**M68k Disassembler**

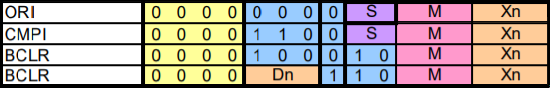
# Project Description

Midori Williams

3/17/2019

### **First Steps**

The first task in this project was to make sense of the opcodes themselves. I condensed the Motorola 68k handbook to just the pages we would be referencing often and cut down the popular color coded opcode pdf, by GoldenCrystal, to the relevant Opcodes and EA modes. There probably wasn’t a single moment we didn’t have that visual aid open on our computers, so it was time well spent. Below is a small expert as an example.



### Program Design and Task Delegation

Our next step was to distinguish patterns of differentiation through flow charting. Our basic approach was as follows: shift the leading 4 bits to the far right, compare it to numbers 0-6, 8, 9, B, C, D, or E and jump to the corresponding section for further narrowing down, else print to screen undecoded data. For each subdivision, we would recopy the opword, identify bits that would reveal the opcode identity with the fewest amount of copy/shift/mask/cmp iterations or repeats. The final step was to branch to the corresponding opcode “method” to decode that particular opword, checking any last bits, if needed, to guarantee identity. This approach aided us later in creating organized code that was also easier to read.

Next we looked for matching sections of the opcodes that could be generalized into further “methods” to cut down on overly repetitive code. It was decided that Derek would focus only on EA modes in the lower 6 bits and provide an entry “api” for the destination EA on MOVE and MOVEA. I would handle I/O, general program flow, buffer management (how ascii strings were added and the order they were added in), and non standard EA modes (ie, not in bits 0-5). I also provided “methods” meant to standardize key operations to avoid inconsistencies and hidden bugs. We decided this was a benefit worth disregarding efficiency over.

The last design step completed mutually was to specify registers, buffers, and constants, especially those that would be used between a JSR and RTS. We created 3 program files (Main, Opcode, EA) and pasted the specifications at the top of each page for quick reference. We found this nearly eliminated issues with integration and avoided any bugs from conflicts in register usage.

### Assembly Coding

What I like about assembly is the granular level of flow control. My mother would tell you I never accepted “because I told you so” as a child, and I don’t think much has changed. Assembly is the grownup answer to my childhood’s “...but WHY?” The parts I enjoyed the most were when I realized I was coding the same thing twice and stopped to think how I could generalize the task for reuse. This led to some opcode sections that were only as many lines as it took to jump through the various “methods”. At a later point I’m interested in learning more about optimization, but stylistically, this beginner’s approach looked nice to me and made it extremely readable (as much as assembly can be). It also saved my hands lots of extra typing!

The first task I generalized was adding a set of variable length strings to a buffer. It is a more trivial task at the end of the 4 week sprint, but took some time to complete and several refinements, closer to the beginning as it was. I also wanted to make Derek’s portion easier, by not handing him a directions to a multi-step process for adding his EA strings. By loading the buffer into A1 at the beginning of each iteration, one needs only to load their string into A0 and jump to ADD2BUFFER (this was clearly inspired by the preconditions for the provided method TrapTask13). A flow chart of this method is pictured below with the others.

The most difficult and time consuming part was decoding MOVEM. I regret not flow charting this particular operation, but it was the last one I worked on and I underestimated the work, based on having already completed all other opcodes. What was unique to this part was having to handle a 32 bit opcode in the confines already set by the rest of the program which expected a 16 bit opcode with with any additional data immediately after dealing with the two directions of the the operation (ea -> Xn or Xn -> ea), and two directions of bit to register mapping depending on the mode. Towards the end, I relied more on running the program to see where it failed than actual intelligent design. Given more time, I would have cleaned this portion up.

The most tedious code (and most prone to typos) was decoding the various Branch on condition codes. I realized AFTER completing them that the project specification only required four.

### Flow Charts

Below are the flow charts we created for this project. It is not at all exhaustive; we found we needed them less and less as our program grew. Code sections with similar routines were easily tweaked and re-used for other operations. As mentioned previously, I only regret not having made one for MOVEM.

