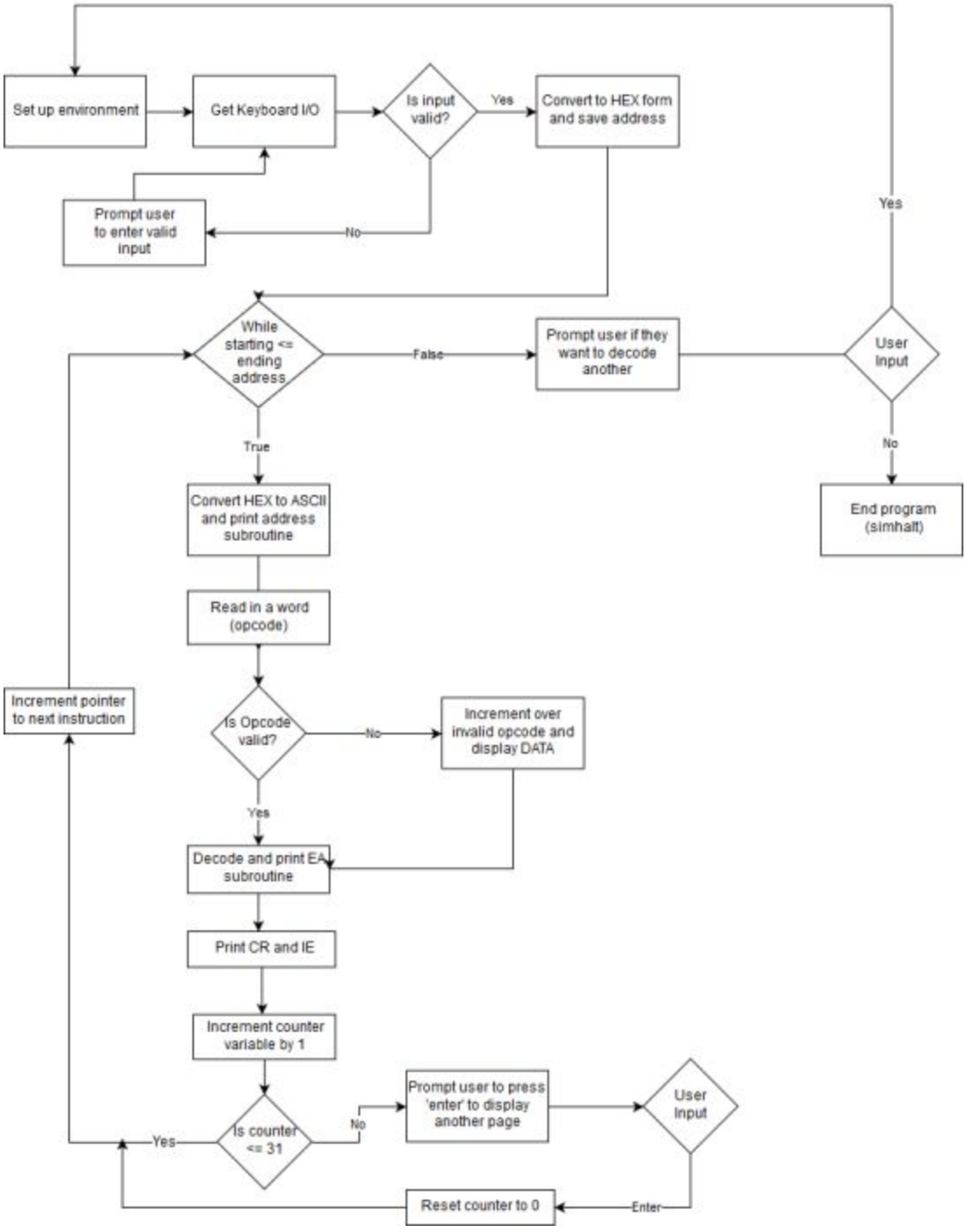
Team Low_Expections
Members: Tyler Do, Ji Kang, Princeton See
CSS 422 Disassembler Final Report
Prof. Yang Peng

