Team Members: Evelyn Mahasin, Kay Phan, Khuzaima Mushtaq
CSS 422: Tyler Folsom

Team Rocket: 68K Disassembler Documentation

Program description:

The Team Rocket 68K Disassembler is designed to fulfill all the requirements of the project, Including disassembling all the opcodes, and all the effective addresses. Our design philosophy was to do all 3 parts of the program separately and independent of all other parts so that we can test from ground up and integrate once all the parts are functioning. This design also required us to assume that the part each of us were not responsible for worked as intended when integrating the project. We did not use any routines from 3rd party software in our project, all the coding was done by the members of the project.

The project was divided up into 3 parts, as recommended by the professor. We had a person for Opcodes (Evelyn), I/O (Kay), and effective addresses (Khuzaima). Our core design was based off the interaction between these 3 parts. The I/O would initialize the project and call Opcodes, Opcodes would figure out the opcode and non EA parts of an instruction and call EA, EA would return the EA and opcode would return the whole instruction, and I/O would print and end the program. This design worked really well for us in debugging because we could spot where things were going wrong and fix them. We used flags to indicate when something was going wrong and where. We also used flags to indicate good instructions vs bad instructions. Some of the limitations of the program are that there are read errors that the program can run into while reading memory from starting at an odd bit or ending up reading an odd bit.

Specification:

1. Reads instructions from a start place in memory to an end place in memory. The minimum address is $6000 and maximum address is $FFFFFF.
2. Decode all the required Opcode
3. Decode all the required EA’s
4. Print the decoded instructions to console 30 at a time
5. Ask user to continue when they are ready
6. For Opcodes and EA not required print Data and the hex value of the instruction

7. For invalid EA print Data and hex value
8. Prompt the user to disassemble another data image or exit when done with the current segment

On opening files and using the disassembler:

You must load the program into memory *before* running the disassembler. A program that you want to disassemble should be opened in the manner as follows:

1. Make sure that the program you want to read has an origin that starts at an address after address $6000.
2. Create an .S68 source file for your program.
3. Go to file > open data, then open your .S68 source file.
4. Run the disassembler.
5. Enter the address of the program into the disassembler.

Test Plan:

Our Team coding standards were based on Register usage as well as naming conventions for subroutines and variables. We Assigned register usage between us depending on our needs so that we do not overwrite each other's data and step over each other's toes. We tested our program initially separately and we tested our program each step of the way as we integrated our parts. For example, for EA we tested if EA worked in isolation from opcodes, then when we combined EA with Opcodes we tested them together. Then as we combined Opcodes with I/O we tested the whole thing together. Our final test was decoding the demo provided on the project page on canvas.

4) **Exception report**:

There have not been many exceptions in our project. We have one 'bug' where the program is unable to read from an odd memory address. We couldn't not fix this as we believe it's a limitation of the 68K. There are a few quality of life things that we could add so that the program is user proof if the starting instruction is an odd address it does not work. Otherwise, we believe our program is solid.

5) **Team assignments and report**:

The Team was based on roles. We had an Opcode person (Evelyn) an EA person (Khuzaima) and an I/O person (Kay). The percent distribution for our code is Opcode (40%) EA (30%) I/O (30%). We decided these percentages as a group.

I/O person was responsible for converting between ASCII and hex, accepting user input for start and end addresses, checking flags to print the buffer or print error data, and updating the pointer. Opcode person was responsible for decoding the opcode, loading it to buffer, updating the flag, and updating the pointer. EA person was responsible for decoding EA fields of each instructions, loading it to the buffer, updating the pointer and the flag.