Project Report

Team 69K | Michael Ward, Cole Liu, Tri Nguyen

Program description

Brief Description

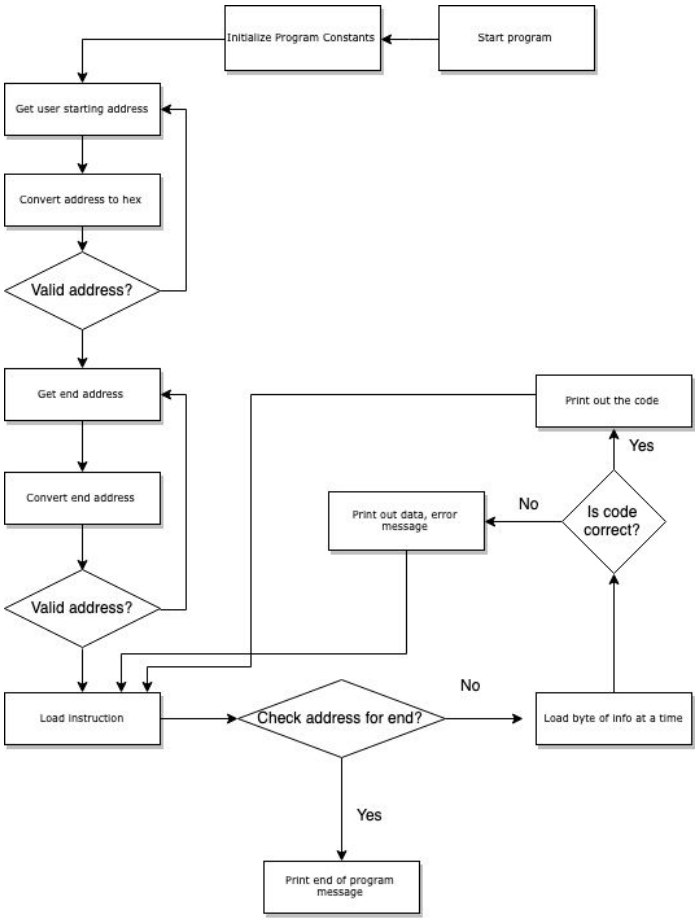
A disassembler is a computer program that read machine language and convert them into assembly language (opposite from assembler). There are more than one type of disassemblers; for instance, if the output language is C, the program is called a decompiler instead. Writing a disassembler is challenging due to its variety in assembly language. One disassembler can only read one assembly language. Our challenge also lies where different addresses are being used in the code.

Design Philosophy

Our design revolves around simplicity and efficiency. It consists of two main parts, validating the input from users and parse code. These two parts were built separately to ensure that there was no coupling between their functions. Nevertheless, there were some shared helper methods to reduce duplicate codes and enhance reusability. In addition, we implemented a jump table to enhance the efficiency while parsing the code. With all those practices combined, our program can ensure high usability as well as maintainability.

Flow Chart (Right)

The program starts by asking user to enter the beginning address of the code being read, as well as the ending address where the user wants the disassembler to stop reading. The program checks both inputs by convert them into hex and validates the range to make sure the disassembler go from a smaller address to the larger address. If the input is invalid, the program asks the user to re-enter those values. Afterward, the program will enter the

