2021

68k Disassembler

CSS 422 – Hardware & Organization

Final Project Report

Azer, Helina
Pairault, Daniella
Petrenko, Alex
Pham, Eric

Responsibilities:

Azer, Helina

Written Code:

- Indirect coding assistance for all written Opcodes
  - Screen sharing and simultaneous communication via Zoom
  - Excluding B-Condition Opcodes (BCC/BEQ/BGE/BLT/BRA)
- Indirect bug testing and improvements on:
  - File I/O / Opcodes / Addressing Modes
  - Screen sharing and simultaneous communication via Zoom

Non-Code:

- Recording internal testing logs for efficient code review
- Writing all progress reports
- Assistance in final report
- Maintaining an organize file management system via Google Drive

Pairault, Daniella

Written Code:

- Opcode definitions
- Opcode methods and subroutines
  - In addition; MOVEA and LSL/LSR/ASL/ASR memory shifts
- Opcode comments
- Subroutines to determine a set of bits
- Bug testing and improvements on
  - File I/O / Opcodes / Addressing Modes

Non-Code:

- Opcode table & flowchart to breakdown coding sections
- Code formatting for the DASM file
- Code organization for the DASM file
- Final Report & Final Presentation

Petrenko, Alex

Written Code:

- File I/O definitions
- File I/O methods and subroutines
- File I/O comments
- Bug testing and improvements on:
  - File I/O / Opcodes / Addressing Modes
  - Integrating Addressing Modes with Opcodes

Pham, Eric

Written Code:

- Addressing Mode definitions
- Addressing Mode methods and subroutines
- Addressing Mode comments
- Bug testing and improvements on:
  - Addressing Modes
  - Integrating Addressing Modes with Shift Instruction (LSL/LSR/ASL/ASR)

Collaborative Efforts

Written Code:

Fun 'ADD' welcome message

Bug testing and improvements on

- File IO / Opcodes / Addressing modes

Non-Code:

Progress reports 1 & 2

Source Control & Code Sharing

Given the time constraints and level of understanding, rather than utilize GitHub, the group used Google Drive. Early on in the project, Helina created a Google Drive and structured it by week and deliverable type (i.e. Project Versions, Progress Updates, Test File).

By the third progress report, the group further utilized the Google Drive to maintain a most current version for a member to utilize if they needed to work on something. Whenever a version of the final DASM file was pushed, if the edits were done independently, then the person who performed the edits would add an underscore with their name; if the edits were done collaboratively, then it would overwrite the most current version as the one to be worked on.

Integration

After everyone had a majority of the work, if not all, completed by the 3rd progress report's end, we immediately spent a few hours properly integrating the code together by connecting File I/O to Opcodes; then followed by Addressing Modes. This was done rather quickly within a matter of a couple of days and connected rather seamlessly as well. Further integrations were done through group meetings on Zoom on the main file.

Schedule:

Progress Report 1 – April 21, 2021

Delivered:	Description:	Goals:
▪ Progress report	▪ The group discussed the overarching purpose and goals to be accomplished throughout the course of the project.	▪ Biweekly meetings ▪ ¼ of the project complete ▪ Flowcharts and tables complete

Progress Report 2 – May 6, 2021

Delivered:	Description:	Goals:
▪ Progress report ▪ DASM flowchart ▪ DASM table	▪ The group had accomplished a majority of the I/O subroutines and other APIs documented. ▪ Partial test code had been written. ▪ We had not met our goals discussed in report 1	▪ Successfully read/output test file ▪ Disassemble simple instructions ▪ Successfully read a few addressing modes

Progress Report 3 – May 20, 2021

Delivered:	Description:	Goals:
▪ Progress report ▪ DASM_Final_ WIP.X68 file	▪ I/O routines had been mostly completed ▪ Opcodes completed: ○ NOP/RTS/JSR/LEA/NOT/DIVU/MULS/ADD SUB/AND/MOVE/LSL/LSR/ASL/ASR ▪ All addressing modes had been completed	▪ Complete the rest of the opcodes ▪ Clean up code/add comments ▪ Begin error handling ▪ Complete written test file ▪ Integrate all independent code

Final Submission – June 4, 2021

Delivered:	Description:	Goals:

▪ Progress report
▪ Source code
▪ Test results file
▪ Project report
▪ Presentation
	▪ The group began integrating all code following the week of May 20
	▪ After all code was integrated, the rest of the opcodes (excluding MOVEM) were completed
	▪ Bug testing began:
	  ○ File I/O had many improvements, could now read a file and output properly with string conversions.
	  ○ Output formatted (i.e. consistent spacing)
	  ○ Output addressing modes and addresses
	▪ Code formatting and code commenting
		▪ Everything had to be completed, with the exception of MOVEM; ***if provided the time after everything, then MOVEM could be worked on by everyone.***

Description

The overall design began by breaking the group members into roles that they were responsible for. Alex took responsibility of File I/O; Daniella and Helina took responsibility of Opcodes; and Eric took the responsibility of Addressing Modes.

