Disassembler Documentation

Oscillation Apparent

Adam Ali, Liam Madson, Sharanya Sudhakar

Table of Contents

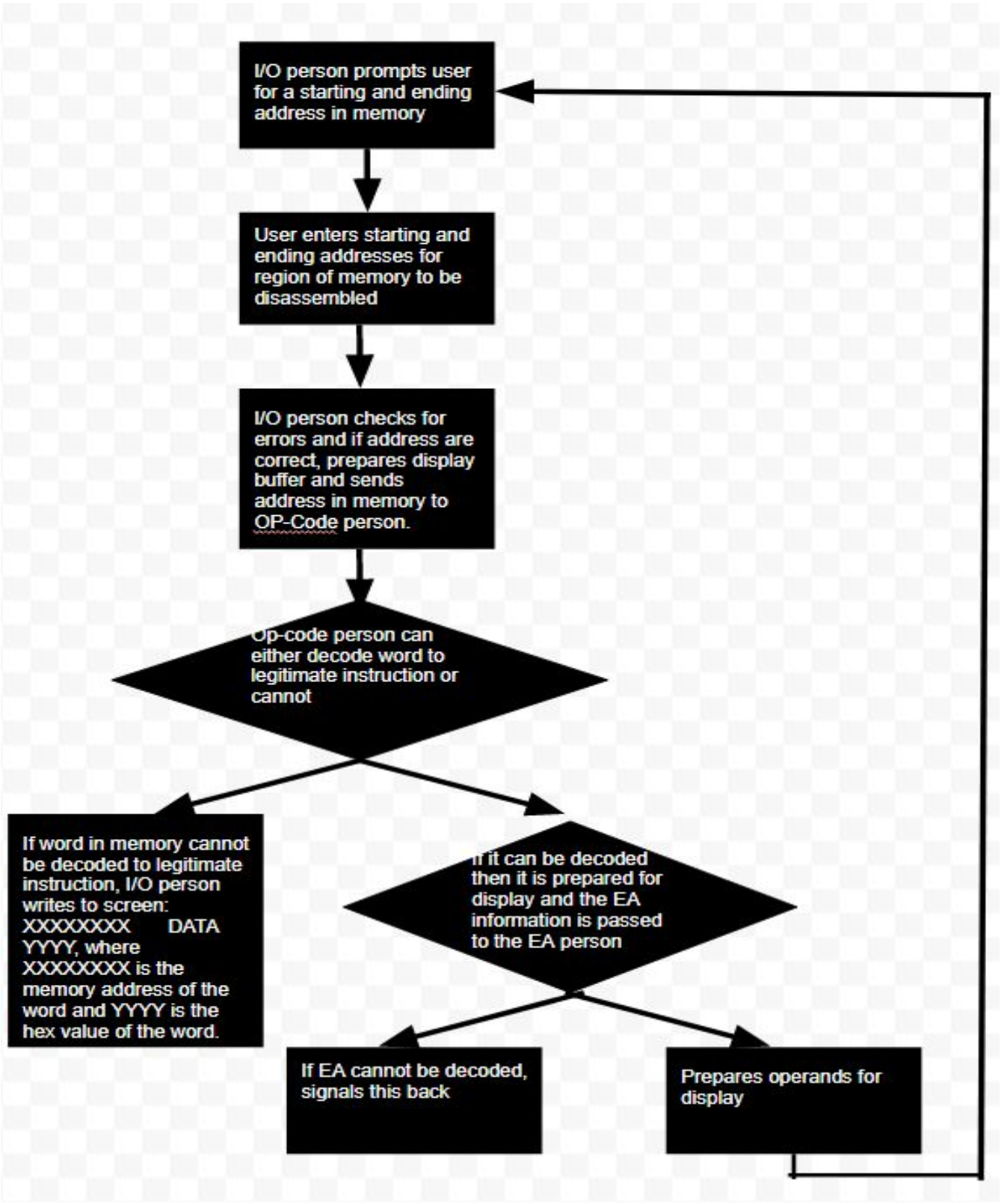
Project Description

For the program flow we used mainly the ideas create from the flow chart during our first week which is below. This breaked down the jobs into manageable task allowed us to clearly define our roles of an input/output person, effective address person and operation code person. The flowchart also showed were we needed to pass off data between roles which helped us establish what data and address registered we wanted to use for those specific task.

To develop the input we used mainly resources and processes from previous work to build the it but needed to add error checking to make sure we didn't accept any invalid starting or ending locations. For output it was a bit harder since we had to handle both ascii and hex as input for the output but needed it all to be in ascii so we designed a converted to use when we inputted hex to output.

To build our operation codes we relied quite heavily on a sheet created by golden creek which had the operation breakdown which had the information from the manual concentrated on operation codes and effective address into a single sheet. This helped us implement the operation code by just looking at the bits on the document instead of looking at the hex outputted by EASy68K. This allowed us to sort into bin using the first few bits of the passed data and a jmp table, which we were then able to sort even further by taking trying the get to the specific operation code and size for the code.