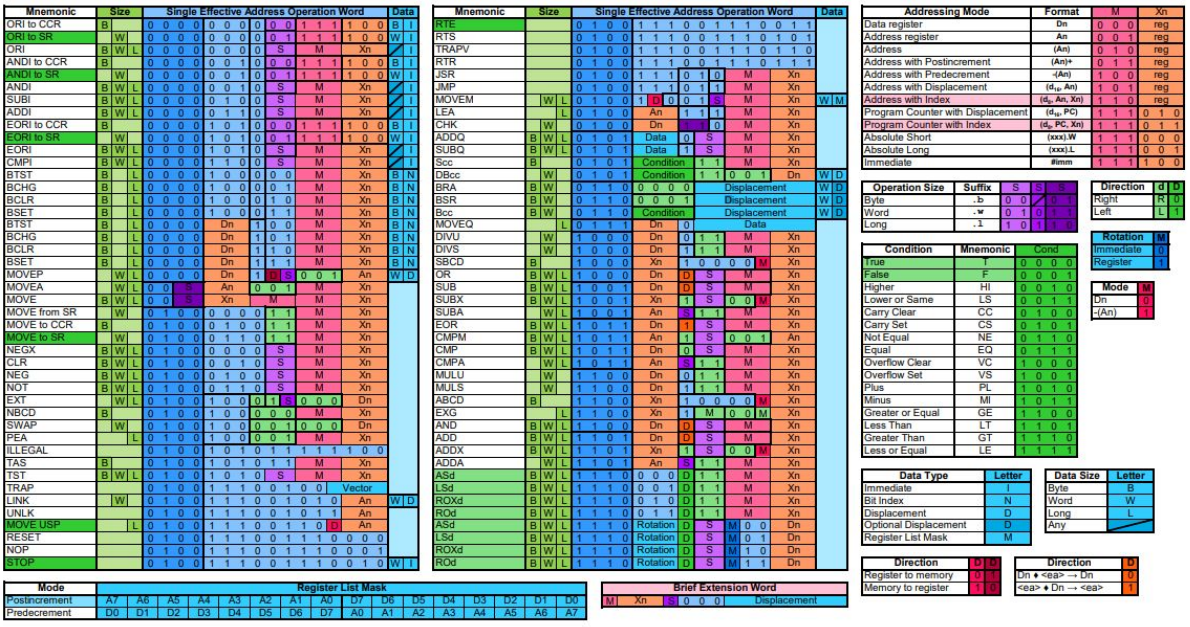


loop, calling method to start parsing opcodes. By starting off comparing with the jump table based on offset, our program is able to quickly determine which opcode matches the given instruction. In addition, this also cleans up the code and make it easier for all of us to debug and trace back. Finally, the program prints the disassembled code to the output. A full sized version of the flow chart can be found below in the appendix.

Algorithms that you were especially proud of,

MOVEM: we had problem to determining the size of the data due to either having mark list data and address or just mark list. This caused programs as our code would read the wrong line of opcodes and print DATA and mess up future codes. It was challenging, but we felt so relieved after fixing the problem by looking into the opcode sheet and breaking down the problem further.

Motorola 68000 CPU Opcodes



Version 2.3 by GoldenCrystal http://goldencrystal.free.fr/M68kOpcodes-v2.3.pdf

Algorithms that you got from other sources

We didn't use any algorithms from other sources, but we got some ideas about the layout of our program from some flow charts we found about this project on the internet. By going through it inspired us the layout of the program, then we were able to separate different tasks to everyone

in the group. Additionally, many online resources like StackOverflow and sites dedicated to 68K issues were used.

Specification

Specification Overview

Due to the complexity of this program, following the specification initially put to our team was highly important to developing the correct code. Below are the specifications of the project including a brief *Description* of our disassembler, *Limitations* of the code, *Language Restrictions, Technical Details,* and final *Deliverables*.

Description

Our team wrote an inverse assembler (disassembler) that converts a memory image of instructions and data back to 68000 assembly language and output the disassembled code to the display. During this project, we were only required to disassemble a subsection of all the opcode instructions and addressing modes that are for the 68K assembly language. The full list of these opcodes and addressing modes are included below. It was assumed that the input of this project would not be filled with wrong data but actual compiled code.

Limitations

As part of this project, we had limitations including not using TRAP function 60 in the simulator and develop and own our own disassembler. Due to these limitations, we worked hard to develop algorithms to solve the complex problems of breaking down the opcodes and addressing modes and did not use any TRAP functions past 14.

Language Restrictions

Your program should be written from the start in 68000 assembly language. Do not write it in C or C++ and then cross-compile it to 68000 code. It is really easy to tell when you've written it in C and it probably won't save you very much time. When you are working at the bit level C is just structured assembly language (or so they say).

