68K Disassembler

Contributors

Scott Shirley @scottin3d
Carl Howing @cjhowing7
Daniel Yakovlev @yakovdan98

(1) Project Links

Github repository: https://github.com/Scottin3d/Disassembler
Project Diagram: https://drive.google.com/open?id=1ZPQ33-BFiRYbSNXN15DFX_wUENFzlZGl

(2) Documentation

Program Description

This program is written in Motorola 68000 assembly language (M68k), and its purpose is to disassemble data back into human readable opcodes and effective addresses.

A disassembler (also called an inverse assembler) should do the following:

- Scan a section of memory and attempt to convert the memory's contents to a listing of valid assembly language instructions
- Most disassemblers cannot recreate symbolic, or label information
- Disassemblers can be easily fooled by not starting on an instruction boundary

How it works:

- Program parses the op-code word of the instruction and then decides how many additional words of memory need to be read in order to complete the instruction
- If necessary, reads additional instruction words
- Prints out the complete instruction in ASCII-readable format
- Converts binary information to readable Hex

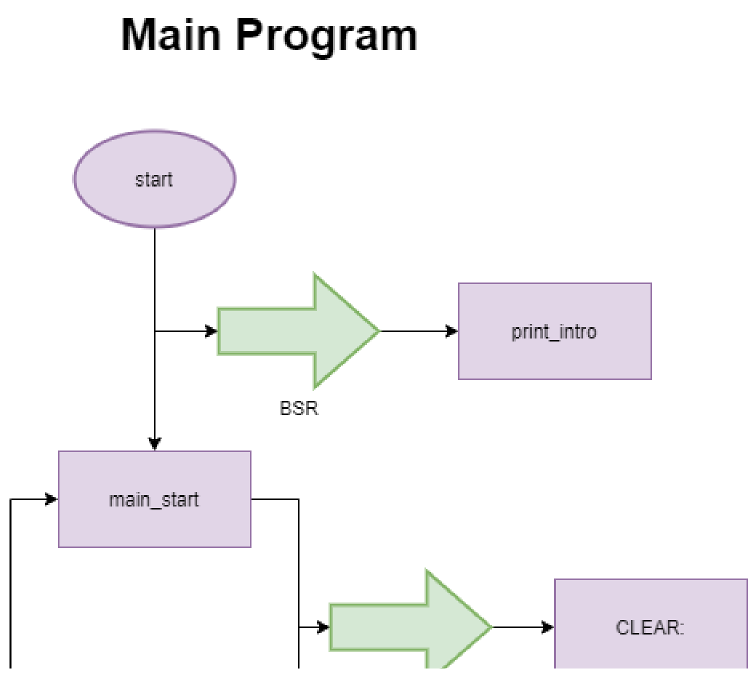
Specifications

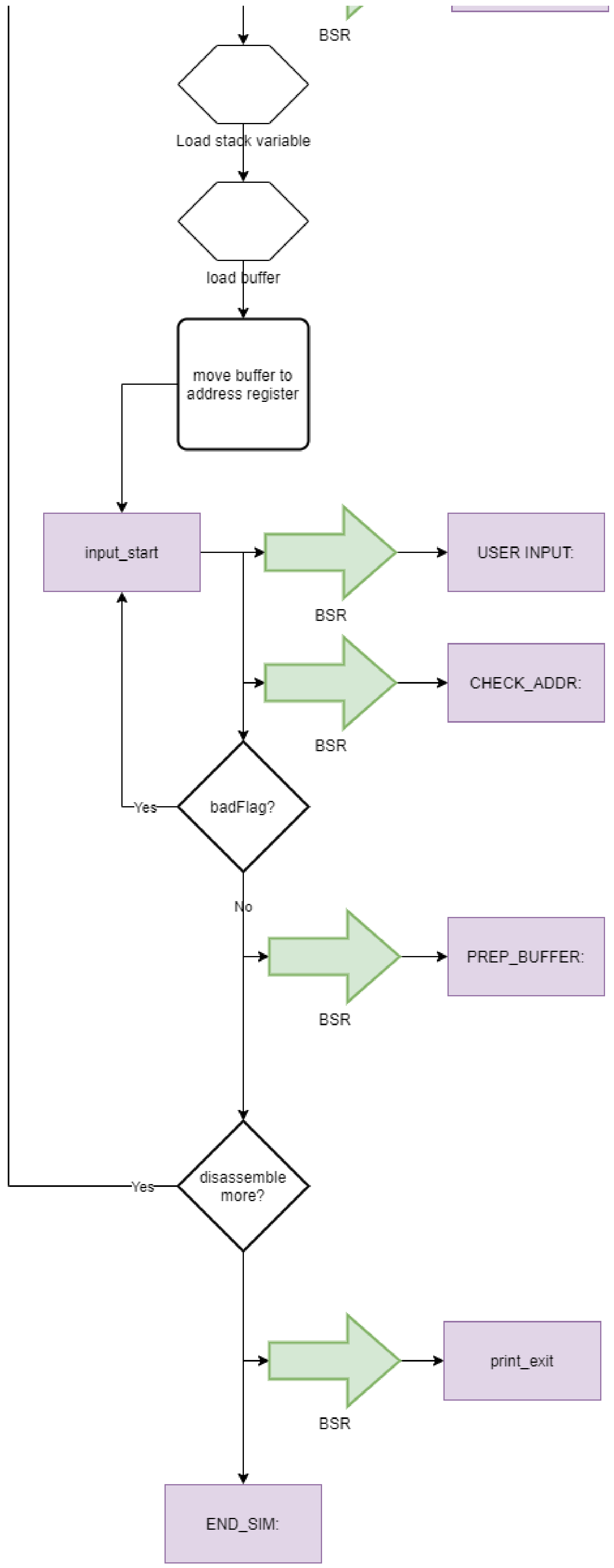
Program Design

Project Diagram: https://drive.google.com/open?id=1ZPQ33-BFiRYbSNXN15DFX_wUENFzlZGl

Main Program

This chart outlines the main flow of the disassembler from start to finish. Please see the project diagram for a more detailed look at the logic of the individual subroutines.





Supported Opcodes

O	P	C	O	D	E	S

O	P	C	O	D	E	S
MOVE	MOVEA	MOVEM	ADD/A/I*	SUB/A/I*	LEA	AND/A/I
OR/I	LSL	ASR	CMP/I*	Bcc(BCC, BGT, BLE)	JSR	RTS
ERO/I*	NEG*	NOT*	NOP*			

