Adam Deehring

Bazen Nega

**Suspicious Disassemblers - Disassembler Project**

**Project Description**

We developed a Disassembler that will breakdown operation codes for the 68k processor. These codes represent the instructions that are understood and can be interpreted by the 68k processor to perform various actions. Our program interpretes the binary instructions that it reads from a file, and converts them back into the assembly language equivalent syntax and grammar that produced those operation codes. In order to accomplish this, instructions need to be placed somewhere in the memory of the 68k, outside the memory space of the disassembler.

Our program is dependent on whether or not a config.cfg file exists, so that it can parse through the file, and if one exists in its directory, and contains a start and end memory address, the disassembler will read hexadecimal words from the memory of the 68k, within that memory region specified by the file. It will begin breaking down words starting at the memory location defined in the first line of the .cfg file (the start address), and run until it reaches the memory location that is defined in the second line of the .cfg file (the end address). Our program only requires that a config.cfg file exist so that it can run, which allows for cases where the memory of the 68k already contains the data that you want to disassemble and where the user wants to import a data file using the easy68k’s “Open Data” functionality, where it places data anywhere in the section of memory desired (outside the scope of the disassembler’s memory).

When our program runs, it will output anything that it prints to the screen (decoded assembly language instructions) into a file called output.txt, using the TrapTask13 method that was provided to us, so that if you blink and miss something on the output window, you can always refer back to that file.

**Approach**

When designing our approach to tackling this assignment, the first thing that we realized that we needed to do was to break down what made a certain sequence of bits unique to the operation code that it represented. We realized that we could break down the commands that we need to support into macro groups, who share similar sections of bits and subdivide those groups into individual instructions, branching off on what pattern of bits makes that operation codes unique, thus separating them from their other siblings commands.

A flowchart was the best way to map out how commands could be broken up, and so with the help of a color coded M68k Opcode manual sSee References), we developed a flowchart (see Appendix A) that listed out how to break down the full opcode in order to reach the supported commands. We would use this flowchart as a resource to refer back to often, referring back whenever we ran into a bug on a command not displaying correctly.

For most commands, the differentiating field was the most significant nibble. This usually provided a unique key for a command in the set of required instructions for the project (although in the greater scope of all the 68k’s instructions this was far from true). When that most significant nibble did not provide enough information, we would broaden our scope and include the next most significant nibble in our decoding process, or whatever other portion of the word had the differentiating information needed. We used bit masks to isolate the bits from that nibble in question, shifting the least significant 1 in the identifying pattern to the least significant position of the 16-bit operation code. With the bits shifted, we would compare the resulting number in decimal (or hex on occasion), to whatever decimal value would be expected for a certain instruction being a correct match.

Whenever we needed to program a part of our disassembler that did not involve breaking down the opcodes (such as reading from a config.cfg file, or converting hex to ASCII), we developed them in an isolated environment within a different x68 file, so that we could minimize the potential bugs. We would then integrate them later, when we were certain they had achieved full functionality. We originally planned on using multiple files and using “include” to glue together these files together, but in the process of working furiously to get all opcodes working properly, we never got around to it. All the functionality is in the “main.x68” file.

There is not necessarily a consistent structure to how we break down the opcodes. We have a main function called “start\_Disassembly” that loops over a specific memory address range, using a constant bit mask declared in the variable section to isolate identifying fields for an instruction. Once it matches said instruction, then it branches to a specialized function that will handle its further processing of effective addresses; if it doesn’t match, via a CMPI statement, then it goes on to the next line where another instruction is checked for until all supported instructions have been checked against. Sometimes you’ll see instructions omitted from start\_Disassembly. This is due to the fact that they are instead checked further down in the disassembler in one of the aforementioned instruction functions, if there are fields in common between the function’s instruction and these other similar instructions. An example of this is if the first four fields match a MOVE instruction, our disassembler will go to the MOVE decoding function, and check in this function if the opcode in fact represents MOVEA, in which case it will branch to its specific label.