Technical Details

It is important to note some critical technical components of our project. These include: the location of the start of the program, display message/user input, display screen restrictions, error handling, address displacements (BRA), and end user prompts which will be discussed below.

- Our program is ORG'ed at $1000.
- At startup, our program displays a welcome message and then prompts the user for the starting location and the ending location (in hexadecimal format) of the code to be disassembled while handling user inputted errors.
- The display shows one line of data at a time on the screen. The screen prints out 25 lines (unless going past the ending address or running out of data) allowing the user to hit the ENTER key to display the next screen of information.
- The program is able to recognize when it has an illegal instruction (i.e, data), and is able to deal with it until it can find instructions again to decode. Instructions that cannot be decoded, either because they do not disassemble as op codes, or because the program is not able to decode them are displayed as:

  1000 DATA $WXYZ

- Address displacements or offsets are properly displayed as the address of the branch and display that value. For example, our dissembler shows the following line correctly:

  1000 BRA 993 * Branch to address 993
- Our program completes a line by line disassembly displaying the following columns:
  - a- Memory location b- Op-code c- Operand
- Finally, when our program completes the disassembly process, the program prompts the user to disassemble another memory image, asks the user to exit the program.

Deliverables

By the end of June 7th, 2019, we will have delivered on canvas our final project including the source code, self grading sheet, project documentation, and Confidential evaluation report.

Test Plan

Project Tests and Coding Standards

In order to have functioning code, we determined that setting up a framework around tests and coding standards were necessary for collaboration. This report describes the effort we took for testing our software and the coding standards we used. Finally, it also documents some of the testing files we used.

Project Report

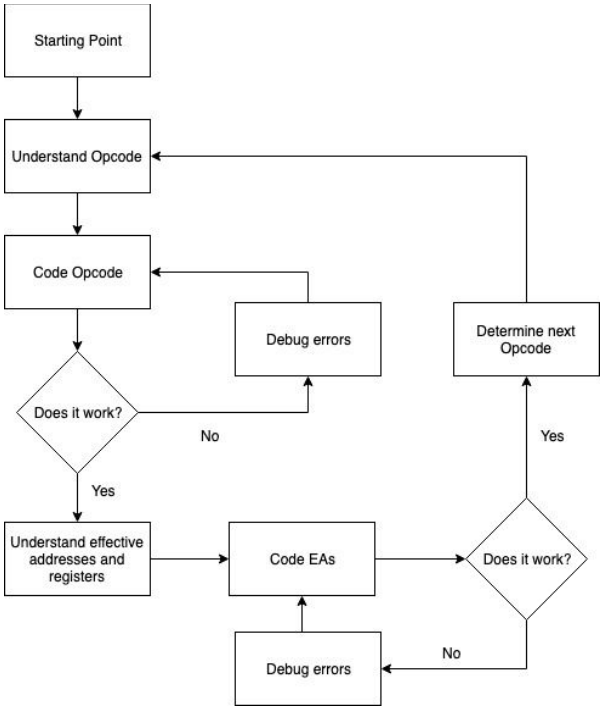
Team 69K | Michael Ward, Cole Liu, Tri Nguyen

Testing Description

As with any major software project, testing to make sure your code runs correctly is critical to the success of the project. To account for this, our team developed a testing regimen to stress test our code and ensure the results make sense. This regimen included starting with individual codes that we worked on and then checking the output to the .L68 files to make sure they were correct. We would also introduce errors to check and see if our program could handle them after we got the correct result. We incrementally continue to add new opcodes and test as needed while ensuring to keep copies of the code to revert back to if needed.

Below is a Chart 1 which shows the progression and steps we took to first tackle the opcode and then EAs. Test would occur at diamond points to determine if we would get the correct result.