*Not required for project

Supported Effective Address (EA) Modes

- Data Register Direct
- Address Register Direct
- Address Register Indirect
- Immediate Data
- Address Register Indirect with Post incrementing
- Address Register Indirect with Pre decrementing
- Absolute Long Address
- Absolute Word Addres

Test Plan

There were 5 phases in our testing plan. We tested the system as we built it up. This was done because newer code relied on a framework to correctly work. Once the framework was tested and debugged. Opcode disassembly code was written and then tested.

1. I/O debugging, making sure that it correctly converted to ASCCI.
2. EA decoding, being able to correcly output the mode and register used. tested with MOVE.B command.
3. Local Opcode testing, once each member wrote the disassembly for the specific opcodes they wrote, testing was done for that specific opcode to make sure that it worked correctly.
4. Whole System, testing was done on the complete system. All of the local opcode tests were brought together and used in a single test file.
5. Decoding disassembler, The starting address of the disassembler was used to test that it properly output the correct disassembled instruction aswell as had the correct output for commands that were not meant to dissassemble.

How Tsting Was Conducted

For each opcode, we created a test file that we would load the data into our main program before execution.

We tracked bad instruction outputs at the top of the file and idicate the bad instruction in-line. We also note the start and stop of the file in memory for easy use when testing. Each test file tests all required cases for the opcode, separated in groups or registers/address and absolute/immediate data.

