The Three Stooges

CSS 422 Hardware

Professor Wooyung Kim

Authors: Christopher Dubois, Brody Schulke, Rob Stachofsky

**Disassembler**

1. **Program Description**

Flowchart shown in Appendix B.

Our design philosophy was to take this project one small piece at a time. We first started by reading in and validating I/O user starting and ending addresses. Once we had fully tested and developed this portion of this code, we started working on moving through memory, printing out the address and data. At this point we started decoding word instructions in memory, one opcode at a time, also fully testing this opcode before moving on to the next step. This was to ensure program integrity in case we wanted to reuse methods down the line. Then we started developing the <ea> portion of the disassembler. We took the <ea> decoding process one step at a time and fully tested the opcode family using various <ea> combinations. Once we finished one family, we moved on to another, attempting to re-use as many algorithms as possible and decoding various <ea> as required. If we ran into a problem in the decoding of the <ea> we mapped out the possible combinations of branching, looking at each bits in the family to determine what to branch to, and bad data otherwise. We made sure that we only decoded valid opcodes and instructions inside of an opcode family. We did this by comparing the word formats between all possible opcodes within specific families. We took the opcode decoding in order of what we expected to be easy first and what we thought was most difficult last. After the program was complete, we developed a comprehensive test program and a quick comparison method for output vs. expected output.

We were able to reuse our HEX\_TO\_ASCII with every opcode decode. Also, in general, we were able to reuse a fair amount of subroutines for various opcodes. We are also proud of our use of jump tables to reduce a huge amount of comparisons.

All algorithms were developed inside of The Three Stooges. We looked to online sources for some test data and appropriated tests for MOVE, MOVEA and ADDA. The rest of the test instructions were written by The Three Stooges.

To our knowledge there are no opcode decoding limitations that cause the program to crash or incorrectly decode *unless* invalid test data is longer than 1 word in instruction length. Additionally, when a user inputs an address, it must be capitalized, an even number, and must start at $7000 or greater and cannot end past $FFFE. When prompted to restart, quit or continue, the user must use capital letters.

1. **Specification**

Our program decodes the following instructions in the 68k Assembly instruction set:

\*0000 SUBI ORI EORI BTST CMPI

\*0001 MOVE.B

\*0010 MOVEA.L MOVE.L

\*0011 MOVEA.W MOVE.W

\*0100 MOVEM LEA NEG NOT JSR RTS

\*0101 ADDQ

\*0110 BEQ BNE BLT BHI BRA

\*0111

\*1000 DIVS

\*1001 SUB SUBA

\*1010

\*1011 EOR CMP CMPA

\*1100 MULS AND

\*1101 ADD ADDA

\*1110 LSD ASD ROD

\*1111

Our program will print out bad data for opcodes that are not matching the above, (for the cases 0111, 1010, and 1111) as well as for cases where the effective addressing does not match with the format of the above instructions. If the first 4 bits match, but there are discrepancies amongst the other bits and we are able to distinguish the instruction to be an invalid opcode, we also print bad data.

1. Reads in a starting and ending address from user input as hexadecimal
   1. Verifies the validity of the addresses
2. Goes from start to end, printing 15 instructions at a time
   1. (printing the actual instruction or DATA if could not be decoded)
   2. Allows user to continue, restart, or quit any time after a set has been printed.
   3. It does not read the opcode at the specified ending address, that is, the disassembler excludes the ending address and stops before decoding once it reaches the specified user ending address.
3. Prints a notification message once the disassembler has completed
   1. Allows the user to restart or quit
4. **Test Plan**

