The Disassembler by Abhi Trivedi, Conor Van Achte, and Sam Meyerding

1) **Program description**

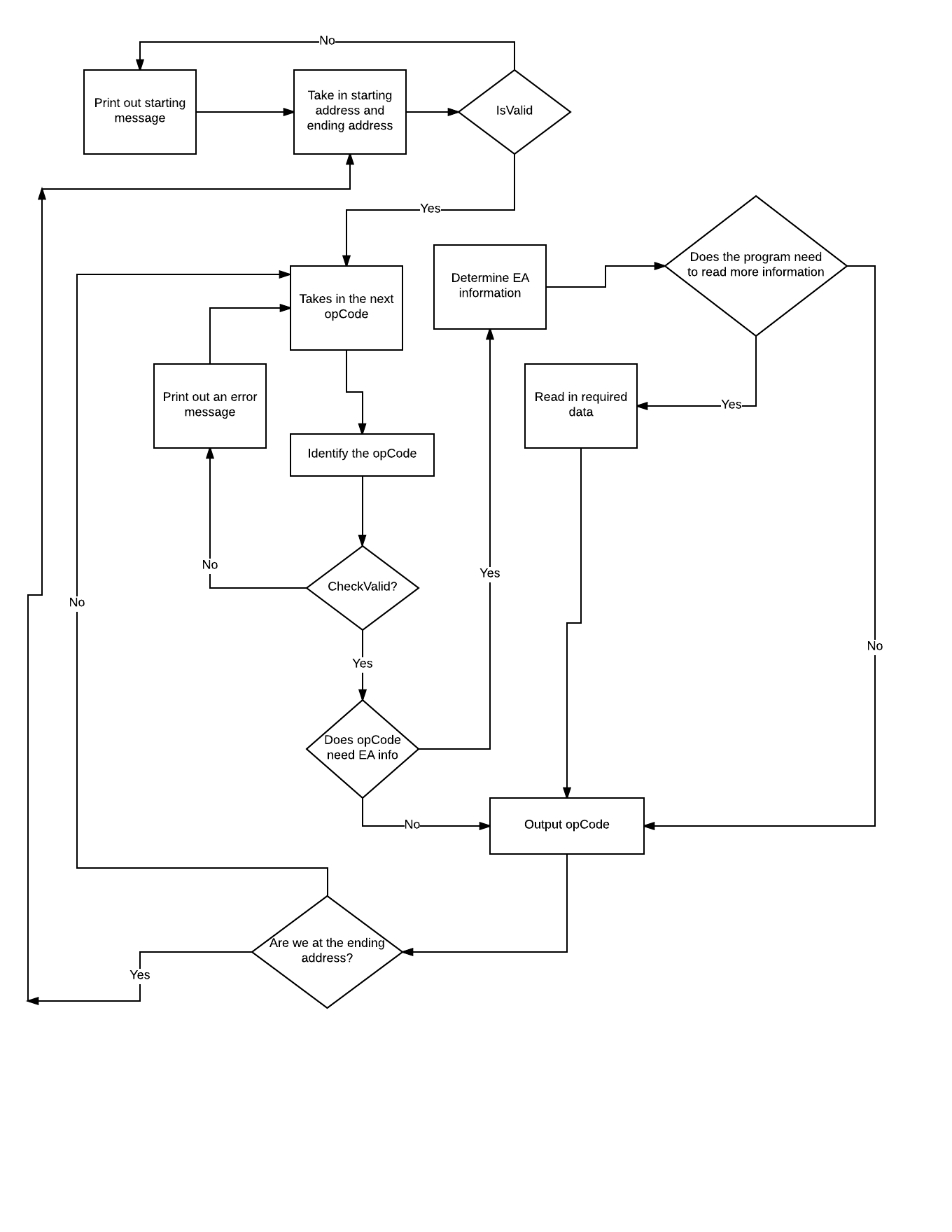
Design Philosophy: We designed our program using an agile method. The first thing we did as a group was to come up with a flow chart of how the disassembler was going to work. Next we created a flowchart for how the coding of the disassembler was going to look like. We continued to meet twice a week and communicated our problems through discord or email. We also created a github repository from where each member could pull and push their code accordingly. We did not use any canned routines other than from TRAP task 1-14.

There were three sections which made the disassembler work, the first section dealt with the input and output, which gathered input from the user such as the starting and the ending location and printed the decoded information.

The second section was the opcodes. The opcodes section deciphered the type of opcode and printed whether it was valid or not. It did not use a Jump table, because it was mostly finished very early on and the person who did it didn’t know about Jump tables yet.

The third section was the effective addressing section which broke down the valid addressing modes for each of the opcodes and printed out whether the effective addressing was valid or not.

The flowchart is on the next page: We created this flowchart in the third week of the project and updated as needed. The flowchart allowed to understand how our disassembler was going to work, and gave us a starting point.