Chart 1: Flow Chart of Test Plan



Coding Standards

Although not critical to the functionality of our project, coding standards helped us keep consistent and clean code that helped with readability and understanding what each other were working on. Our standard are set below:

Type	Example Code	Notes
Method Headers:	PRINT_OUT_LEA	Capitalize letters, use underscores for
Method Code	LEA LEA_STR, A1 JSR PRINT_FUNC JSR PRINT_SPACE MOVE.W TEMP_WORD, D3 JSR CHECK_TYPE_DATA	Use one tab, for all instructions, Instructions statements need two tabs or lined up. Instructions must be capitalized.
Repeated code:	LSL_4 LSL #4, D3 RTS	Determining code has been repeated, simplify it in a method
Jump Tables	FIRST_HEX_VAL_JMP_TABLE JMP FIRST_HEX_VAL_IS_0 * N/A JMP FIRST_HEX_VAL_IS_1 * ...	Jump tables need to have consistent jump methods names
Variables	* Starting variables START_ADDR EQU $50 END_ADDR EQU $100 TEMP_WORD EQU $150 DEST_VAR EQU $200	Variables start with 0 indents, and are followed with two tabs then initialization. Memory addresses are spaced every $50
Comments	* Get starting address from user	Always start with asterisk, then capitalized statement describing method

Testing Files

During the course of our project, we developed a few testing files to help with determining if we have the correct output. The original test file is below and will be included when submitting the project. The other test file was the one for the demo modified to exclude MOVEM as that was still being worked on.

```
*-----------------------------------------------------------
* Title      :      Test File for Team 69K
* Written by :     Michael Ward, Cole Liu, Tri Minh
Nguyen
* Date       :     5/11/2019
* Description:     Initial test file for disassembling project
*-----------------------------------------------------------
    ORG    $7000
START:                  ; first instruction of program

* Put program code here
    NOP
    RTS

    JSR        (A2)
    JSR        $00123658
    LEA        $00001234, A5
    LEA        (A4),A5

    NOP        (A4), $5000

    OR         D1,D2
    OR         D1, $00001234
    OR         D2, (A1)
    OR         (A1), D3
    OR         D1, (A2)+
    OR         -(A2), D3
    OR         D3, 0(A3,D5)
    OR         D3, 0(A3,D5)

    SUB.B      D5,D4
    SUB.W      D5,D4
    SUB.L      D5,D4
    SUB.L      D1,$00001234
    SUB.L      $00001234,D1
    SUB.L      D1, $FFFF1234
    SUB.L      $FFFF1234,D1
    SUB        (A0),D3
    SUB        D1, (A1)
    SUB        -(A1), D3
    SUB        D4, 0(A3,D5)
    SUB        D4, 0(A3,D5)

    BSR        TEST        * Extra

    BRA        NO_TEST
    BPL        TEST
    BGE        TEST
    BHI        TEST
    BVC        TEST
    BLT        TEST
    BCS        TEST

    ADD.B      D0,D1
    ADD.W      D1,D0
    ADD.L      $00001234,D1
    ADD        D1,$00001234
    ADD.L      $FFFF1234,D1
    ADD.B      D2, $FFFF1234
    ADD.B      #1,D5
    ADD.W      #10,D6
    ADDA       #5,A1
    ADDA.W     $0001234,A2
    ADDA.L     $FFFF1234,A0

    ROL        $00001234
    ROL        $FFFF1123
    ROR        $00001234
    ROR        $FFFF1123
    ROR        (A1)+
    ROR        -(A2)
    ROL.B      D1,D2
    ROL.W      D1,D2
    ROL.L      D3,D4
    ROL.B      #8,D5
    ROL.W      #4,D6
    ROL.L      #1,D1

    LSL        $00001234
    LSL        $FFFF1123
    LSR        $00001234
    LSR        $FFFF1123
    LSR        (A1)+
    LSR        -(A2)
    LSL.B      D1,D2
    LSL.W      D1,D2
    LSL.L      D3,D4
    LSL.B      #8,D5
    LSL.W      #4,D6
    LSL.L      #1,D1
```

```
ASL         $00001234
ASL         $FFFF1123
ASR         $00001234
ASR         $FFFF1123
ASR         (A1)+
ASR         -(A2)
ASL.B       D1,D2
ASL.W       D1,D2
ASL.L       D3,D4
ASL.B       #8,D5
ASL.W       #4,D6
ASL.L       #1,D1

MOVE        D1,$00001234
MOVE.L      $00001234,D0
MOVE.W      D1, $00001234
MOVE.B      $FFFF1234,D5
MOVE.B      (A1),$000012134
MOVE.W      -(A2),$FFFF1234
MOVE.L      (A1)+, D5
MOVE.W      A2,D3
MOVE.W      #123, D5
MOVE.B      #12,D6
MOVE.L      #14,D5
MOVE.L      #20, D7
MOVEA.W     $00001234,A1
MOVEA.L     $FFFF1234,A2
MOVEA.W     (A2),A1
MOVEA.L     (A2)+,A3
MOVEA.W     -(A3),A3
MOVEA.W     -(SP),A3
MOVEA.W     -(A7),A4

and.w     d0,d1
and.w     d0,$00004000
and.l     $00004000,d7
and.w     d0,(a3)
and.l     (a6),d7


not.b     d0
not.w     $000048C0
not.w     (a0)
not.b     $0B(a0)
not.l     (a0)+
```

```
    *SIMHALT            ; halt simulator
* Put variables and constants here
TEST
    MOVE.W      #0,D1
    RTS


NO_TEST
    MOVE.W     #4,D2
    RTS
    END    START        ; last line of source
```