Bad Instructions

When the program tries to decode a hex word that is not supported, it will generate a bad instruction. When a bad instruction is generated, the word is output to the console in the below format. At the end of the program, it will tell the user how many bad instructions were generated.

Example:

```
$0x0000040A     DATA    $005F
```

Coding Standards

When writing code, subroutines should be left justified, followed by the instruction at three tabs. Comments should be made at nine tabs. Each line does not require a comment, however an asterisk should be placed at nine tabs.

Register Usage

For continuity, we utilized registers as followed:

Register	Usage
D0	copy of mask
D1	utility register
D2	copy of working address
D3	utility register
D4	local counter

Register	Usage
D5	flag condition
D6	master counter
D7	working address
Address	**Usage**
A0	temp address holder
A1	trap address
A2	buffer address of decoded instruction
A3	
A4	starting address
A5	ending address
A6	jump tables
A7	stack

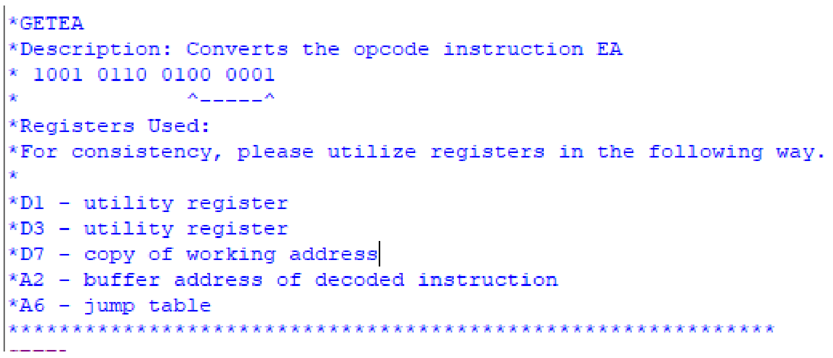
Subroutine Standards

When writing subroutings, clearity is important. Coders should be able to tell what the subroutines task is by reading the code as well have imformative comments.

```
1. **********************************************************************
2. *NAME
3. *Description:
4. *Registers Used:
5. **********************************************************************
6. **NAME of operation***************************************************
7. *=====================================================================
```

1. A solid line of 70 asterisks indicates the start of a subroutine
2. The name of the subroutine should be clear and unambigous
3. A brief description of what the subroutine accomplishes
4. The registers utilized in the subroutine, adhering to the usage chart above
5. A solid line of 70 asterisks indicates the end of the header
6. Sub-operations should be indicated with a solid line of 70 asterisks, the the name two (**) from the left
7. a solid line of 70 equals indicates the end of a subroutine

Eample:



```
*GETEA
*Description: Converts the opcode instruction EA
* 1001 0110 0100 0001
*             ^-----^
*Registers Used:
*For consistency, please utilize registers in the following way.
*
*D1 - utility register
*D3 - utility register
*D7 - copy of working address
*A2 - buffer address of decoded instruction
*A6 - jump table
************************************************************
-----
```

