# Test Plan

### *Early Stage Testing*

This stage of development was focused on building up a robust library of subroutines that would reduce redundancy, as well as make the code easier to extend with additional opcodes.

The first routine that we wrote was *sub_handy_mask*. The subroutine given a starting bit position and size, will mask out the values and return them. We tested this extensively on known hex strings of all sizes and confirmed the results by viewing the return in the registers.

This method of methodical testing through watching both memory and the registers helped us develop the majority of our library. Ensuring that each of these tools worked without error, let us abstract out further and further to build methods that could handle larger and larger tasks.

### *Single Operation Code Testing*

At the bottom of our file we setup a little testing area:

```
ORG     $3000
    EOR.L          D3,D7
```

Since we went about building this as modular as possible, we only tested on EOR initially. We built the output string in memory and monitored the building of the operation label, size, and then the fields. Since EOR supports most EA modes that are available, it was a perfect operation for initial testing.

### *Extending the Opcodes*

After finishing EOR, we started exploring with other operation codes that were more complex or required more features such as MOVE, Bcc, and ADD. The addition of these opcodes required us to extend our library further to support more fields and more operation code structures. At this time we still only would test one operation at a time and focused on getting correct output in the in memory string.

### *Testing Main*

The next phase of the project was building out the IO operations. We needed to accept input for starting and ending addresses, cycle through memory displaying the operation strings, and handle the pause at the end of each screen of data. This final testing stage gave us the ability to start batch testing the operations. By filling our testing area full of these different methods, we were able to quickly verify the correctness of the opcodes that had been defined.

### *Testing all Opcodes*

Defining the remaining opcodes was very rapid due to the robustness of the subroutines that were built. By using the language we developed, we were able to build out the remaining ~25 opcodes in a matter of hours. The majority of errors that we would see in output would be where we made mistakes in the definitions and were easy to fix. Problems with the underlying subroutine library would be reflected in many of the opcodes making debugging very rapid. Each opcode was produced with a wide variety of EA modes and sizes to ensure that the definitions were correct.

### *Moving Memory*

The final phase of testing was moving our large list of testing codes to different places in memory. Initially moving the testing code to memory locations above $FFFF created errors in parsing and output. We were able to fix these and ensure that the program could read in from any point in memory.