**Code Highlights**

One section of our code that we wanted to highlight was our MOVEM subroutine. It ended up being a particularly difficult opcode to decipher, but luckily we got it out of the way early, having implemented it in Week 2 of the Assignment. It is easily the biggest Disassembler subroutine that we implemented, and tends to be very redundant, as we have to account for the MOVEM pre-decrement mode and the post-increment mode, and the reversed order of the two from each other.

Another section of we wanted to highlight was our “HexToAscii”, “Add\_Hex\_String\_To\_Buffer”, ”Start\_String\_Appending”, “Clear\_String\_Buffer” function. They collectively facilitate the creation of output by, converting hex characters in memory to ASCII chars, preparing for the addition of the ASCII interpretations of hex characters to a string buffer within our disassembler’s memory, appending chars to the string buffer, and clearing the buffer whenever a new instruction needs to be built in the buffer or “DATA” need to be printed out, respectively. These complement each other as modular components that allow seamless building of the string that prints out to the console, via TrapTask13.

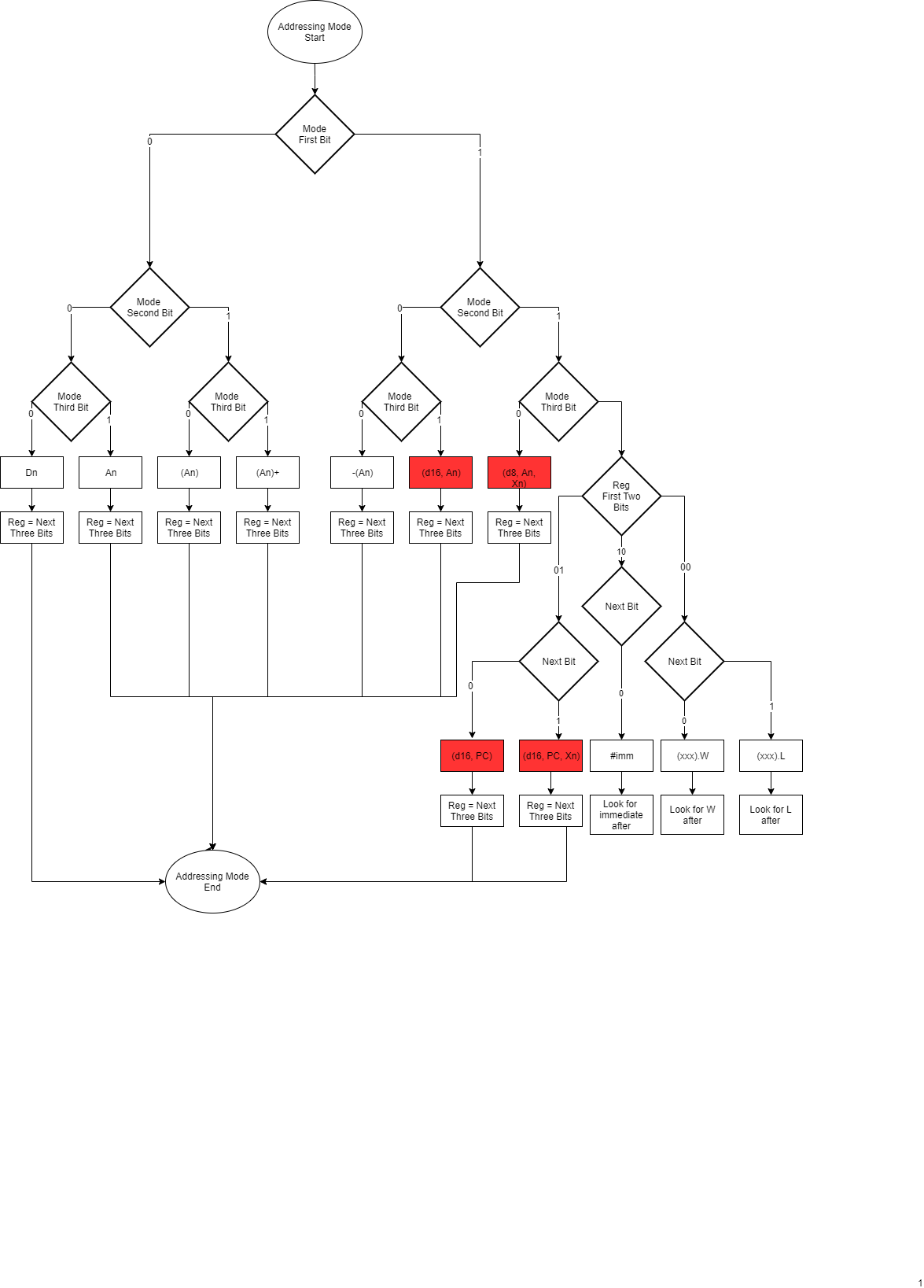
**References**

<https://www.markwrobel.dk/post/amiga-machine-code-part1-deep-dive/>

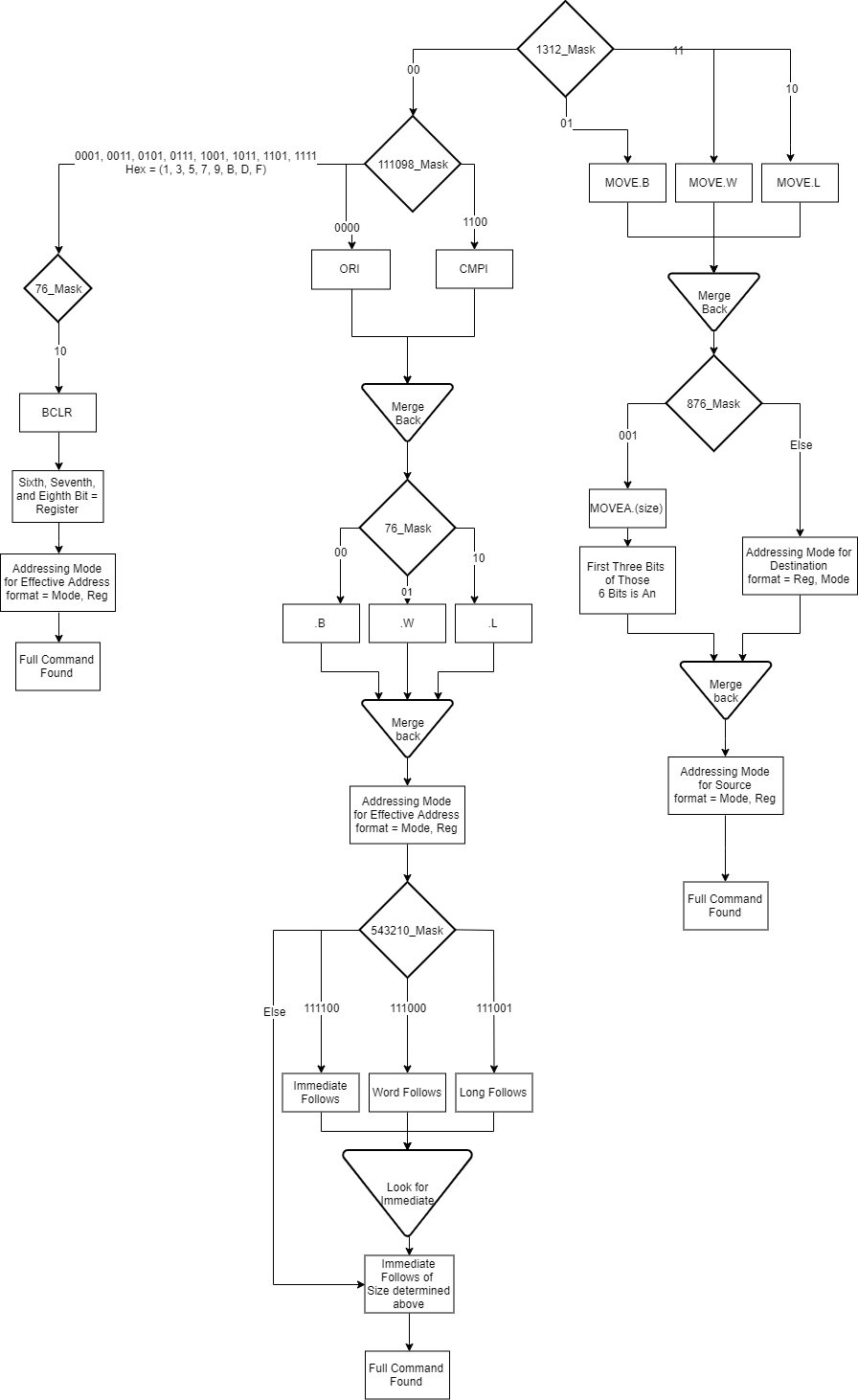
<http://goldencrystal.free.fr/M68kOpcodes.pdf>