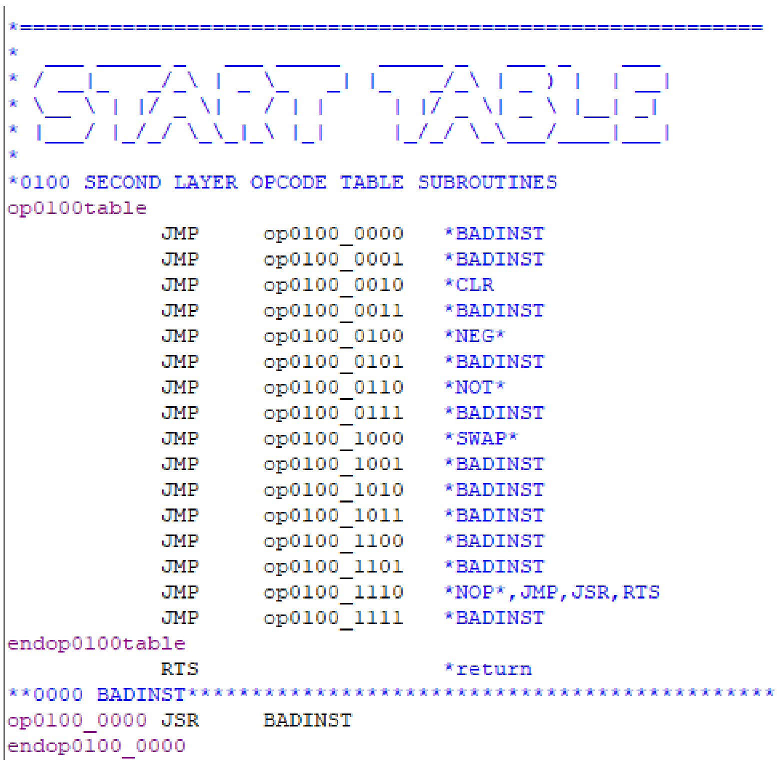
Jump Tables

While used seldomly, formatting is important in the use of jump tables. When writing a jump table, the following formatting guidelines should be followed along with the general coding and subroutine guidelines already eastablished.

```
1.*0100 SECOND LAYER OPCODE TABLE SUBROUTINES
2.op0100table
3.          JMP      op0100_0000    *BADINST
            JMP      op0100_0001    *BADINST
            JMP      op0100_0010    *CLR
            JMP      op0100_0011    *BADINST
```

1. Name and description of the jump table
2. The name of the table should be clear and unambigous and include "table" at the end
3. The jump routines have a comment idicating what the action is

Example:



```
*=============================================================
*  ___ _____ _   ___ _____   _____ _   ___ _    ___
* / __|_   _/_\ | _ \_   _| |_   _/_\ | _ ) |  | __|
* \__ \ | |/ _ \|   / | |     | |/ _ \| _ \ |__| _|
* |___/ |_/_/ \_\_|_\ |_|     |_/_/ \_\___/____|___|
*
*0100 SECOND LAYER OPCODE TABLE SUBROUTINES
op0100table
            JMP      op0100_0000    *BADINST
            JMP      op0100_0001    *BADINST
            JMP      op0100_0010    *CLR
            JMP      op0100_0011    *BADINST
            JMP      op0100_0100    *NEG*
            JMP      op0100_0101    *BADINST
            JMP      op0100_0110    *NOT*
            JMP      op0100_0111    *BADINST
            JMP      op0100_1000    *SWAP*
            JMP      op0100_1001    *BADINST
            JMP      op0100_1010    *BADINST
            JMP      op0100_1011    *BADINST
            JMP      op0100_1100    *BADINST
            JMP      op0100_1101    *BADINST
            JMP      op0100_1110    *NOP*,JMP,JSR,RTS
            JMP      op0100_1111    *BADINST
endop0100table
            RTS                     *return
**0000 BADINST*************************************************
op0100_0000 JSR      BADINST
endop0100_0000
```

Test Files

Because of how we divided individual contributions to the project, each opcode routine was written separately from the main program. This allowed us to work on multiple subroutines at a time without conflict in the main program.
When we integrated each opcode subroutine into the main program, we wrote a test case for the opcode and tested it extensively to ensure it was integrated properly and functioned as written.

