**M68k Disassembler**

# Project Specification

Midori Williams

3/17/2019

### Program Overview

Our disassembler is comprised of 4 files: Main.X68, Opcode.X68, EA.X68, and Config.cfg. Program flow starts in Main. Opcode and EA are included into memory at address $1500 (the memory space physically following the Main section ORGed at $1000). After this, the first command branches to the actual beginning of the disassembler, as specified.

The first 8 bytes (ascii) are read from Config.cfg, converted to a 4 byte hexadecimal starting address (using the provided AsciiToHex method) and loaded into A3. After discarding 2 bytes for a carriage return and line feed, the 2nd 8 bytes are read from the file, converted, and loaded into A5 as the ending address. The program now begins the DISLOOP. The printBuffer is loaded into A1, the address in A3 is converted to an ascii string by jumping to subroutine HexToAsciia which loads the converted string into hex2ascBuffer. This buffer is loaded into A0, then appended to the head of printBuffer by jumping to subroutine ADD2BUFFER. The good flag is set to 1 (good) and the start and end address are compared to ensure start adr < end addr. If false, the program branches to ENDPROG and halts the simulator. If true, program flow continues on to Opcode.X68.

The first word of data is pulled from A3 with post decrement and stored it in D3. D3 is copied into D4, shifted to the right and bitmasked to identify the leading bits 12-15. D4 is compared to 1-6, 8, 9, B, C, D, and E to break it down into one several subgroups and branches if equal to the corresponding OP#. If no match is found, control jumps to BADFLAG where the flag is set to 0 (ie, bad/can’t decode), before returning to Main.

The subgroups are named after their 4 bit hex value.

* OP0: ORI, CMPI, BCLR, or ANDI (extra decode completed)
* OP1: MOVE or MOVEA
* OP4: CLR, NEG, RTS, JSR, MOVEM, LEA, or CHK
* OP5: ADDQ *(accidental extra decode)*, or SUBQ
* OP6: BRA or Bcc *(BCS, BGE, BLT, BVC required; extra decode BHI, BLS, BCC, BNE, BEQ, BVS, BPL, BMI, BGT, and BLE)*
* OP8: DIVS or OR
* OPB: EOR or CMP
* OPD: ADD or ADDA
* OPE: ASL/ASR, LSL/LSR, or ROL/ROR

Other aliases between these groups are for refining the decoding of each section. The bulk of these copy the current opword from D3 to D4, shift it as needed, bit mask to isolate, and compare to identifiable bit values. Once a match is found, the next jump works on decoding a specific opcode. If no match, the opword cannot be decode and jumps to BADFLAG, then returns control to Main.

On a match, the corresponding opcode subroutines identify any remaining bits needed to verify the opword identity, then reads sections of the opcode in the order the human readable text will be displayed. The goal was to all but eliminate rereading any bits of the word or needing to store arbitrary values across subroutine jumps (which would have increased the risk of bugs). That is, once identified, the opcode instruction is loaded into A0, and the program jumps to subroutine ADD2BUFFER which appends the string to the beginning of printBuffer. If there is a size, the size bits are decoded, the associated (.B, .W, .L) string is loaded into A0, and ADD2BUFFER appends the size behind the opcode string in printBuffer. If there is a source operand, the source bits are decoded (if the source is the effective address, program jumps to the EA subroutine). Once the source operand is decoded, any related symbols are appended to printBuffer first, then the associated string through subroutine ADD2BUFFER. Lastly, the destination operand is decoded (in subroutine EA if applicable) and the string appended to printBuffer. Once the full string is appended, flow returns to Main.

Back in Main, the good flag is compared to 1.

* If still set, a #0 is appended to A1 (the printBuffer) directly to null deliminate the string (as required by subroutine TrapTask13). The program jumps to the provided TrapTask13 to append string to the Output.txt and program output window.
* If cleared, jumps to CANTDECODE. This subroutine reloads the head of printBuffer into A1 to overwrite any strings that may have been appended before failure. The undecoded word address is again converted to hex in the hex2ascBuffer, loaded into A0, and added to the head of printBuffer through ADD2BUFFER. The string “DATA” is loaded into the printBuffer. The undecoded word of data is converted to hex in the hex2ascBuffer, then that buffer is appended to printBuffer. Finally a null delimiter of #0 is added to printBuffer and TrapTask13 is called to print it to output window and Output.txt.

In either case, the program loops back to the begining to make the same address comparison again.

### Usage Directions

Main.X68, Opcode.X68, and EA.X68 must be opened in the same program window. Main contains include statements to load the other two files into memory, so it must be the one executed.

Config.cfg must be located in the same file storage location as the others. The format of config is expected to be an 8 number long string, followed by a carriage return / line feed, then another 8 number long string (totally 18 bytes). If the two numbers are formatted in any other way, the program will not read the start and end address correctly.

The program Main initially ORGs at $1500 to load the other two files into memory, then ORGs the remaining (working) code at $1000. This should not be a problem under current project specifications. I’ve also verified that memory space between $9000 onward is free for the graders testing program to be used.

### Code Standards

Our coding standard was as follows:

* 68k opcodes and registers are in all caps
* Leading or “parent” subroutines/methods for general use are in all caps, making them easier to identify
* Method aliases referring to opcodes (all caps) are necessarily preceded by a lowercase “dec” for “decode” and followed by any other differentiating letters when needed.
* General aliases used mid method for navigating comparisons are generally in camelcase, but may be a mix of upper/lower in some other fashion. These lowercase minor aliases are never far from the uppercase method name, and are not used by anything but the “parent” (ie related starting routine)
* Constants/strings/buffers are generally lowercase with either camelcase used or an uppercase “MSG” following the more descriptive part of the variable name, making them easier to identify
* Methods are headed by a descriptive comment and eye catching lines of asterisks (\*\*\*\*)
* Line by line comments are identified with a semicolon ( ; )

Code organization prioritizes first: major entry points and top-to-bottom execution (ie, minimize unnecessary, repetitive jumps up in code). Second: relation of tasks (ie, organization of ideas by conceptual topic), this was for readability and maintenance as the code grew and has nothing to do with optimization. For example, top to bottom, Opcode.X68 starts with the highest level of subdivision, next are the 2ndary subdividers that ultimately discover which possible opcode the word is, last is the actual decoding of each type of opcode. I like to think of this as “locality of ideas”. The third prioritization is “global helper methods” ie, intended for use across files. Examples are ADD2BUFFER, HexToAscii, BADFLAG, etc. The last forth prioritization is for file specific “helper methods” which are not intended for use in other files and were scoped as needed, not by design. Main contains all variables, buffers, and constants/strings used in the entirety of the program at the bottom.