Adam Deehring

Bazen Nega

**Suspicious Disassemblers - Disassembler Project**

**Test Plan**

As we were implementing our disassembler, we were continuously moving opcodes into specific locations in memory and then reading from those memory locations via a loop. The start and end locations of this loop were hard coded into variables in our variables section, which we would then use for initializing the loop (we moved the start address to A4) and validating we didn’t cross the end address upper bound (make sure A4 never goes beyond the end address without branching to the end). A4 pointed to the test opcodes in memory. This was done before we had begun reading our config.cfg file input, and was used until we began our final tests of our disassembler. This was done primarily so that we could temporarily skip the importing of the Start and End locations from the config.cfg file.

When we believed that we had completed implementation of an operation code, we made sure to check to see if our beliefs were correct. We checked this by giving our disassembler an operation code for each effective addressing mode, regardless of which effective addressing mode was legal for that operation code., and checking to see those operation codes we gave our disassembler resulted in output that matched the specifications of the Motorola 68K Manual. We paid special attention to making sure that addressing modes that weren’t allowed for a given instruction in either the source or destination position were printing “DATA”.

During implementation, in order to be able to check functionality at every step, we were using function called “TrapTask14” (no longer in the final code), so that we could print out the ASCII strings of the disassembly instructions as we confirmed what each field in an opcode represented. This resulted in a problem where if we determined that a field was unrecognizable, and would need to print “DATA”, we would have already printed the representations for previous fields, and would get outputs like this - “MOVE D1, DATA”. This was helpful enough early on because we would know instantly if there was an invalid addressing mode/opcode.

At first, we didn’t realize that we were required to use TrapTask13, but once we realized this, we worked to complete our code for a string buffer that would build the output ASCII string in memory as we progressed through the operation code. We were able to simply replace all instances of the usage of TrapTask14 with our String Buffer subroutine, and then call TrapTask13 at the very end of each function dedicated to disassembling a specific operation.

When we were using TrapTask14, we were also using Trap Task #15 (task 15) in order to print numeric values to the screen from D1. But with the revelation that we were required to use TrapTask13, we could no longer use this method, and thus we began creating a Hex to ASCII function that would allow us to add the numeric value words that came after an opcode to the string buffer after a subroutine handled conversion.

We needed a sufficient method of testing our disassembler after we implemented all of our supported opcodes. Our testing method needed to be a comprehensive test that would be able to realistically implement given our time constraints. This motivated us our own test file that consisted of groups of similar instructions in an S68 file, that could be loaded into the main disassembler X68 executable. We would only need to modify our config.cfg file to match the location the test file “ORGed” at as a starting address, and make the end address the last line of the test file.

Each group of similar instructions consisted of different effective addresses for each instruction (i.e. one block group consisted of all effective addresses for ADD and SUB, another for MULS and DIVS, etc). In the event that we encountered a bug in our tests, we would change the start address in our config file to be at the line that caused the bug or just above it (in case the instruction immediately before and the manner in which it affected register values somehow interacted oddly with the state of registers for the next instruction). This would allow for simple code tracing to debug more efficiently. These measures aided in making sure our program functioned properly.

We knew that if we were to create an extensive test file that would test a multitude of different types of commands, we would need to create one that would be similar to the one that is going to be used when grading our disassembler. Thus we made sure to test not only the commands that we supported but also ones that were not required (ones we expected to go to “DATA”).

Our primary motivation behind developing this was because if we only gave the disassembler the commands we explicitly supported, we would only be supporting half of its functionality. The second half of the specification is printing DATA when we run into command that is unsupported. Thus we will know we have completed our disassembler if our output file has reproductions of the commands we supported, with DATA in the places where we didn't.