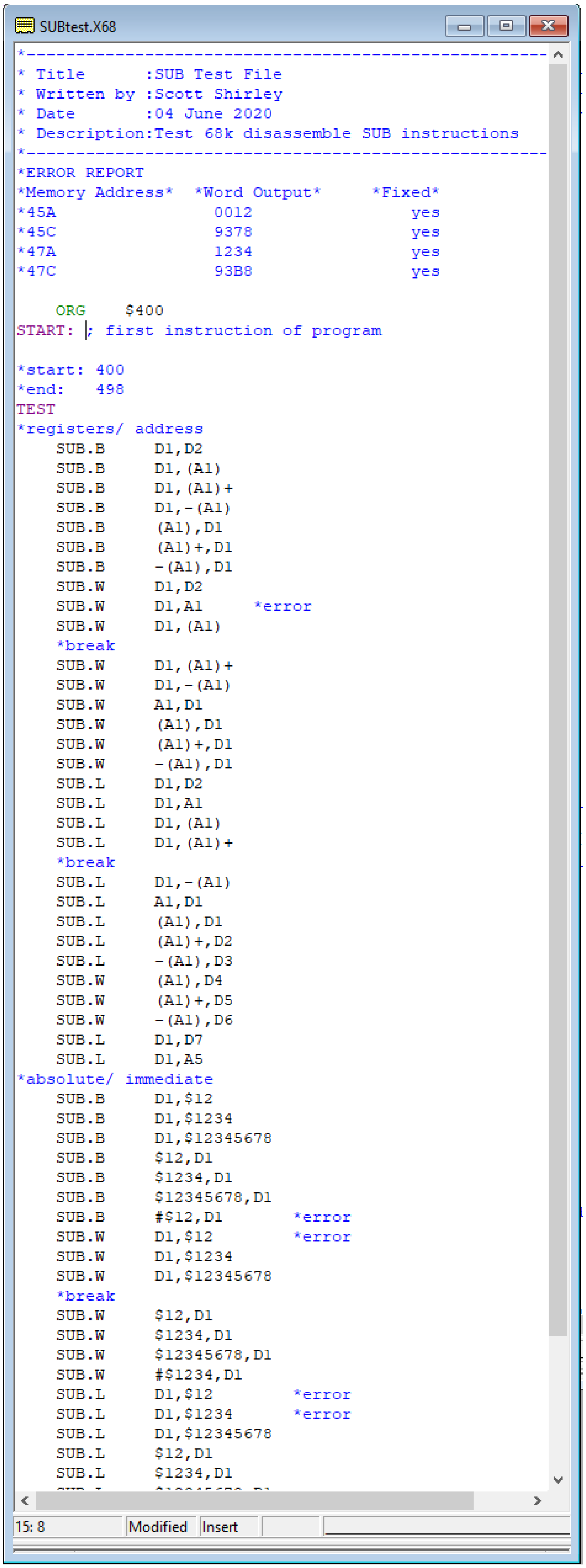
```
1.*ERROR REPORT
  *Memory Address*  *Word Output* *Fixed*
2.*45A                0012          yes


3.    ORG    $400
  START:
4.*start: 400
  *end:   498
  TEST
5.*registers/ address
      SUB.B     D1,D2
6.*absolute/ immediate
      SUB.B     #$12,D1       *error
```

1. Report of errors while decoding the test cases
2. Error location, output, and if its been fixed
3. All tests start at $400
4. Mark file with start and end locations for use
5. Included test cases for registers and address if applicable
6. Included test cases for absolute and immediate data if applicable