We began by doing research on logical breakdowns of the project, so Daniella and Helina wrote up flowcharts and tables to help visualize the breakdown. This was a challenge early on given that we all felt new and inexperienced with Easy68K.

		INSTRUCTION FORMAT															
INSTRUCTION	PAGE	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	108	1	1	0	1	REGISTER			OPMODE			EA MODE			EA REGISTER		
AND	119	1	1	0	0	REGISTER			OPMODE			EA MODE			EA REGISTER		
ASL	126	1	1	1	0	COUNT? REGISTER			dr	SIZE		i/r	0	0	REGISTER		
ASR	126	1	1	1	0	COUNT? REGISTER			dr	SIZE		i/r	0	0	REGISTER		
BCC	130	0	1	1	0	CONDITION			8-BIT DISPLACEMENT								
BEQ	^																
BGE	^																
BLT	^																
BRA	159	0	1	1	0	0	0	0	0	8-BIT DISPLACEMENT							
DIVU	201	1	0	0	0	REGISTER			0	1	1	EA MODE			EA REGISTER		
JSR	213	0	1	0	0	1	1	1	0	1	0	EA MODE			EA REGISTER		
LEA	214	0	1	0	0	REGISTER			1	1	1	EA MODE			EA REGISTER		
LSL	218	1	1	1	0	COUNT / REGISTER			dr	SIZE		i/r	0	1	REGISTER		
LSR	218	1	1	1	0	COUNT / REGISTER			dr	SIZE		i/r	0	1	REGISTER		
MOVE	220	0	0	SIZE		DEST. REGISTER			DEST. MODE			SRC. REGISTER			SRC. MODE		
MOVEM	233	0	1	0	0	1	dr	0	0	1	SIZE	EA MODE			EA REGISTER		
MULS	240	1	1	0	0	REGISTER			1	1	1	EA MODE			EA REGISTER		
NOP	251	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1
NOT	252	0	1	0	0	0	1	1	0	SIZE		EA MODE			EA REGISTER		
RTS	273	0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1
SUB	278	1	0	0	1	REGISTER			OPMODE			EA MODE			EA REGISTER		

In terms of design and structure, after everyone had their respective parts, an integral and interesting part of the File I/O was in its ability to have a semblance of UI. Alex had thought of code within the specified guidelines to properly take in user input, and depending on the user input, it could print to the terminal in segments of 15 lines, print the entire file being disassembled in one printing, as well as exit if so desired. Another neat design used in File I/O was the logic behind converting hex to readable string, and almost how simple it is in hindsight. It operates as a backwards logic as one would use to convert string into hex but wasn't as easy to implement as initially thought. The implementation of the conversion of Hex to String provided a lot of usefulness in the implementation of MOVEM. The way in which Alex wrote the bits being shifted in a given register proved to be versatile in other sections of code which was extremely time-saving and helpful.

Daniella was most familiar with macros within the group since she played around with them in the previous homework assignments. As such, macros implemented which greatly reduced the total number of lines were printing macros to print an individual string. Another useful macro found along the way utilized the outStr (print string) macro which took in two string parameters. The purpose was to take in an opcode, and it's given size (byte, word, or long) and would ease the printing process. There was a desire to implement macros with the function to isolate a set of bits, but with the given time there was not an opportunity; looking at the "Determining Bits Section" in the DASM file there is an awful lot of similarly written code, so it seems as though it could be reduced to macros. Overall, the macros saved a lot of meticulous code that would have occurred otherwise.

With Eric on Addressing Modes, he learned how to utilize tables with the code to make locating the correct addressing mode much more efficient and organized. This greatly reduced the amount of code needed for addressing modes and made for easy debugging within the individual subroutines of the addressing modes themselves.