2) **Specification**:

Our program prompts the user with a request for a starting address to disassemble from. Once the program has been given a starting address, the program will ask for an ending address. Once given an ending address, the program will print to the command screen the lines of code that were in between the entered starting address and end address. The command prompt will display a number of lines on the screen until a button is pressed for the rest of the disassembled code, if the addresses given have too many lines to be displayed on one screen.

2.1 OpCodes Completed- There are some extra non-required ones.

MOVE

MOVEA

ADDI

BTST

NOP

RTS

LEA

CLR

JSR

MOVEM

ADDQ

MOVEQ

ROL

All conditional branches (Bcc)

DIVU

OR

SUB

CMP

CMPA

ADD

ADDA

MULUS

AND

LSL

LSR

ASL

ASR

2.2 Effective Addressing:

1. Data Register Direct
2. Address Register Direct
3. Address Register Indirect
4. Immediate Data
5. Address Register Indirect with Post incrementing
6. Address Register Indirect with Pre decrementing
7. Absolute Long Address
8. Absolute Word Address

3.1 **Test Plan**:

We tested our program originally with test code contained in our disassembler. We kept all of that test code at the beginning of the disassembler, so that it would be easy to remove. Also, the first instruction was a BRA that went past the test code. This was our original test plan, because we had not yet learned how to load code into memory directly.

When we implemented validation for our OPCodes, we tested it by telling our disassembler to disassemble code that contained instructions that were invalid for our OPCodes, such as SUBI, a memory location with no instruction, and SUBA.

In order to test our input and output portions of our code, we sent the related subroutines valid data and stepped through them to make sure that they were doing everything correctly. We then tested them by running through sections of code repeatedly with different data and verifying that the input and output were producing correct results.

Our test plan in recent weeks involved testing each OPCode comprehensively with every combination of size and EA. We created a file that had every implemented OPCode with every valid case. We then disassembled that code and validated the results of our disassembler section by section. We ran into a problem with how we had implemented disassembly of MOVEM because we had forgotten that the syntax for MOVEM included slashes. Without slashes, the syntax for MOVEM is

MOVEM.L An-Am  
MOVEM.W An-Am  
MOVEM.L Dn-Dm  
MOVEM.W Dn-Dm

With slashes the syntax is altogether much more complicated, so we had to redo the handling for MOVEM.

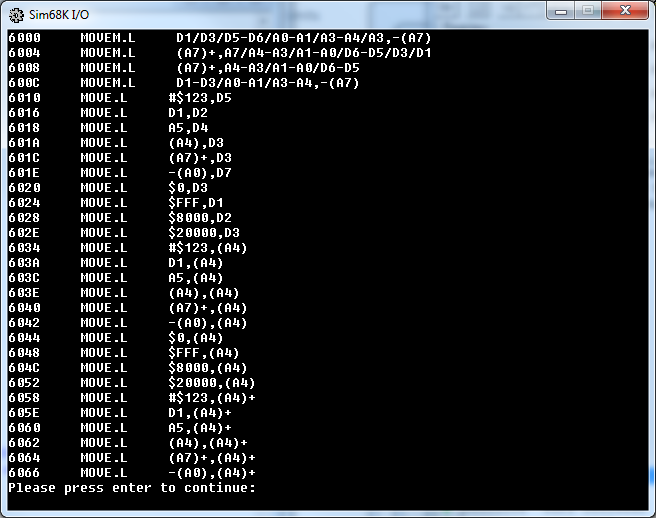
When an error was found, we would step through the code in order to determine where the error was coming from. Then, when the error was found, we would fix it.

The bulk of our testing was done in the final two weeks after we had made a comprehensive test file.

3.2 **Coding Standards:**  
Comments! (We wish this had been kept to better.)  
Tell the rest of the group if you’re modifying the code to avoid collisions.  
Update table on Google drive when an OPCode is finished.  
The current address should be incremented by a word once at the start of each OPCode and then only by the EA person.

4) **Exception report**:

Many of the problems our team encountered had to do with meeting and debugging. We ran into some issues with the weather and timing of meet ups due to differing schedules, and unpredictable weather. We were able to implement all of the opcodes mentioned in the specification document as well as full implementation of Bcc, BTST, JMP, and CMPA.

4.1 Output   


5) **Team assignments and report**:

We divided the project into three suggested roles

Abhi Trivedi - I/O person

Conor Van Achte - EA person

Sam Meyerding- Opcode Person

– I/O Person: Interfaces to user • Decodes inputs from user • Formats and displays– -Op-code person: Decodes op-code

– EA Person: Decodes effective addresses, uses EA field information passed on by Opcode person

disassembled code.

Group responsibilities:

Decide on roles,

Design algorithm, coding conventions and parameter passing rules,

Design test program ,

Meet to integrate and test

–Test, test, test!

Turn it in.

Status reports