List of Test Files

GitHub: https://github.com/Scottin3d/Disassembler/tree/master/tests

Example:



```
*-----------------------------------------------------------
* Title      :SUB Test File
* Written by :Scott Shirley
* Date       :04 June 2020
* Description:Test 68k disassemble SUB instructions
*-----------------------------------------------------------
*ERROR REPORT
*Memory Address*  *Word Output*     *Fixed*
*45A                0012               yes
*45C                9378               yes
*47A                1234               yes
*47C                93B8               yes

    ORG    $400
START: ; first instruction of program

*start: 400
*end:   498
TEST
*registers/ address
    SUB.B     D1,D2
    SUB.B     D1,(A1)
    SUB.B     D1,(A1)+
    SUB.B     D1,-(A1)
    SUB.B     (A1),D1
    SUB.B     (A1)+,D1
    SUB.B     -(A1),D1
    SUB.W     D1,D2
    SUB.W     D1,A1      *error
    SUB.W     D1,(A1)
    *break
    SUB.W     D1,(A1)+
    SUB.W     D1,-(A1)
    SUB.W     A1,D1
    SUB.W     (A1),D1
    SUB.W     (A1)+,D1
    SUB.W     -(A1),D1
    SUB.L     D1,D2
    SUB.L     D1,A1
    SUB.L     D1,(A1)
    SUB.L     D1,(A1)+
    *break
    SUB.L     D1,-(A1)
    SUB.L     A1,D1
    SUB.L     (A1),D1
    SUB.L     (A1)+,D2
    SUB.L     -(A1),D3
    SUB.W     (A1),D4
    SUB.W     (A1)+,D5
    SUB.W     -(A1),D6
    SUB.L     D1,D7
    SUB.L     D1,A5
*absolute/ immediate
    SUB.B     D1,$12
    SUB.B     D1,$1234
    SUB.B     D1,$12345678
    SUB.B     $12,D1
    SUB.B     $1234,D1
    SUB.B     $12345678,D1
    SUB.B     #$12,D1        *error
    SUB.W     D1,$12         *error
    SUB.W     D1,$1234
    SUB.W     D1,$12345678
    *break
    SUB.W     $12,D1
    SUB.W     $1234,D1
    SUB.W     $12345678,D1
    SUB.W     #$1234,D1
    SUB.L     D1,$12         *error
    SUB.L     D1,$1234       *error
    SUB.L     D1,$12345678
    SUB.L     $12,D1
    SUB.L     $1234,D1
```

Exception Report

Limitations

Opcodes & Effective Addresses 1. This program is limited to the opcode and effective addresses listed above. While significant testing was done to ensure that the supported codes functioned properly, there is a chance that unspoorted opcodes will not trigger a bad instrction as intended.
See above for more detail about bad instructions.