Specification

This program is intended to read and take in machine code and output the disassembled code to a console, as well as log it to a text file via the user's selection. To further explain, the machine code being taken in is taken in as binary data in memory and the program is to convert the data into a set of instructions, addressing modes, and operands all within a set of addresses. The disassembler is written entirely in 68000 Assembly Code and provides a minute user interface to print out the disassembled code in a desired manner.

Our program was designed in such a way to take in a long hex address to set a starting and ending address for the desired file to be read. The I/O Console then prompts the user with a set of numbers and depending on the user input, the console will either 1) display the disassembled code 15 lines at a time, 2) print the disassembled code all at once without pause, and 3) exit and end the program.

Following the user's input, if the user has decided to input, the console will output the disassembled code as follows:

```
Memory Address    Opcode            Addressing Mode/Operand(s)
```

Given that the program was not designed and written to disassemble all opcodes, if there is an invalid, or unknown, opcode to disassemble, the disassembled code will output as follows:

```
Memory Address    HEXd
```

'Hex' refers to the hex data at the given address *or* the hex-address depending on the opcode. The data is then followed by a lowercase 'd' to indicate data. If an instruction is disassembled properly but the addressing mode could not be disassembled, then the output would be as follows:

```
Memory Address    Opcode            Invalid EA, Invalid EA
```

The written opcodes and addressing modes included in the program are as follows:

Opcodes:		Addressing Modes:
ADD	LEA	Data Register Direct
AND	LSL	Address Register Direct
ASL	LSR	Address Register Indirect
ASR	MOVEA	Immediate Addressing
BCC	MOVEM*	Address Register Indirect with Post Incrementing
BEQ	MULS	Address Register Indirect with Pre Decrementing
BGE	NOP	Absolute Long Address
BLT	NOT	Absolute Word Address
BRA	RTS	
JSR	SUB	

**MOVEM is in working condition and outputs almost completely correct for the professor's provided tests.*

Test plan

The opcodes section was the only portion of the code that entailed a test-driven development model, however it only emphasized testing for opcodes and not for addressing modes. Consequentially, most of the opcodes tested were not included in the test file, and most of the testing involved utilized the professor's testing file. After testing was complete with the professor's test file, we began to compile our own test file. Most of our test file was almost identical to the professor's test file, however it was formatted a bit differently as well as we attempted to test different aspects that the professor's test file did not cover.

One example of our tests that accounted for output that the test file provided did not, were the branching opcodes. It appeared that within Professor Hogg's test file, the branching statements would branch to itself, so it was not a good test to determine how branches with differing displacements going to different address locations would appear. As a result, this made for more debugging as our initial code had actually been missing some minor details which impacted the outputted data.

Overall, our testing which differed from Professor Hogg's test file mostly differed to simply serve as an extension to the test provided. Given our structure and division of labor, the test file was not made until *after* most of the program was completed. Consequentially, this meant for testing that extended the provided test file to attempt to push our program's boundaries and find errors or account for instructions formatted in such a way that the original file did not account for. All of our tests can be found in the submission folder "Our_ADD.X68_Tests" for the original 68K file, or "Our_ADD.S68_Tests" for the S68 file to be loaded in and verified.

Problems:

There was only one opcode not entirely completed: MOVEM. Surprisingly with the limited time left, our group managed to produce MOVEM in such a way that it is in mostly working condition. With the professor's test, the MOVEM test using pre-decrement, the terminal outputs D4 and does *not* output A3, so our hypothesis is that the data and address register direction is switched. Given the time, we did not manage to fix it, however, we are pleased with the progress we made in such a limited amount of time with such a complex opcode.

In terms of bugs, one bug we had which is not quite a bug, but more so a feature, is that the program writes decimal values as hex, and not decimal. Meaning that the number is technically accurate, but given our conversion, in some instances, when the output is a decimal and not hex, our program defaults to outputting hex.

Another bug we were unable to address was for opcodes with an immediate long value, the addressing mode was unable to determine the long value and adjust correctly. As such, the following test in the professor's file:

```
0000045C    MOVEL       #$12, $12345678
```

Our program had output the line:

```
0000045C    MOVEL       #$0, $000121234
```

This is essentially due to the inability to adjust for long immediate values inside the addressing modes.

The final bug detected, was more so a feature than a bug; when the log text file is made, any uses of '#$8' is logged as a character. This particular notation, '#$8', is output as a backspace and used in our program to get rid of an extra forward slash in the opcode MOVEM, however, in the log file, both the forward slash *and* the backspace are logged where it otherwise displays normally.