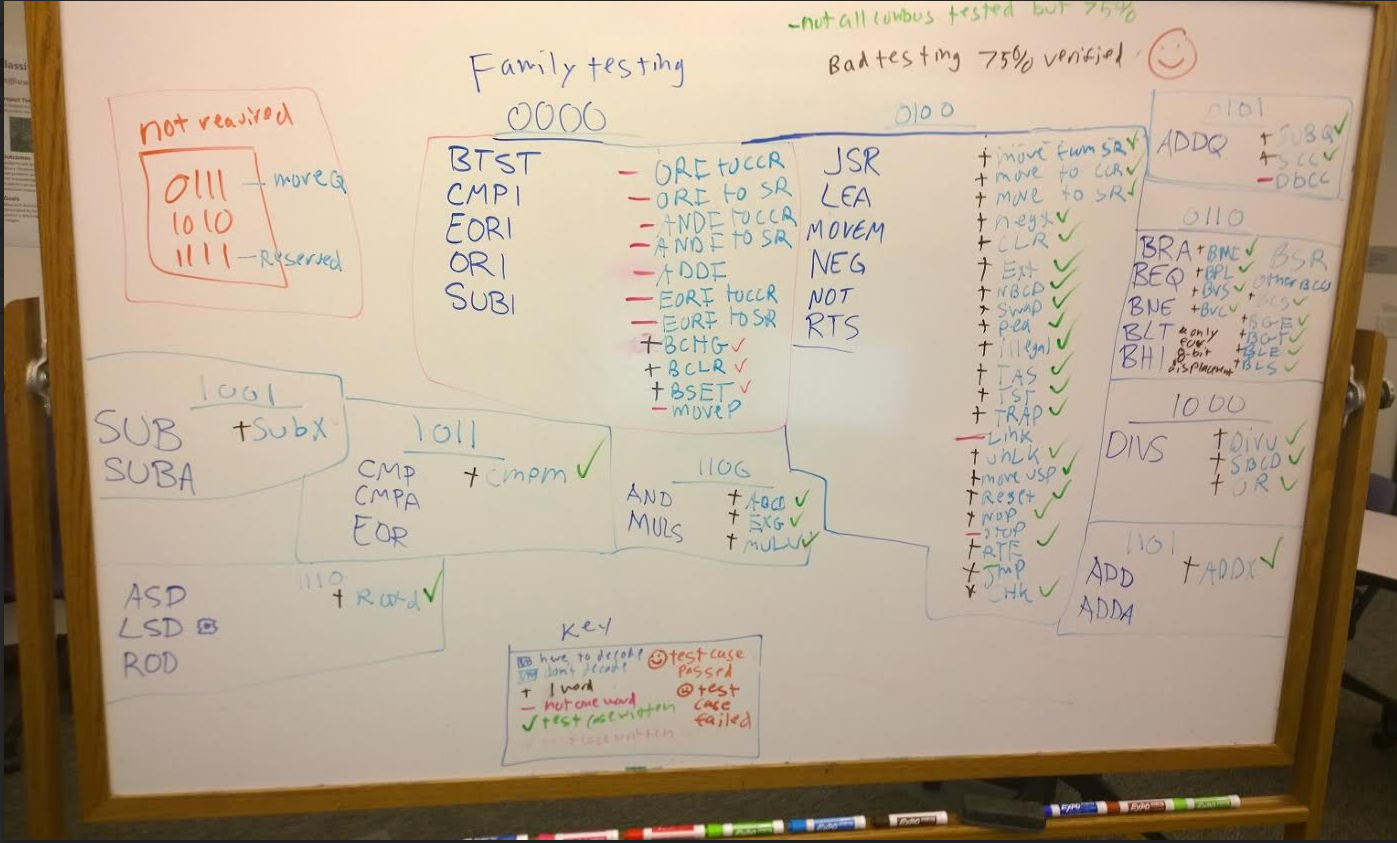
Testing files included. (3) (1-Good test data, 2-Bad test data, 3-combined)

Java code for parsing out particular data from the L68 of our testing file included (explained below).

Use of <http://www.mergely.com/> (explained below).