**Appendix A**

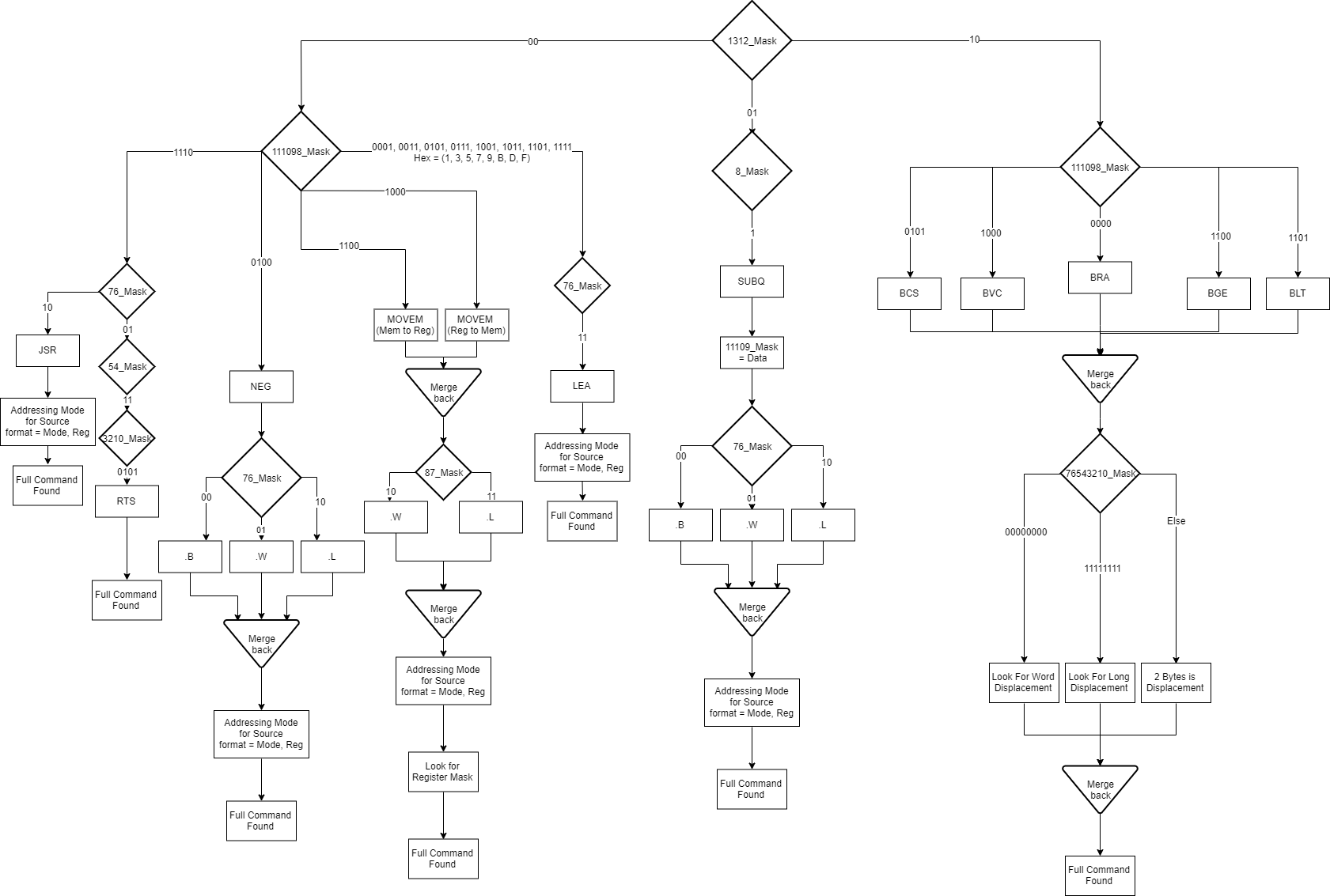
**Addressing Mode Flowchart**



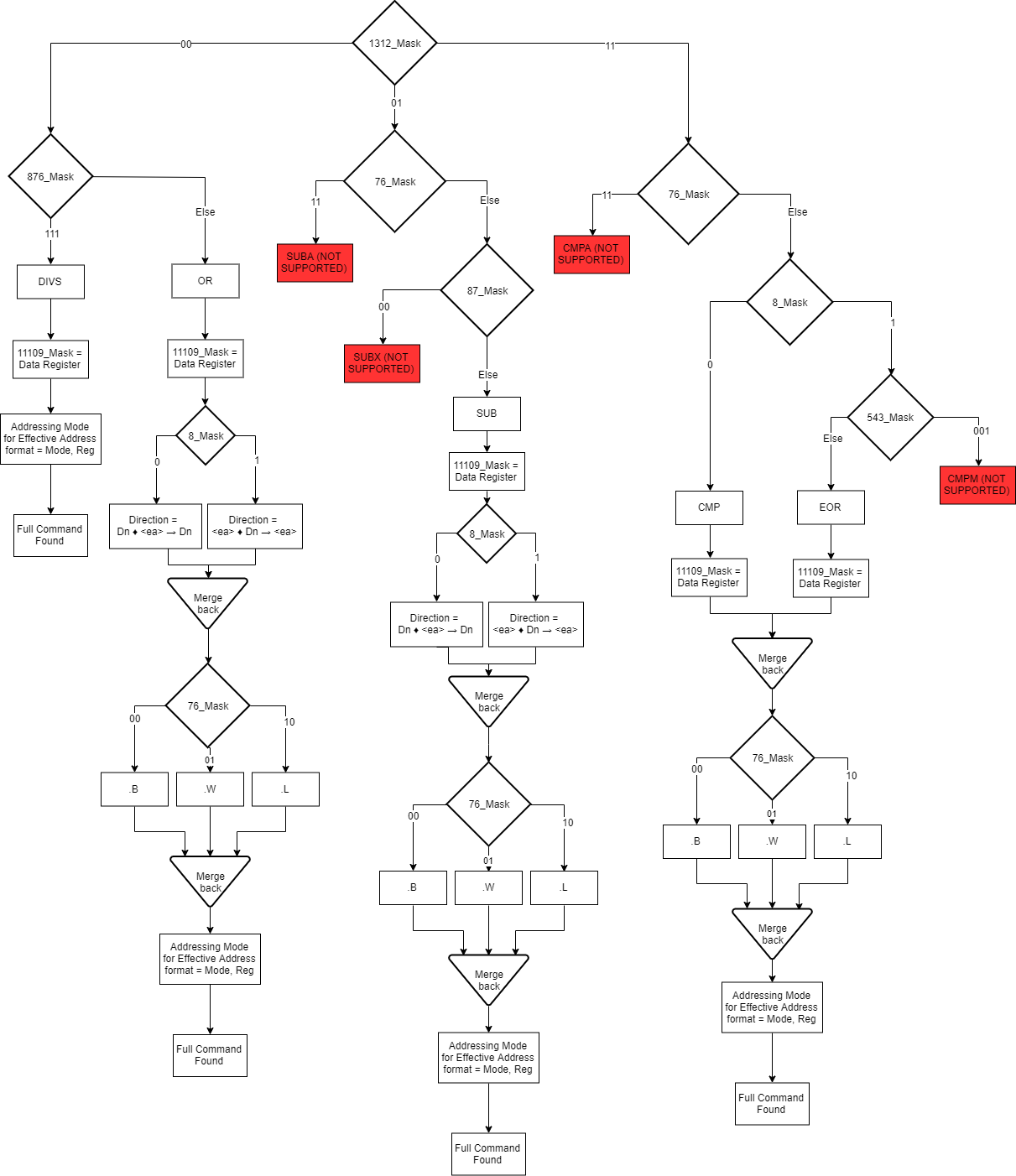
