Disassembler Documentation

Team Awesome (Olga Rocheeva, Dwina Solihin, Peter, Stanton)
CSS 422 - Spring 2017
May 31, 2017

Program description

Our program successfully is able to disassemble through different test codes we have tried it with. When first starting this project, we met up many different times to determine a final flow of how our disassembler will run but, also, how we plan to integrate all the separate parts together. We planned on having each member work on their respective section of the disassembler then we will begin integration and testing once we all finished our sections. We saw that there needed to be the use of jump table for the OpCode and EA portion of the disassembler, masking and a buffer to store all the information of each line.

As seen with the flowchart below, the welcome message appears, then we have the system prompt the user to input a start address. That start address then gets inputted into an input_buffer where it will be stored in an address register. Then the system prompts the user to input an end address that is gets put into an input_buffer where it will be stored in an address register, as well. After each input, we have the system do a check to ensure that the addresses are valid meaning that they do not end in an odd number and that the user inputted the addresses in order (smallest to biggest).

From here, the system jumps into the OpCode section where it begins to read each line of the test code one by one and disassembling that information which is then stored in a buffer. In the OpCode section, the system first checks the address of the OpCode it is on and then prints out the address. Then, the system continues on with the OpCode check to see if it is a valid OpCode and its size. If all the test are valid for this OpCode, then the information is saved onto the buffer and gets called into the EA section.

In the EA section, the logic of the system is similar to the OpCode section except the checks it goes through are to check the order of the source and destination, what is the mode and the register for each source and destination. From here, the information from OpCode and EA have been thoroughly checked and then saved in the buffer.

The information stored in the buffer is then pushed back to the IO section grabs the buffer and uses a print address subroutine to display the function on the console. After jumping to the print address subroutine, it returns to then do a check to see if it reaches the end of reading address stored in A4. If the address matches, then the systems lets the user know that the test has ended and prompts the user to determine whether they want to continue or not. If they do

then the systems prompts them to type in another start and ending address but if not then the program ends.

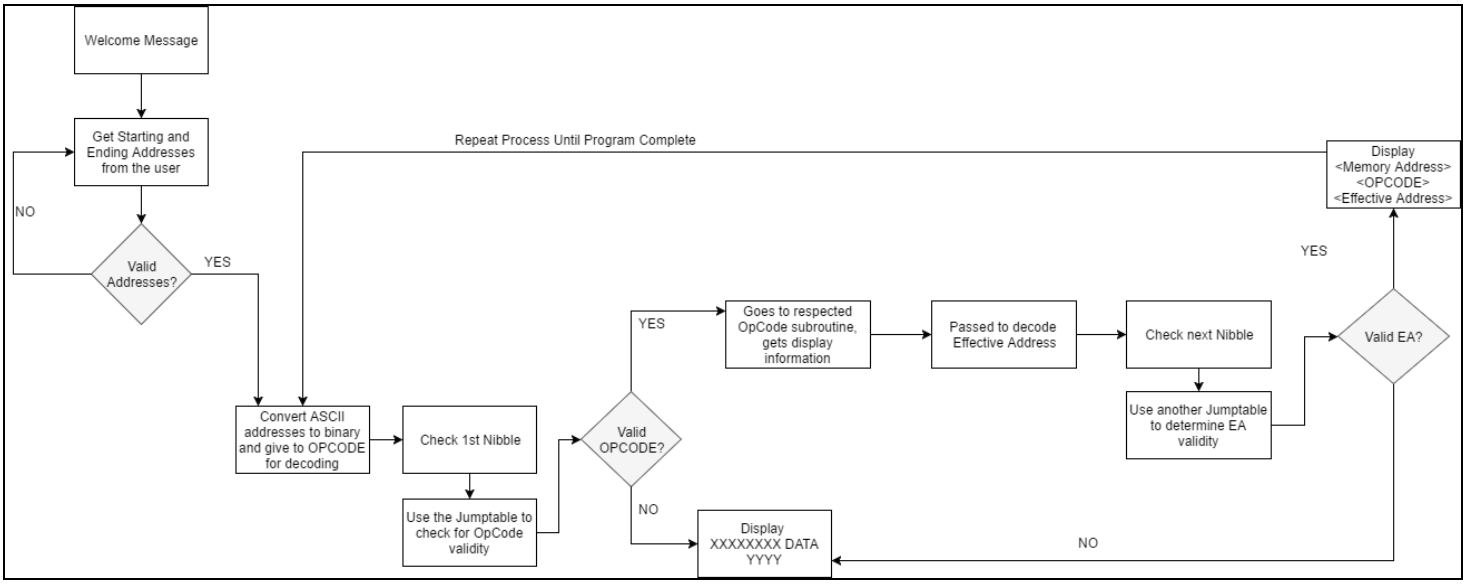

Figure 1.1, Flowchart of Disassembler Flow (version 2)

