from “TeamVisualBasic\_Disassembler\_OutputBuilder.X68”



As we did not end up using a callee-saving method for our subroutines, we had to be extremely careful about what we were saving in each register, and making sure we weren’t overwriting any information that we needed. To do this we decided ahead of time what each register we needed would be used for so that there wouldn’t be any overlap. What we ended up deciding on was:

A1 - temporary/calculation

A2 - temporary/calculation

A3 - temporary/calculation

A4 - address register for printing opcodes

A5 - start register for opcodes

A6 - end register for opcodes

A7 - reserved by 68k for program counter

D1 - temporary/calculation

D2 - temporary/calculation

D3 - temporary/calculation

D4 - main\_loop counter variable

D5 - temporary/calculation

D6/D7 - save opcode and operand size here (how much to increment address pointer by)

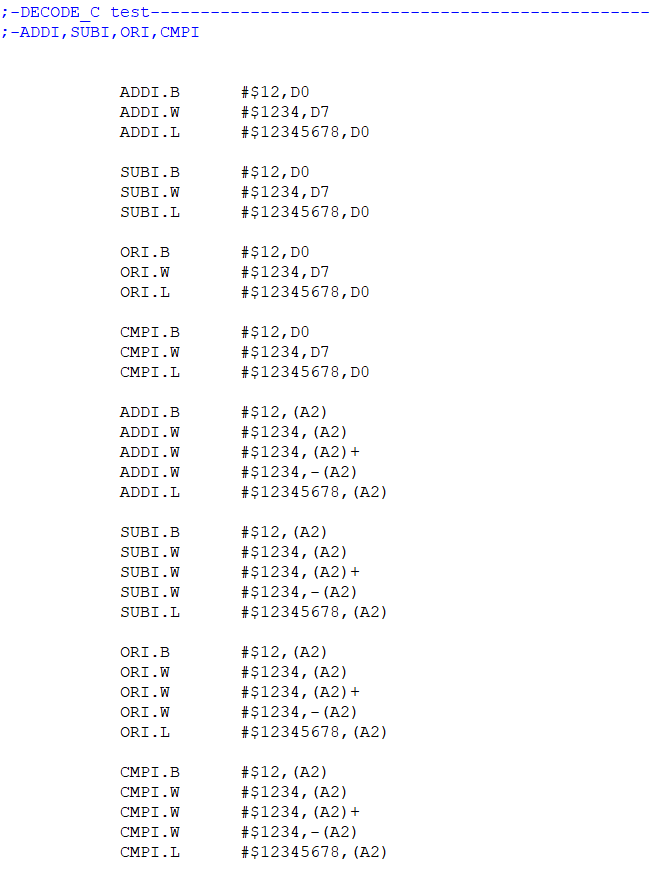
# Test Plan

Early on in the decoder process, our method of testing was a bit more free form. Each time a subroutine of opcodes was finished, we would set a data register at the beginning of the file to be a valid hex value for an opcode in that subroutine, and then make sure that the correct opcode was printed. Also before the jump table and decoder file were both at the point where they could be connected, we also created the testing file “TeamVisualBasic\_OP\_Decoder\_TEST.X68” with a mock printer and mock main for testing incrementing across addresses after an opcode is printed.

After this initial methods of testing, our main method of testing is the file called “TeamVisualBasic\_TestingData” (see figure 3 below) and continually updated it as we went. In this file, each time we completed one of the subroutines in the decoder we would add a section in the testing file for the opcodes in that subroutine. Typically, we started with one type of EA mode and keep testing those until all of the opcodes in that subroutine were printing correctly in that EA mode. Then once they were, we would add statements for another EA mode, adding on to whatever we already had. This would continue until all of the EA modes were printing correctly for the opcodes in that subroutine. Once they were, we would move on to the next subroutine and begin testing.

For actually running these tests, we would run the testing file in order to get the .S68 file for it. Then in the main disassembler file, we would load that .S68 file as data. In the main disassembler file, we used $7F00 as our start address for testing, and our end address depended on how many opcodes we were testing at that time. The reason we chose to do $7F00 for the majority of our testing was to ensure that decoding addresses past the 16 bit addressable range ($7FFF), would still work. However, we did do some testing at other memory locations towards the end. At the end we removed that part of the range and pointed our decoder file at itself to see if we would find any interesting bugs. When we did that we noticed that the TRAP tasks would throw off the incrementing of our program causing the opcodes after that to print out incorrectly, and also that when our program isn’t sure what the opcode it is getting is, but thinks that it is still an opcode and not data, it prints that opcode out as ORI.