Project Report

Team 69K | Michael Ward, Cole Liu, Tri Nguyen

Exception report

Overview

Overall, our program works well for the op codes that we were tasked with. However, there are still some lingering issues that we were unable to completely fix. In this report, we describe the problems and the results of the strenuous testing we did.

Problem Description

Address Issue:
Although our program gets inputs alright, when the program gets to FFFF, the program will stop due to our algorithm running word for word when going over data. Unfortunately, when we started, we did not think of the case when FFFF0001 could be reached and thus causing our program to stop. At this point, we are content with our programs execution but would change to long next time we implemented this or to have a check for the next word to see if there is a address value iterating to FFFF####.

Range Limit for Branch Statements:

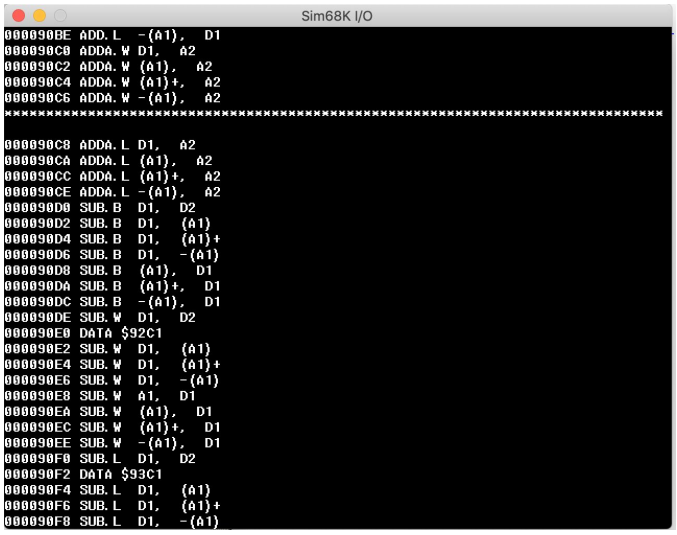
Another issue which deals with the address and amount of data that an opcode can handle is when a branch statement disassembles the instruction, it can print only the word, or #$1234, not the entire #$12345678 that could potentially exist. As such, when the address gets too long, it will cause issues with when the next address that the program will read next and could cause the program to break.