How to run our code:

1. Go to ‘Main.x68’ and click run button
2. When the .S68 file pops up, go to ‘File’, then ‘Open Data’ and choose the TEST_CODE.x68 file.
3. Then run the code
4. Starting address should start at $6000 and ending address should be at $6500

Specification:

With this disassembler, we are have our program running so that:

1. The user will enter a starting and ending address
2. A line counter will, also, be started so that the screen will keep track of how many lines have been printed
3. Those addresses will get converted from ASCII to hexadecimal
4. The converted addresses will get saved into A2 and A3, respectively, to be used with the printing out of information
5. From here, buffer for the OpCodes will get put into A6 where it will be holding information of the disassembled OpCode and later on EA
6. Goes into the jumptable used for Opcodes to checks for which OpCode it is based on the first 4 bits of the machine language

7. From this, the system checks the validity of the OpCode and stores that information into the buffer in A6
8. System prints out the OpCode it has decoded
9. The system then goes into EA to determine the source and destination mode and register
10. In the jump table for EA, the program goes through the next bits to decode the source and destination and stores the information in the buffer in A6.
11. System prints out the source and destination in correct order
12. At the start of the Main_LOOP, the program will increment the count of lines that have been printed out on the screen and check if the lines equal 29.
13. If they do then the system will branch to the IO_PauseOutput subroutine where it will display a prompt for the user to click enter if they want to continue
14. At the end of the IO_PauseOutput, it will branch back to MAIN_LOOP where, after the linecount check, it will check if the current address it is on is greater than or equal to the ending address the user inputted at the start of the program
15. If the ending address is not equal, it will continue to loop until it is equal
16. If it is equal then the program will branch to the IO_EndProgram subroutine where it will prompt the user if they want to continue with another test
17. If they do then the program will go back to the start and prompt them to enter a new start and ending address
18. If not, then the program will end

Test Plan:

In order to test our code, we used 3 different test codes: one we made ourselves and two from other groups. As we finished IO, OpCode, and EA, we took the time to individually test each part before beginning to integrate them together.

The first test started was with IO where we made sure that the system was printing out the welcome message and the two prompt to get the user to type in a start and ending address. From here, we tested to make sure that each address was stored in the correct address register where it will be later used for comparing and checks. After that, we tested to ensure that IO was able to pause the program after 29 lines had been shown on the console and that the program will branch to the end program subroutine once it has read all the information from the starting address to the ending address. From here, IO was completed and we began testing OpCode.

The second test was with OpCode where we had to make sure the system was going through each OpCode correctly and printing out the information correctly. Within the OpCode test, there were tests to check the validity of the OpCode meaning that it was an OpCode that was

invalid then it would branch to the invalid OpCode subroutine, print out the invalid information as XXXXXXXX DATA YYYY and then continue on with the program. When it was correct, it will print out the respective OpCode.

The third test was with EA where we had to make sure the system was reading the information about EA correctly and printing it in the correct order since the order of source and destination vary based on OpCode. In this test, we had to check if each different effective address would print out correctly. With this test, there were parts such as a closing or open parenthesis would not print out. These were tests that we had to make sure the correct information would be put into the buffer but also, the information would be put into the buffer in the correct order.

Exception report:

One thing that we were not able to fix in our program is that we were not able to fix in this program is making sure that the system can only read in long-word addresses. With the system, we implemented the system so that both word and long-word addresses can be used for the start and end addresses. One of the final goals was to include that check so that the output of our disassembler will have a similar look between each OpCode and its EA.

Another issue we encountered in the program is that when we type in 'yes' at the end of the program to go through it again, the formatting on the screen becomes weird and prints out horizontally instead of vertically. The first time it runs, everything prints out correctly but anytime after it runs, it prints out weirdly.

Another issue we encountered with this program is that with some of the immediate addressing, it prints out the dereferenced address one after another until it hits an address that has an OpCode that we can print out. That is why in our code, you can see some lines where the address seems to print out 2 or more in one line causing it to not be in sync with its neighboring code.

We managed to get the Branching instructions work for about 75% of the time. We were not able to implement MOVEM under the OpCode portion of it.

Team assignments and report:

Team member	What they did	Percentage of coding
Olga	OpCode	40%
Dwina	I/O & Testing & Documentation	20%
Peter	EA	40%