Disassembler Test Plan

by Team A: Orpal Nagra, Yuunbum Lim, Gihwan (Kris) Kwon

Testing

Before our team jumped into the phase of testing, we tried to analyze the program that we are writing, which is a disassembler. As the name of the program already represents the purpose of itself, we tried understand how it works and is used for, and what APIs we can use. After we reviewed the final project specification, it helped us to understand all the features of the disassembler has and what we should have for our testing. Finally, we decided to have a testing code for the tests.

The Test Strategy

1. Define scope of testing for instructions that our disassembler should support
2. Identify testing type of instructions such as identifying what size CMP instruction can handle like byte, word, long.
3. List out all possible instructions set
4. Test and modify the testing instructions set.

I/O

For the first phase of running a disassembler, which is an I/O part, we had to check whether user puts proper starting and ending addresses. To test, we tried to put different sets of address such as 6000, 6500, and 7000. To test this phase, we implemented it in a way where whenever user puts address other than 7000 for starting address, the program keeps asking until the user puts the right address, which is 7000. And for ending address, we set the limit according to the given specification.

Opcode and EA

For the Opcode and EA, since we only had to test whether our disassembler convert a memory image of instructions and data back to human-readable, 68K assembly language and output this disassembly to the display. What we expected to have for testing is just a bunch of various instructions, which we thought this would be very simple. However, there were few cases that we should have consider ahead before we writing our EA parts, because 68K convert some of instructions differently. For example, we had a instruction of “ADD #1, D1”, and this instruction is legal according to the 68K manual, whereas The 68K converts it into ADDI instead of ADD because it is actually adding an immediate value to data register. Even before we figured this out, we had to debug to figure out where it’s causing problem. Also, to indicate whether the program makes an error or not, Whenever we face an error, we implemented the OP and EA code to raise an exception by printing “DATA $XYZ”. Thus, we had to modify our testing plan by modifying the testing code as we implement the EA Code for easier debugging.

The process of testing worked in a way that we run the disassembler with the testing code we wrote as we implemented and compare the output with the expected output. We were able to

find all errors and fixed them through this way except those errors that 68K makes such as what I mentioned above, for example, reading ADD instruction as ADDI.

Few Errors we found during the test:

1. 68K reads ADD as ADDI when source operand is an immediate value. (For all size: byte, word, and long)
2. 68k reads SUB as SUBI when source operand is an immediate value. (For all size: byte, word, and long)
3. 68K reads ADDA with immediate value on source operand as SUBQ.