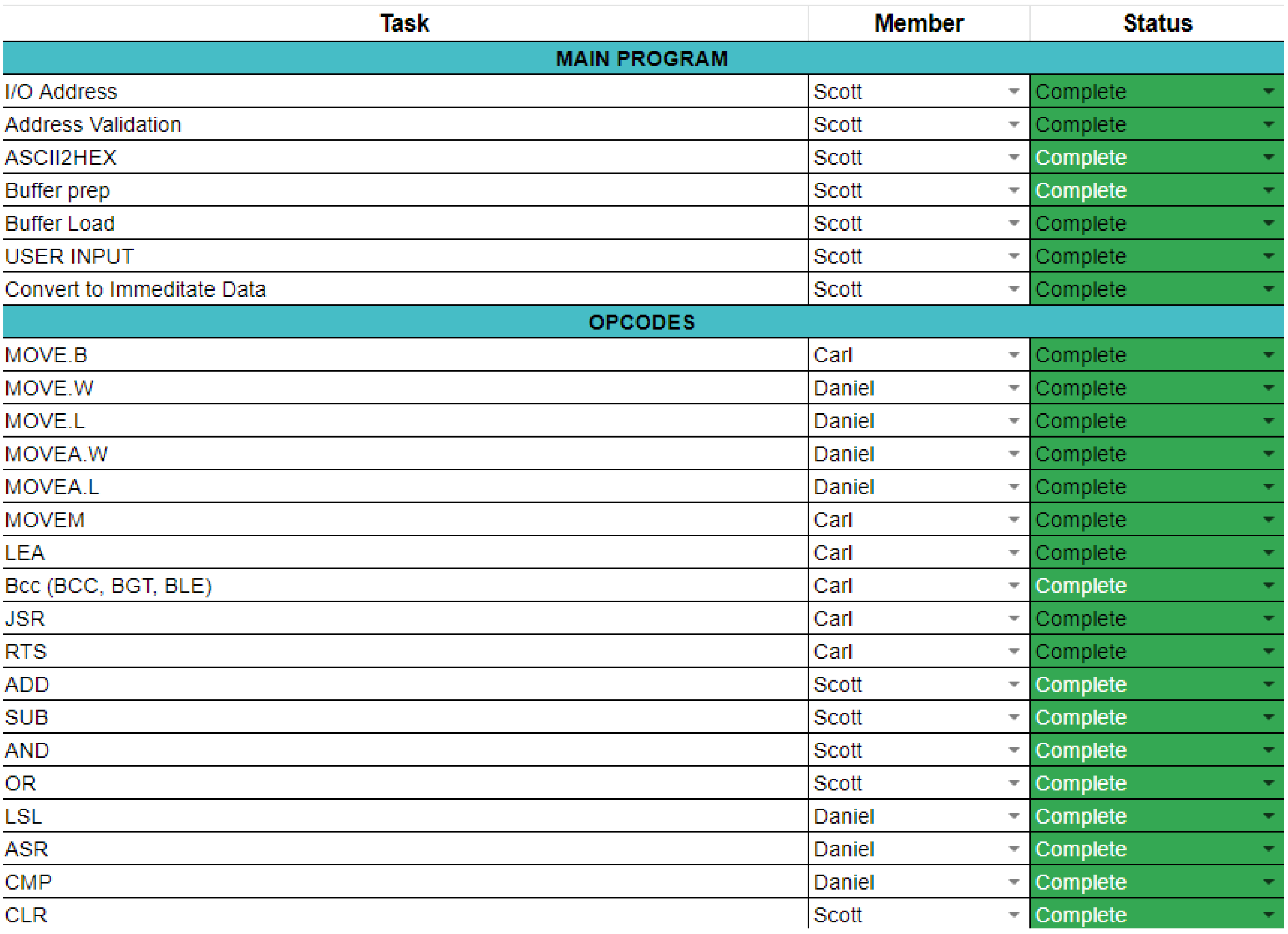
Ending 1. If Y/y is not pressed the program will end. If nothing is entered and return is pressed, the program will end.

Know Bugs

Team Assignments and Report

We used a spread sheet with tasks and assignments to track progress of our project. It broke down the project into small tasks which each contributor self-assigning a role. Tasks were marked either "Not Started, In Progress, or Testing". Once a task had successfully passed a test case, it was marked complete.

Project Task Board



Task	Member	Status
MAIN PROGRAM		
I/O Address	Scott	Complete
Address Validation	Scott	Complete
ASCII2HEX	Scott	Complete
Buffer prep	Scott	Complete
Buffer Load	Scott	Complete
USER INPUT	Scott	Complete
Convert to Immeditate Data	Scott	Complete
OPCODES		
MOVE.B	Carl	Complete
MOVE.W	Daniel	Complete
MOVE.L	Daniel	Complete
MOVEA.W	Daniel	Complete
MOVEA.L	Daniel	Complete
MOVEM	Carl	Complete
LEA	Carl	Complete
Bcc (BCC, BGT, BLE)	Carl	Complete
JSR	Carl	Complete
RTS	Carl	Complete
ADD	Scott	Complete
SUB	Scott	Complete
AND	Scott	Complete
OR	Scott	Complete
LSL	Daniel	Complete
ASR	Daniel	Complete
CMP	Daniel	Complete
CLR	Scott	Complete

Project Link: https://docs.google.com/spreadsheets/d/1MeqKPhHo7Z_27Mj2EwtO_A_3y4oFlFxuZ0nDk8N5tSQ/edit?usp=sharing

Scott

- I/O
- Helper Subroutines
- Opcodes
- EA
- Testing

Carl

- Opcodes
- EA
- Operational Direction

Daniel

- Opcode decoding

(3) Demonstration

Video Demonstration

Video Link: http://www.youtube.com/watch?v=LZHVBtVmlug