We wrote our disassembler one instruction at a time, and fully tested all of the possible combinations of instructions before we moved on to the next instruction to decode. This allowed us to ensure that method reuse would be on a solid foundation and bugs would not be compounded. At the very end of developing our disassembler, we compiled all of these tests into one .X68 file and ran the .L68 output of it through a very basic, pared down java parser that we could then run through a “difference checker” against our console output for quick verification of our code. Essentially, the parser would cut out the unneeded details from the .L68 file and place the new data into a formatted text file. We then ran our disassembler on the test data and logged the console output. After we had both files, we placed the text in to the “diff-checker”, using mergely.com, and found that no errors were produced after doing some debugging. This sped up the testing process by a large factor.

To verify our bad data testing, we went through the instruction set manual and our required opcodes and reproduced valid test cases within an opcode family that we were meant to decode. It’s mapped out in the picture below:



While we didn’t test every single combination of effective addressing across bad opcodes, we found it wasn’t necessary because the effective address wasn’t a determinant factor as to whether or not it was an invalid opcode (or if we just print bad data). This is why the 75% verified is listed in the upper right hand corner of the whiteboard.

1. **Exception report**

To the best of our knowledge, our program decodes all of the required opcodes as desired. This is based on the assumption that any opcode that comes in as test data that is also bad data takes up only one word of space, as in the project specifications.

For entering starting and ending addresses our program requires that you enter **FOUR HEX CHARACTERS. LETTERS ALSO MUST BE CAPITALIZED. THE ADDRESSES MUST ALSO BE EVEN.**

Also, we noticed a small bug when using the “Rewind” feature of the .S68. For some reason, that we can’t figure out, occasionally our program will behave incorrectly after using the “Rewind” option. We suspect that it may have something to do with address registers/memory.

1. **Team assignments and report**

**Individual percentages**

Christopher DuBois – 33.333%

Brody Schulke – 33.333%

Rob Stachofsky – 33.333%

**Team Task Organization**

The Three Stooges developed the disassembler by coding as a group of three, that is, 0% of our time was spent on the project as individuals. Every time we met, one person would type code on the computer (rotating) while the other two would verify the code, suggest idea, and watch for errors. We essentially operated as a 3-person human IDE. When certain “external” tasks such as testing, debugging, or mapping out difficult opcodes came up, we would split off a group member to quickly discover a solution or complete the work. Once the group member finished with the task he would merge back into the group workflow. If the group member that split off needed assistance, then always *both* other group members would assist him with the task at hand. Everybody in the group did an equal amount of everything, but each person spent 1-2% more in certain tasks.

1. **Appendix** 
   1. **Java Code – Basic java code that parses out some bad details from the .L68 of the testing file**
      1. **Code:**

package delcom;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileWriter;

import java.io.IOException;

import java.io.FileReader;

import java.io.BufferedReader;

import java.io.InputStream;

import java.util.ArrayList;

import java.util.Scanner;

public class delcom {

//this would have to be change

private static String filePathway = "C:\\Users\\chrisad\\Desktop\\inputfile\\COMPLETETEST.L68";

//private static String filePathway = "C:\\Users\\chrisad\\Desktop\\EASy68kOLog.txt";

public static void main(String[] args) throws IOException {

parseLinesFromFile(filePathway);

}

public static void parseLinesFromFile(String filePath) throws IOException {

Scanner fileIn = null;

try {

fileIn = new Scanner(new FileInputStream(filePathway));

}

catch (FileNotFoundException e) {

System.out.println("Can't open the file");

System.exit(-1);

}

//this would have to be changed

//File fileToWriteTo = new File("C:\\Users\\chrisad\\Desktop\\outputfile\\outputForDiff.txt");

File fileToWriteTo = new File("C:\\Users\\chrisad\\Desktop\\outputfile\\outputForDiff3.txt");

try {

if ((fileToWriteTo.exists())) {

fileToWriteTo.delete();

fileToWriteTo.createNewFile();

}

}

catch (Exception e) {

System.err.println("File could not be created.");

System.exit(-1);