Program description

Our main design philosophy for our disassembler consists of checking the bits of each opcode that are always the same regardless of the size or effective addressing mode that the instruction uses. The flow of our program is as follows: We would set up the environment (print a welcome message, load appropriate strings, display, and switch trap tasks). Get user input, validate (and keep doing until the input is valid). If the input is valid, we move onto converting the ASCII representation of the starting and ending addresses to their numerical counterparts. From there, we iterate through memory from starting to ending address, taking in a word of data at a time. Each word (4 hex characters) are then shifted and compared to key identifying bit patterns to determine, in a general sense which instruction it MAY be (i.e MOVEM and LEA have similar bit compositions but some specific bits set them apart). From this subroutine, it may branch to various other subroutines specifically for the families of instructions. In each specific subroutine, it'll double check if it is indeed the instruction we want and recognize and print the opcode, size, and formatting (i.e tab, newline, etc.). If it is not the opcode we want, we won't know exactly which opcode it is but we have a sense of the composition of its bits and their significance and send it to our invalid opcode handler which will print and increment over the invalid instruction accordingly. Everytime we print something to console, we increment a counter variable by 1 and check if we've printed 31 lines of output (one screen) and allow the user to press 'CR' to display the next (resets counter to 0). The program would repeat this

process until the current address is past the ending. At this point, the program would ask the user if they would like to disassemble another portion of memory and would repeat or end depending on user input. In order to keep our program uncluttered, we divided the program into separate files that performed the aforementioned functionality. For example, our main “Disassembler” file contained the I/O logic as well as primary disassemble logic and EA logic. Our “Disassembler_subroutine” file contained the logic for all the required opcodes and our “subroutine” file contained any extra subroutines we needed to use (i.e printing the size, newlines, addresses, etc.). One keynote for our disassember is that we took up address $0 to $1CE4 so all file input data should be org’d at addresses past that point. There are a couple algorithms that we are proud of. To name a few, we made a subroutine named “DEST_TO_SOURCEEA” that utilized shifts, swaps, and more shifts to flip the bit ordering of the destination EA to match the source EA, so our DECODE_EA subroutine would be able to handle both. Another subroutine we want to showcase is inverting the bits of a word by using shifts, AND, and OR in “MOVEMPRE_DEC” to allow us to utilize the same MOVEM subroutine for recognizing the register ranges. From there, the last one we want to showcase also has to do with the MOVEM instruction. It was a subroutine to format the register ranges in MOVEMPOST_INC (i.e D1/ D3/ D5-7 would show has that and not D1/D3/D5/D6/D7). In the entirety of the project, there were no “canned” or “appropriated” subroutines used from 3rd party sources. We, team Low_Expectations, take intellectual property of all work done in this submission.


Set up environment
Get Keyboard I/O
Is input valid?
Yes
Convert to HEX form and save address
Prompt user to enter valid input
No
Yes
While starting <= ending address
False
Prompt user if they want to decode another
User Input
No
End program (simhalt)
True
Convert HEX to ASCII and print address subroutine
Read in a word (opcode)
Increment pointer to next instruction
Is Opcode valid?
No
Increment over invalid opcode and display DATA
Yes
Decode and print EA subroutine
Print CR and IE
Increment counter variable by 1
Prompt user to press 'enter' to display another page
User Input
Is counter <= 31
No
Yes
Reset counter to 0
Enter

