CSS 422 Dis**assembl**er Project

Guiragos Guiragossian

Jonathan R. Hendrickson

Omar Nevarez

Dylan L. Thibault

University of Washington – Bothell

# Team

A four-person team worked on this project. Each person had a dedicated role in the group and was integral in completing this project. Jonathan Hendrickson worked on identifying opcodes. Guiragos Guiragossian was in charge of decoding opcode instructions. Dylan Thibault implemented the I/O for the program. Omar Nevarez was in charge of testing.

Collaboration and source control was the first objective the team looked to solve. The team agreed it was best to use GitHub for source control. After members were done with a section of code, they would push the change into GitHub. Due to GitHub not supporting continuous integration for 68k assembly code the team set ground rules for pushing commits. Before pushing commits developers needed to communicate with the rest of the team what changes were being made in case those changes affected another developer’s progress.

# Project Description

The program takes 68k machine code and disassembles it into assembly code. There are 100’s of 68k assemble code instructions; this program focuses on disassembling only 20 of them as well as only 8 addressing modes. The program is written in 68k assembly code.

The program begins by asking the user to input the beginning and end addresses in memory using hex. The user input is checked for possible errors. Then beginning address is stored in A2 and the ending address is stored in a memory address location. Then the main loop in the program uses A2 to continually obtain the next word to decode. After each opcode is decoded and the next word is pulled from A2 there is a check to make sure that the program is still within the range that the user inputted.

A trie structure is used to help identify the opcodes. This is done by incrementally testing the most significant bit on the operation, each bit can either be a 0 or 1. As each bit is tested the program is able to narrow down the possible opcode until there is only option available. After identifying said opcode the program branches into decoding that opcode. There is a subroutine for decode all 20 opcodes.

Each subroutine takes the machine code and through bit testing, shifting and manipulation each instruction is decoded to show the opcode, size, and addressing modes. Depending on the addressing mode the program will then pull additional words from A2, as required. When each instruction is fully decoded, the program increments A2 by 16 bits and returns to the main loop to identify the next opcode.

# Project Specification

This program disassembles machine code into 68k assembly code. It can identify 20 opcodes.

* MOVE
* ADD
* SUB
* MULS
* DIVU
* NOP
* LEA
* AND
* NOT
* LSL
* LSR
* ASL
* ASR
* BLT
* BGE
* BEQ
* JSR
* RTS
* BRA

When reading the machine code, the program can decode 8 addressing modes.

* Data Register Direct
* Address Register Direct
* Address Register Indirect
* Immediate Addressing
* Post Increment Address Register
* Pre Decrement Address Register
* Absolute Long Address
* Absolute Word Address

This program has 3 major components: the I/O, identification of the opcode, and decoding the instruction. The program is run by first loading a S-Record file (.S68) into memory, this is where the test machine code is stored. Then it prompts the user to input a beginning location in memory and the ending location, in hex. The input from the user is received as string of ASCII characters and in order for the program to utilize the input the ASCII characters need to be converted into hex characters. After conversion the memory addresses are stored in their proper locations. As each operation is disassembled the results are printed directly to the console. If the program encounters an invalid operation it is printed to the console as data. When the program reaches the ending address location it terminates.

## Test Plan

Testing for this project was very extensive, over 300 test cases were created to cover as many possible permutations in each opcode. There are 20 opcodes with 3 possible sizes, byte, word, long, and 8 different addressing modes.

With so many test cases needed for full coverage, testing was done methodically. Going through each opcode one by one and testing each addressing mode one by one. As a group we decided to work on the easier opcodes first in order to ensure that we were on the right track. Below is the testing schedule the group performed.

Table

Description automatically generated

# Problems

The first problem our group encountered with this project was creating an effective system for commenting and formatting a file in 68k assembly. Unlike modern programming languages, we had to operate out of a single file, and bugs in one person’s code could easily break the entire program. In our first two meetings, we devised a system of comments using headers (with names, so we can find who implemented that particular section of code), as well as a system for pushing changes (create branches if the code doesn’t work yet, check in before pushing to main).

About halfway through the quarter, a problem was found with the use of a trie structure for identifying opcodes. When program was first designed, it was decided to use to trie structure for quick and easy identification of opcodes, however one of the main things we overlooked was error checking. When using the trie structure as soon as the program found a match for the opcode it moved on to decoding the instruction, without verifying to make sure it’s a proper opcode and data, or an incorrect opcode instruction. The program was modified to verify the opcode is valid before moving on to the decoding.

Lastly, a few of our group members were busy with their other classes and outside interests, so a significant portion of the program was left until the final few weeks. We all had to work quickly to identify bugs and prepare a capable program in time for the presentation. In this, we were unable to get a properly formatted MOVEM working, though being extremely close.

# Schedule

Below is a list of the task and the schedule of when they were completed and by which team member.

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Description** | **Date** | **Owner** |
| I/O Ask User for Input | Ask the user to enter the beginning and ending addresses in hex and store starting point in A2 and ending point in a memory address. | 6/01/2021 | Dylan |
| Convert ASCII to Hex | When user inputs memory addresses Easy 68k stores it as ASCII characters and not hex values. Need to change ASCII characters to hex. | 6/01/2021 | Dylan |
| Main Loop | Loop through each Word between given addresses. Use A2 to track current address | 6/01/2021 | Guiragos |
| Identify Opcodes | Read input hex bit by bit to determine the correct opcode.  Once determined call proper decoding subroutine. | 5/10/2021 | Jonathan |
| Decode Instructions and addressing modes | When the proper subroutine is called the instruction needs to decoded. There are 8 addressing modes to account for. | 6/04/2021 | Guiragos |
| Create Test Cases | Create a comprehensive set of test cases for all 20 opcodes and 8 addressing modes. | 5/23/2021 | Omar |