CSS 422 Final Project: 68k Disassembler

Jason Fu, Nick Abel

Program Documentation

## Program Description

A basic description of our program is that it is a disassembler. Essentially, it takes a block of machine-readable code (in hexadecimal) and translates it to a human-readable output, in the same formatting that one would input opcode commands to make the program work. We had to break down the program into several parts to ensure that it could all be covered, the three parts being I/O, Opcode decoding and EA decoding.

We decided to write the test code first. This would help us better understand exactly what was expected of us, and it would be easier to write test code without bias of knowing what isn’t working. This took a little longer than expected, since we underestimated the amount of time it would take to compile all the information required to test.

Next we began to write I/O decoding. We felt that this would be the most important thing to write first, since it is how the user will run the program and it will be very easy to tell when it’s not working correctly. We had a lot of success with this part after writing the subroutines that convert hex to char and char to hex. We also decided that we could push everything to a single buffer line-by-line, rather than printing each part individually.

After I/O decoding, we worked on effective addressing decoding. This was fairly straightforward: we used the opcode’s size to determine what needed to be output to the buffer, making it a little easier to find out the exact output.

The last thing we worked on is Opcode decoding. We decided to do it sequentially (without using subroutines that decode the individual bits) so it would be easy to debug when we ran into errors. This also meant that we couldn’t copy/paste large amounts of code—most of the code was written fresh off the mind. Because of this, opcode took up most of the time. We reused subroutines from EA decoding to use for isolating the size of the opcode.

## Specification

Introduction:

* Displays welcome message

Input Handling:

* Prompts user to input starting address
* Convert ASCII address input to hexadecimal.
* Convert from opcode hex to ASCII
* Print error if starting and ending addresses conflict

Output Handling:

* Pushes decoded address, opcode and EA to buffer in human-readable format
* Stops display at 10 lines, prompts for user input to continue

Opcode Handling:

* Store remaining memory location into stack and pop from the stack to decode opcode in that memory location until reach end of stack
* Break down opcode by the first four bits to determine which part of the jump table we want to jump to
* Create a lookup table using index addressing models
* Implemented opcodes belong to each addressing model in the lookup table
* For each specific addressing models, check for opcodes that aren’t required and print error out to avoid crashing
* Check the size of an instruction (Byte,Word, or Long) if applicable
* Implemented various display subroutine (EA,Rn...etc…) to quickly jump to that subroutine and display opcodes accordingly
* Decode Effective Address by breaking it down and create a separate subroutine for each type of EA
* Decode word, long, and as well as immediate data in EA
* Decode Register by checking whether if it is An or Dn
* Display all Data and Address Register correctly

Ending:

* Prompts the user to begin again
* If yes, start over from Introduction
* If no, print thank you message

## Test Plan

The test file was the first thing that was written—it was essentially a thorough compilation of all possible combinations of opcode and EA. Since it was the first thing that was written, we had no problem making it very thorough. This also caused some problems: we didn’t know exactly which opcodes worked with which sources, destinations, opcodes or effective addresses. Fixing up this test code took a substantial amount of time. We also made sure to test with upper and lowercase hexadecimal numbers.

## Exception Report

## Team Assignments and Report

Nick:

* Wrote I/O
* Wrote EA Decoding
* In opcodes, wrote:
  + Group0000
  + Group0001
  + Group0010
  + Group0011
  + Group0100
  + Group1100
  + Group1101
  + Group1111

Jason:

* Wrote test code
* Added error checking to I/O
* In opcodes, wrote:
  + Group0101
  + Group0110
  + Group0111
  + Group1000
  + Group1001
  + Group1010
  + Group1011
  + Group1110