**0x0 - 0x3 Commands Flowchart**



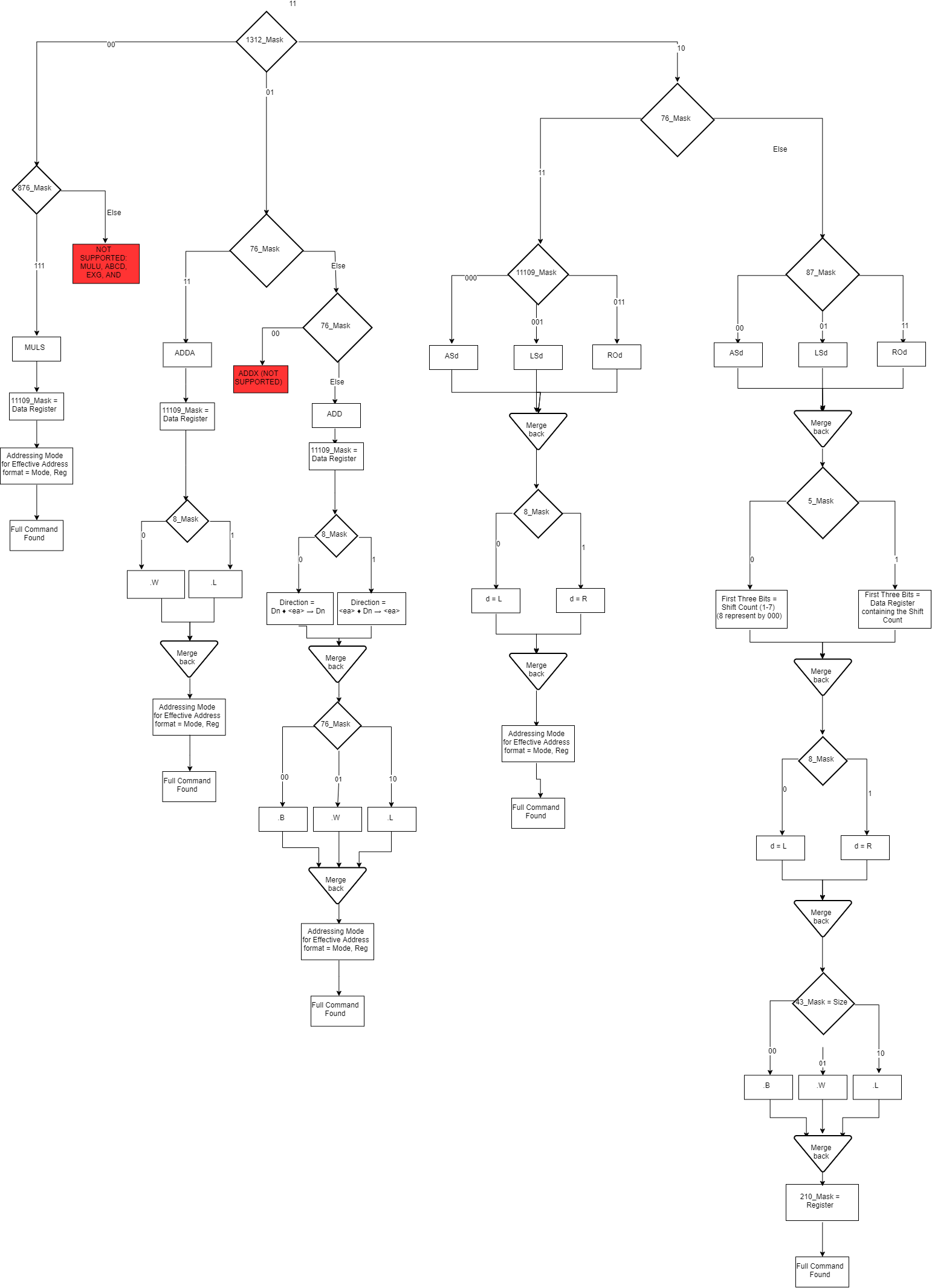
**0x4 - 0x7 Commands Flowchart**



**0x8 - 0xB Commands Flowchart**



**0xC - 0xF Commands Flowchart**



**Masks referred to in Flowchart**