Add Conversion Issue:

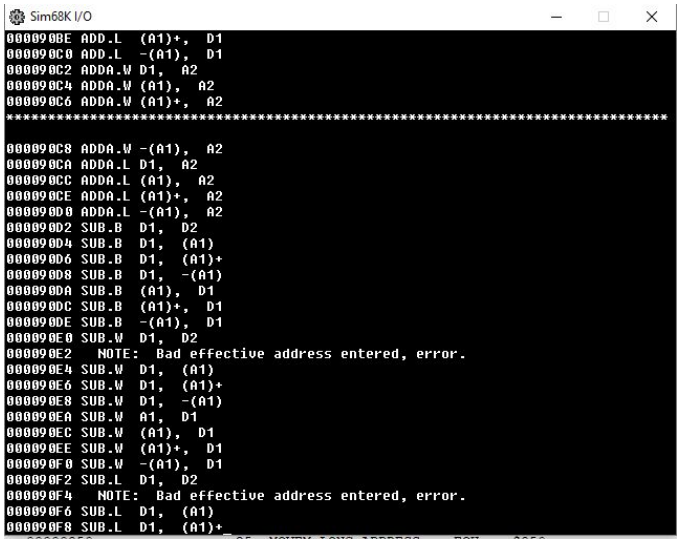
The subtraction code add sometimes has issues with automatically converting to ADDQ due to the similarity of the opcodes which we could not narrow down enough at the end of the project.

Results of Testing

The results of our tests were successful for the most part. If they were not, we would investigate the error in the opcode or EA. In order to understand our process for running tests, please review the Test Plan report. In terms of this paper, some actual examples of errors from the demo.X68 file are below for SUB:



Fortunately after working on the project after the presentation to the professor and TA, the issue with SUB has been fixed when the EA mode or destination mode is AX providing the user with a note of the incorrect mode (see image below).



Project Report

Team 69K | Michael Ward, Cole Liu, Tri Nguyen

Team Assignments and Report

Task Organization

The task organization for our group was evenly split with coding broken up into three main roles: *1) I/O and Documentation person*, *2) OPCode Person*, *3) EA and Algorithms Person*. Each role is described as below with explanation of impact to project.

I/O and Documentation/Coding Guidelines:

The in input and output aspects of our project, although less complicated than the opcode or effective addresses, is nonetheless important and critical to get correct. The team member responsible for this task was Cole Liu. This part of the code involved asking the user for start and end address. Additionally, this role was responsible for asking the user if they wanted to end the program or continue with disassembling other code.

Finally, due to the lighter role, the team decided that this would also include the documentation, coding guidelines and reports of the project to this role. By including the documentation, coding guidelines and reports, this person was able to learn about the other sections of the code and ask questions and make suggestions on algorithms. The percent of coding for this role would be 25% with most of the extra effort going towards documentation, code review, and reports and should be regarded as a equal member of the team.

OPCode Person

The main part of the code would be around the understanding the opcodes and parsing the information. The team member responsible for this section was Misha Ward. Parsing the opcodes and understanding the breakdown of them was instrumental to producing the correct output to the screen. Although the entire team helped with understanding how to parse the opcodes, this team member worked hard to put it into code.

Much of the coding was accomplished with the help of the EA and Algorithms Person who helped the jump tables and initial logic. These algorithms included finding the different hexadecimal values for the opcode and then breaking down how the registers/addressing modes and data will be printed. Sections critical to this role include