Specification

Once the program launches, it first displays a welcome message to the user as well as instructions on how to input memory addresses. The **starting address** is intended to be the starting address of the first instruction and the **ending address** is intended to be the starting address of the last instruction. After the user has entered in the starting and ending memory addresses that the user wants to see, the console screen is erased and 30 opcodes with their address and EA are displayed on the console in the format of [ADDRESS] [OPCODE] and [OPERAND]. Any unknown opcodes the program encounters will be displayed as "DATA" and will display $ value of the instruction or data in WORD-size. This assumes that the user has put in a file for the program to disassemble before the program asked for the starting and ending address. If the user does not put in a data file for the program to disassemble, then the program will attempt to disassemble whatever is at the range they indicate. If the memory location they specified does not have anything in it, the program will just print the address and a newline to indicate the absence of data. Should the user enter in an invalid address or format it incorrectly, the program will inform the user to enter the address again or choose to quit the program. Once the opcodes are displayed for the user (one page of output), the program will prompt the user to press the 'enter' (carriage return) key again to display more opcodes. This will continue until the user reaches the end of the specified address. Once the user reaches the specified address, the program will ask if the user wants to continue by either entering in "Y" or "N" (not-case sensitive). Should the user enter in "Y", then the program will reset and display the same message from the start and essentially restart the entire program. Should the user enter in "N", then the program will display a "goodbye message" and close itself. Should the user either

neither of those things, the program will prompt the user to pick one of the options until it matches one of the specified options.

Test Plan

Our plan to test the program mainly revolved around using the debugger that Easy68k provides to the user. In order to see if a certain part of our program was working, we would manually step-through the code to see if the correct values were being shifted and the results were ending up as expected. This method was used most commonly when the group was coding the logic for the program to recognize and print out the required opcodes. For example, if we were trying to test if ADD was working correctly, we would set a breakpoint in the program where the ADD logic started. From that point on, we would skip to the breakpoint and then start tracing in through line by line whilst looking at the D0-D7/A0-A7 values given by the program. If we were unsure if the program was shifting correctly, pen and paper would be used in order to see if the program was getting the intended result. Since each member worked on different opcodes, each member was responsible for making their own test file to see if their opcode was being correctly processed. We would essentially do incremental development individually then meet up to successfully merge our individual subroutines into our main one. This consisted of creating temporary test files with a plethora of variations of each of the opcodes completed and run the test. We would essentially redo this process until our program was successfully able to decode every single one listed. As for testing the main completed program itself, the group used the demo_test file as well as individual test files in order to see if our program as a whole was running as the description specified. This involved running through all the given opcodes in the

test file and seeing if the addresses and opcodes in the test file matched up with what the console outputted in our program. There were a few basic coding standards we followed in order to complete the project. First, we made sure all labels and subroutines had simple and recognizable names to make sure each of us knew what that particular section of the code did. Also, if the subroutine or section of code had specific/complicated code, then we made sure to add comments so everyone in the group understood what was going on in the code. Finally, we made sure to keep our coding branches separate in order to not mess with each other's code when trying to complete the program.

Exception report

One of the only minor problems our program has is not covering the right side of the M68k CPU opcode table in terms of being able to recognize that it's invalid opcode AND to increment over to the EA/data and print it out. Right now, our program will recognize and skip most opcodes/illegal instructions and print out the desired output but due to time constraints could not get the program to catch all of the M68K opcodes. Another one of our other exceptions to our disassembler was the ability to recognize invalid effective addressing modes. For the project, we only tested with the 8 effective addressing modes and did not error check for any other modes that may be thrown in.

Team assignments and report

While dividing the work up, the team divided the work so that an even amount of tasks would be assigned. For opcodes, Ji Kang took care of the entire MOVE family (MOVE/MOVEA/MOVEM/MOVEQ) and the branches (JSR, BRA, BEQ, BLE, BGT), Princeton worked on SUB/MULS/DIVU/AND/OR and the shift family and Tyler worked on the ADD family (ADD/ADDA/ADDQ/NOT). The remaining opcodes were split amongst the group. Beyond the opcode, Princeton focused on completing the IO of the project, Jayden also helped out on IO and debugging and Tyler worked on error handling and the EA. If the coding was broken down into percentages, it would break down like: 40% for Ji Kang and 30% for Princeton and Tyler.