CSS 422: Hardware and Computer Organization

Disassembler – Test Plan

Team – Single Precision REEE

## Overview

For our disassembler project, we used extreme programming and a test-driven approach. Test files were created to extensively check every single supported opcode and its effective addressing mode by loading the file into the disassembler. Additionally, we checked Null and Burger textbooks as well as Professor Nash’s slides to find a variety of special cases. Our opcode and IO team members created a test environment for supported and unsupported opcodes.

The use of logging was extremely helpful not only to validate the correctness supported instructions but also to find minor bugs or defects that were noted in the exceptions report. The team relied heavily on EASy68K for debugging purposes and for finding the line of code that was generating issues. The team made sure that we do not stack smash each other’s subroutines by backing up the data register and address registers that are being used in each subroutine. **Figure 1** highlights a sample portion of the test plan

|  |  |
| --- | --- |
|  |  |

**Figure 1: A sample portion of the test.X68 file**

## Test Phases

**The table below demonstrates the test cases that were implemented in the IO phase:**

|  |  |
| --- | --- |
| **IO Test Cases** | **State** |
| Checking if any or both the starting and ending address isodd | Invalid |
| Checking if the start and ending address is even | Valid |
| Validating that the start address or the end address is not negative | Invalid |
| Validating that the start and ending address are positive | Valid |
| Verifying that the start address is not greater than the end address | Invalid |
| The starting address must be less than the ending address | Valid |
| Checking if the addresses provided are more than 8 bits | Invalid |
| The addresses provided must be 8 bits | Valid |
| Calle-saving the used register in IO subroutines | Completed |

For the OP code section, we started identifying the largest number of distinct bits that can be recognized. We created a private GitHub repository, highlighted in **Figure 2**, that contains a spreadsheet with a description on how each opcode bits should be divided as well as its effective addressing mode.

|  |
| --- |
|  |
|  |

**Figure 2: Displays a private GitHub branch for testing purposes**

**The following table highlights test cases that were achieved in the OP code phase:**

|  |  |
| --- | --- |
| **OP code Test Cases** | **State** |
| Identifying the first nibble of each opcode | Completed |
| Testing against the loaded test file and making sure that it is outputting matching instructions | Completed |
| Testing size prints for each opcode | Completed |
| Testing identification for every supported opcode | Completed |
| Testing identification every unsupported opcode | Completed |
| Validating proper opcode prints after each identification | Completed |
| Ensuring buffer loading between IO, OP, and EA roles is properly incremented | Completed |
| Genera validation testing for all supported opcodes | Completed |
| General validation testing for all unsupported opcodes | Completed |
| Ensure proper routine call depending on whether IS\_VALID bit is set to 0 or not | Completed |
| Ensuring each role of IO, OP, and EA call the proper routines after either an INVALID opcode has been found or a valid opcode that has been processed. | Completed |
| Validating INVALID opcodes that would never result in G\_BUFFER which are being loaded with the respective opcode print | Completed |
| Testing EA parsing for data constants such as (MNEMONICS, FIRST\_3, SECOND\_3, THIRD\_3, and FOURTH\_3) | Completed |
| Testing bit differences between MOVE and MOVEA opcodes | Completed |
| Testing size parsings for 2 bit size fields (bit 7-6) and loading them to OP\_SIZE | Completed |
| Testing size parsings for 1 bit size fields (bit 8 or 6 for ADDA or MOVEM) and loading them to OP\_SIZE | Completed |
| Callee-saving all major routines | Completed |
| Testing especially tricky inputs that seem initially valid such as SUB.W D1,A1 which is 92C1 which becomes SUBA (unsupported). | Completed |
| Making custom bit shifts or bit masks for specific cases such as MOVEM or EXG and MULS to differentiate between very similar commands. | Completed |

**The following table highlights test cases that were achieved in the EA phase:**

|  |  |
| --- | --- |
| **EA code Test Cases** | **State** |
| Testing that the mnemonics are valid, in case the OP code is not handling the valid bit | Completed |
| Testing the instructions that do not support immediate values (e.g. JSR which has an immediate value as destination) that would ultimately throw an error if not handled properly | Completed |
| Testing each instruction would look into a word or a long should there is an absolute word or an absolute long code | Completed |
| Testing the immediate values that would be correct regarding both its HEX values or decimal values | Completed |
| Testing the instruction bits that could be valid (e.g. if MOD = 111, there can’t be REG = 111) | Completed |