**Figure 4:** Code Snippet from “TeamVisualBasic\_TestingData.X68”



# 

# 

# Exceptions Report

Although we were very close to finishing everything, towards the end we ran into some bugs that we were not able to fix by the time the project was due. These bugs involved the opcodes MOVEM, and BCLR immediate version. On MOVEM we had a lot of trouble due to the ranges. Because of this we opted to make sure that the size is found and the program increments by the correct amount afterwards, in order to ensure that it doesn’t crash the program or cause the rest of the printing to no longer work. So currently, MOVEM will not crash our program or ruin the printing as far as we can tell, and print out no register list or effective addressing. BCLR Immediate was not printing effective addressing entirely correctly, and we could not address those issues in time.

One odd behavior noticed when looping the program multiple times was that the starting address input cursor would appear before some printed text, causing that input to not be valid. Continuing through and inputting the address again would bypass the issue.

# Task Breakdown

After our initial meetings to decide on group norms, the meeting schedule, and how we would be collaborating, we broke up the team into two main groups. Henry worked primarily on the main Disassembler file, and input / output files, while Austin and Dylan worked on decoding the OPCODEs.

## Schedule Breakdown

Here are some of the main tasks that we completed in the first two weeks:

Henry:

* Making an intro screen
* Take user input for memory start address and end address
* Input sanitization and verification
* Set up the structure for the main disassembler file
* Set up the structure for the Jump Table file
* Established coding + commenting conventions

Dylan:

* Created excel sheet of commands to decode
* Wrote code to:
  + Decode MOVE
  + Decode MOVEA
  + Decode BRA
  + Decode CMPI
  + Decode MOVEM
  + Decode LEA
  + Decode NEG

Austin

* Code to sort the commands by their first 4 bits
* Wrote code to:
  + Decode NOP
  + Decode RTS
  + Decode BCLR
  + Decode JSR
  + Decode ORI
  + Decode DIVS
  + Decode LSL/LSR
  + Decode ASL/ASR
  + Decode ROL/ROR

After the first 2 weeks we felt like we were behind where we should be, and redoubled our time in an attempt to catch up. In the next week we completed:

Henry:

* Completed the input file
* Completed the main Disassembler file with loop code
* Created and completed an OutputBuilder that accumulates the opcode address with the JumpTable + OP\_Decoder output
* Restructured Jump Table organization + added basic implementation to all ops
* Implemented several Decode functions

Dylan:

* Wrote the code to:
  + Decode SUBQ
  + Decode Bcc
    - BCC
    - BCS
    - BGE
    - BLT
    - BVC
  + Decode SUB
  + Decode EOR
  + Decode CMP
  + Decode MULS
  + Decode ADD
  + Decode ADDA

Austin:

* Sorted the commands based on structure
* Started a decoder file for figuring out the information to print after the OPCODE and size
* Broke up the decoder into sections for OPCODE structure in order to reuse as much code as possible
* Initial coding for each of the sections for finding the correct information to print
* Implemented all source/dest immediate data making sure the opcode length was correct
* Did all post pre increment for address registers

At this point we knew we only had a week left and that there was still a lot of work to do. As Henry was already finished with all of the input and output work, he helped Austin finish the OPCODE information decoding and printing section as it was the most complicated and difficult thing we still needed to do.

In our final week we completed these tasks:

Henry:

* Bug fixing Jump Table
* More implementation in the OP\_Decoder, include DECODE\_C, and one-off OPCODES that don’t fit into categories, such as D\_MOVEQ, and D\_MOVEM
* Bug fixing on the decoder file
* Documentation revision

Dylan:

* Wrote subroutines to append the size to the OPCODE and store the size in a register for incrementing after
* Wrote code to decode the extra OPCODEs:
  + MOVEQ
  + ADDI
  + ADDQ
  + AND
  + DIVU
  + SUBI
  + Bcc codes
    - BEQ
    - BGT
    - BLS
    - BNE
    - BHI
    - BVS
    - BPL
    - BMI
    - BLE
* Initial documentation write up and structure

Austin:

* Worked on implementing the rest of the subroutines in the decoder file
* Bug fixing in the decoder file, after github upstream issue
* Documentation revision

Throughout the project we tried to meet 2-3 times a week in person, usually thursdays and saturdays but sometimes on tuesdays as well. During these meetings we spent our time getting on the same page, deciding on the tasks we needed to complete by the next time we would meet, and pair program difficult parts of the coding together. We also worked together on testing all of the elements as we created them, and creating a large test file at the end to comprehensively test our system.

## Work Splits

Henry 33%

Austin 33%

Dylan 33%

Despite the percentage skew being uneven earlier on in the project, throughout the process we were all involved in the planning process and overall project structure planning. In the final week, we ran into issues that hinder our progress severely. One of our project members lost a significant amount of work due to GitHub upstream issues, and unexpected last minute bugs rendered us unable to finish MOVEM in time.We reloaded an older version and then incorporated snippets from new versions to troubleshoot bugs. However, we all pulled together and finished the project through a lot of pair programming and team effort.