For effective addresses we build all of the subroutines needed, and then build specific code to handle each operation code. This allowed us to refined what would and wouldn't be accepted when passed, it also allowed for once the infrastructure was done of all the subroutines it was almost a template we could follow to complete all effective addresses.



http://goldencrystal.free.fr/M68KOpcodes-v2.3.pdf

Program Specification

Our program is a program written in 68K meant to disassemble a precompile 68K which has created a usable `.S68` file. The goal was so that when you have a 68K program file which is written in binary/ hex you are able to get a output of the code instead in assembly an easier to read language. As per the specification sheet it handles all operations codes and effective addresses in the table below, as well as the additional functions of `ADDI`, `SUBI`, and all `BCC` condition instead of just `BCS`, `BGE`, `BLT`, `BVC`. We also build a test file to confirm that all methods were implemented and outputted the correct data.

Instructions (OPCodes)
MOVE, MOVEA, MOVEM
ADD, ADDA
SUB, SUBQ
MULS, DIVS
LEA
OR, ORI
NEG
EOR
LSR, LSL
ASR, ASL
ROL, ROR
BCLR
CMP, CMPI
Bcc (BCS, BGE, BLT, BVC)
BRA, JSR, RTS

Effective Address Modes	
Data Register Direct	`Dn`
Address Register Direct	`An`
Address Register Indirect	`(An)`
Address Register Indirect with Post-incrementing	`(A0)+`
Address Register Indirect with Pre-decrementing	`-(SP)`
Immediate Data	`#`
Absolute Long Address	`(xxx).L`
Absolute Word Address	`(xxx).W`

How to use Disassembler:

1. Open `disassembler.X68` in EASy68K.
2. Compile and load `disassembler.S68` in SIM68K.
3. If you want to test Disassembler on itself, simply run the simulation and set the starting and ending addresses at some range within the bounds indicated in `disassembler.L68` (listing file). It is ORG'd at $1000. Skip to step 7.
4. If you want to test Disassembler on a different `.68K` file, such as `test.68K`, then generate a `.S68` file for it.
5. With `disassembler.S68` open in SIM68K, go to *File* > *Open Data…* and choose `test.L68`, then run the simulation.
6. Point Disassembler at some address range of your test program, no less than what is is ORG'd at and no higher than its last instruction.
7. Press [ENTER] to cycle through pages of disassembly. Re-run the simulation to restart the program. Remember to reload data if you are using an external test file if SIM68K is closed.

Any line with `ADDR. DATA XXXX` is an instruction that could not be decoded by Disassembler. This is expected behavior for when Disassembler encounters an address with no definitive instructions present.

Our team didn't create a set group of standards at the beginning since none of us knew what to expect when coding in assembly, so our standards were developed concurrently with our code. We tried to build the code groups which handled all binary code and then from there limits them from there to valid operations codes. When we get to a valid operations code then we add it to A2 and then print it out, then look at the size bits and print out the appropriate size if require. Then we start to look at the effective address bits and print out the effective address once we have confirmed it a valid type.

Test Plan

The test file we created is called `testing.X68`, the goal of the file to contain brute force tests for all of required functions with all size, source and destinations. Once we had this file we used it to do test on the effective addresses, operation codes.

To do test with the operations codes we decides to do tests with each individual operations code when building them to confirm they worked and once all of them were tested individually we did bulk testing of the whole `testing.X68` file. For EA testing we used the same approach of individual and then bulk this allowed us to have good coverage of test and catch any errors. To test both EA and OP code with our test file we used the `testing.S68` file and used it as data for our `dissembler.X68` when running. We didn't really implement any logging to test we mainly just used the brute force technique of running all of the functions and then checking that we had the correct output ourselves.

To test our input and output we did independent testing since we didn't have our other parts complete at the time so we did manual testing for it. Where we test at multiple different starting and ending address to make sure that we didn't encounter errors, to stop the error of us running into our own memory we added a lower limit to the starting location, we also tried some invalid operation code to make sure we were getting the data output.

Exception Report

Issues

`MOVE` and `DATA` (for unrecognized opcodes) have incorrect EA outputs, but are recognized correctly by the OP handler.

Completed

Otherwise, in the time allotted, all other requirements and deliverables appear to be met according the rubric and specifications. We have also added the additional functionality of `ADDI`, `SUBI` and all of the branch conditions.

Task Breakdown Report

Our organization and communication wasn't the best since around each week at least one of the team member were out sick so we were not able to follow our planned meeting time of meeting before class. We tried to substitute this by meeting more on discord but that didn't work too well until later weeks.

Our project completion was also very backheavy with us completing most of the work during week 4 and finals week since we had other project due taking most of our time the first few weeks.

We broke the work down into 3 distinct parts of E/A,I/O, and OP code and then each took a subpart like testing creation, documentation, and merge.Since all parts also required them to be both tested independently and together we worked all on it.

Task	**Team Member**	**Hrs.**	**Team Member**	**Approx. Contribution**
I/O	Sharanya	10	Adam	~33%
E/A	Adam	10	Liam	~33%
OP Codes except B.	Liam	10	Sharanya	~33%
OP Code for B.	Sharanya	10		
Testing Document	Liam	4		
Testing	Liam/Sharanya/Adam	15		
Merge	Adam	5		
Documentation	Liam	3		