}

FileWriter fw = new FileWriter(fileToWriteTo, true);

int lineCount = 0;

while (fileIn.hasNextLine()) {

String inputLine = fileIn.nextLine();

while (   inputLine.length() > 0 && (inputLine.contains("\*") || inputLine.contains(";") )) {

if(fileIn.hasNextLine()) {

inputLine = fileIn.nextLine();

}

}

if (lineCount > 8) {

try {

StringBuilder sb = new StringBuilder();

if (inputLine.length() > 40) {//PRE: the L68 file has the proper tabs, POST:actual L68 information important is from 0-8 but only if there's stuff after [40] index - the instruction begins

sb.append(inputLine.substring(0,8));

sb.append(inputLine.substring(38,inputLine.length()));

String outputLine = sb.toString();

fw.write(outputLine);

fw.write("\n");

fw.flush();

//System.out.println(sb); // to test without looking at file

}

else {

continue;

}

}

catch (Exception e) {

continue;

}

}

lineCount++;

}

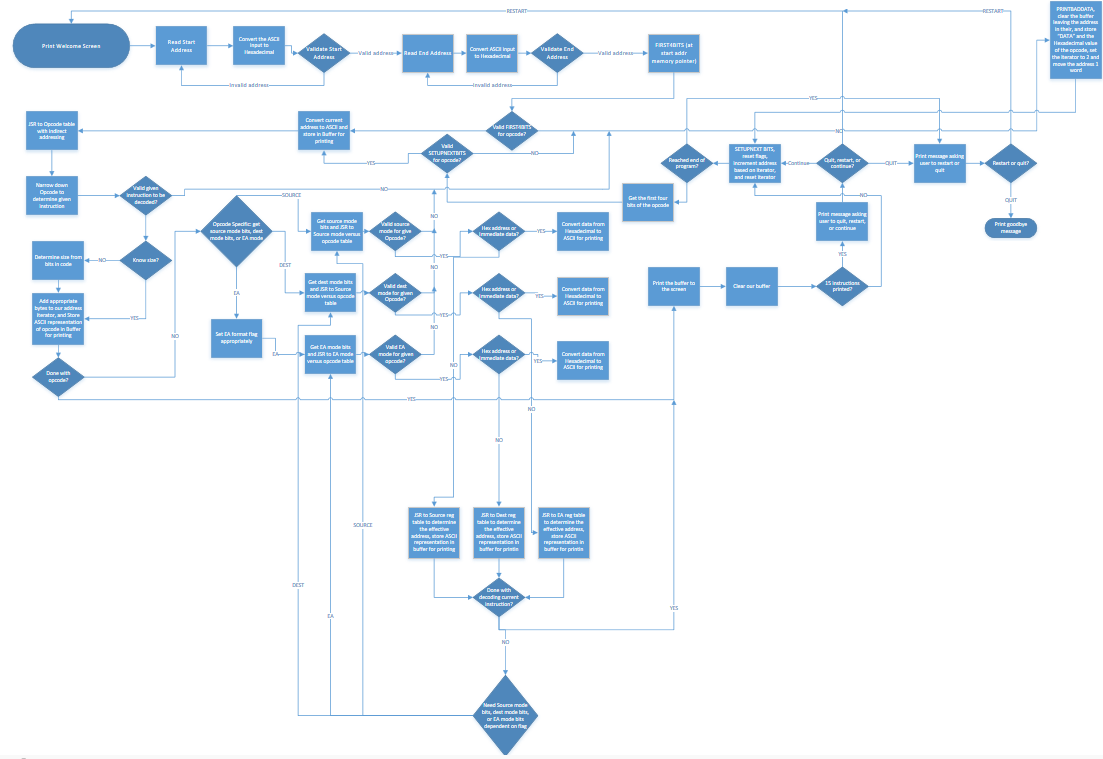
fw.close();

System.out.println("End of L68.");

System.exit(0);

}

}

* 1. **Flowchart**