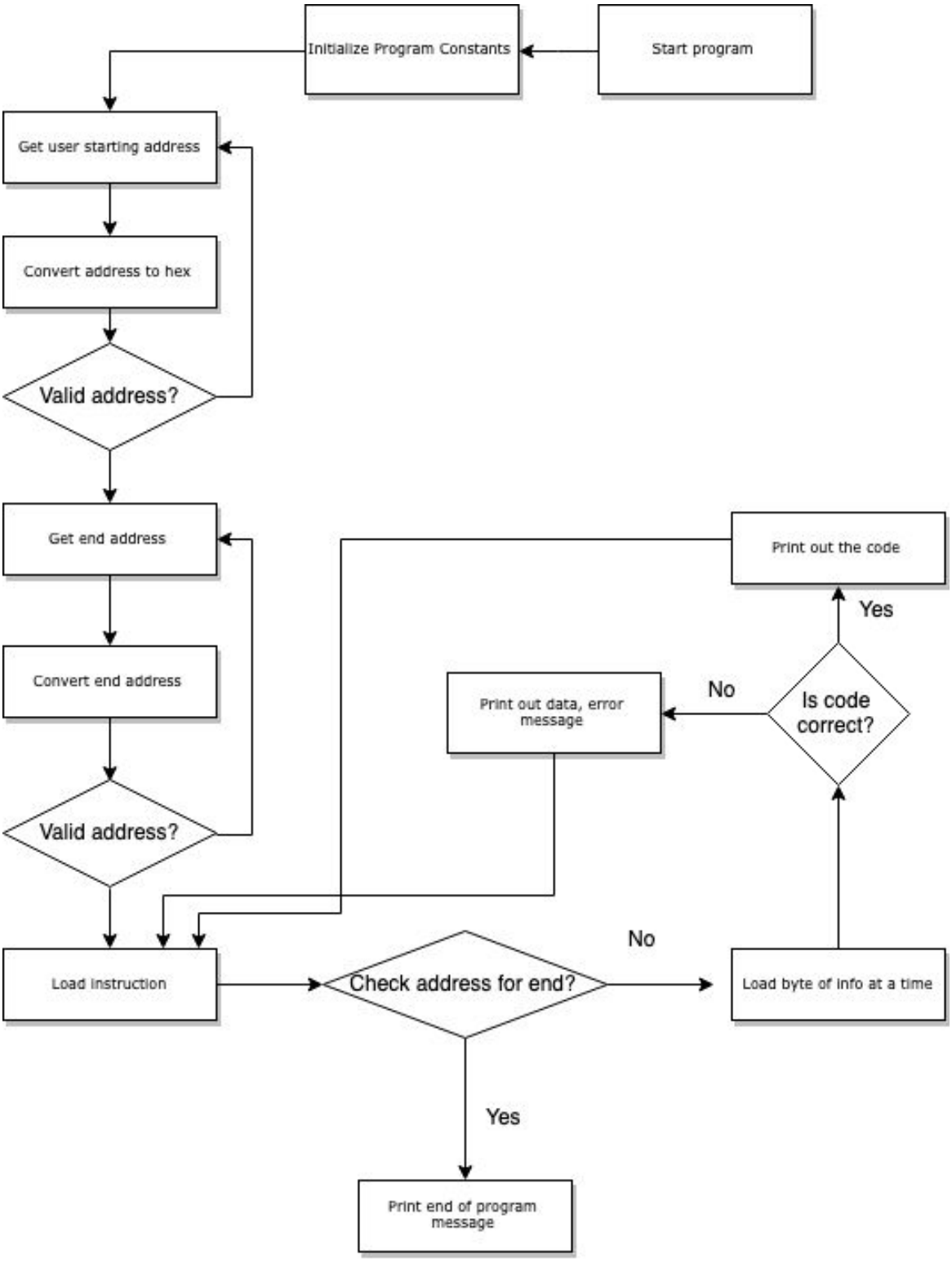
the PARSE_OPCODE, FIRST_HEX_VAL_IS_X, and various print opcode methods. Due to the heavy coding aspects of this role, the percent of coding is estimated at 40%.

EA and Algorithms Person

Although the OPCode section was critical to getting correct results to the screen, the effective address and algorithms person was also instrumental to the project. In our team, Tri Minh Nguyen filled this role. This person not only worked on analyzing the effective addresses and how to break them down and print them to the screen but also helped the OPCodes person with algorithms. This person was responsible for researching better ways of coding such a huge project. As such, Tri was able to find information on how to use jump tables which organized our code and made it easier to switch to different cases. This was important due to the complexity of the project and the multiple paths that an op code could potentially take.

The code behind this role was significant and thus, accounts to roughly 35% of the total code base that our team created. This role's contribution to the project was critical and the research into better methods to organize our code paid off by reducing errors and allowing each of us to work more effectively as individual due to the segmented coding that the switch statements provided.

Appendix


Initialize Program Constants
Start program
Get user starting address
Convert address to hex
Valid address?
Get end address
Convert end address
Valid address?
Load instruction
Check address for end?
No
Yes
Print end of program message
Load byte of info at a time
Is code correct?
No
Print out data, error message
Yes
Print out the code