I did a rough implementation of parsing IO and getting user input of starting and ending address. Check the commit section of github. This is a rough prototype so will be improved in the future. Tomorrow, I will add that with the NOP test program. I will also look into what operations below is easier to implement first

Your task: Look into the logic that I wrote below. Also look into what operations below you want to implement. Ping me on messenger if you need anything. This is in case you wake up before me. When I woke up, we can start discussing about this

P.S: I prefer typing messages of calling

Logic: Load the test file starting from address $7000. Start reading from address 7000 for each op code, the rest ( a byte at a time). For each byte, jump to appropriate place ( look at the table below for example of first byte) . Then do similar for other. Along the way, determine the length of the instruction( to know exactly how many bytes we should go through)

Print out the completed decoded instructions when finished

P.S: Did not take into error checking

Binary format of all code

<http://goldencrystal.free.fr/M68kOpcodes-v2.3.pdf>

Hex format for code

<http://info.sonicretro.org/SCHG:68000_ASM-to-Hex_Code_Reference>

Stretch goal

* MOVEM
* OR (8)
* ORI (0)
* BCLR (0)
* EOR (B)

|  |  |
| --- | --- |
| Starting left most byte | Possible command |
| 0 | CMPI, BCLR, EOR done / |
| 1 | MOVE.B, MOVEA.B done/ |
| 2 | MOVEA.L, MOVE.L done / |
| 3 | MOVE.W. MOVEA.W done/ |
| 4 | NOP, LEA, NEG, JSR, RTS done/ |
| 5 | SUBQ done / |
| 6 | Bcc (BCS, BGE, BLT, BVC), BRA done / |
| 7 |  |
| 8 | DIVS OR done / |
| 9 | SUB done / |
| A |  |
| B | CMP done / |
| C | MULS done/ |
| D | ADD, ADDA done / |
| E | LSR, LSL, ASR, ASL, ROL, ROR done / |
| F |  |

* Need to fix CMP and MULS since there can also be immediate data

Opcode is different with left most byte

Table for op code starting with 0

|  |  |
| --- | --- |
| ORI.B | 0001 |
| ORI.W | 0074 |
| ORI and immediate data | 007C |
| CMPI.B | 0C38 + something |
| CMPI.W | 0C78 + something |
| CMPI.L | 0C82 + something |
| BCLR | Complicated + depending on data register |

TAble for op code starting with 1

|  |  |
| --- | --- |
| MOVE.B | Complicated + depending on data register |
| MOVEA.B | Complicated |

Table for op code starting with 2

|  |  |
| --- | --- |
| MOVEA.L | Complicated + depending on data register |
| MOVE.L | Complicated + depending on data register |

Table for op code starting with 3

Complicated like 1 and 2

Table for op code starting with 4

|  |  |
| --- | --- |
| NOP | 4E71 |
| LEA | 41F8 + something |
| NEG.W | 4478 + something |
| NEG.L  JSR | 4480 + something  4E89 + something |
| RTS | 4E75 |

Table for op code starting with 5

|  |  |
| --- | --- |
| SUBQ | Complicated + depending on data register |

Table for op code starting with 6

|  |  |
| --- | --- |
| BCS | 65 + something |
| BGE | 6C + something |
| BLT | 6D + something |
| BVC | 68 + something |
| BRA | 60 + something |

Table for op code starting with 8

|  |  |
| --- | --- |
| DIVS | Complicated + depending on data register |
| OR | Complicated + depending on data register |

Table for op code starting with B

|  |  |
| --- | --- |
| EOR | Complicated + depending on data register |
| CMP | Complicated + depending on data register |

Table for op code starting with C

MULS is complicated

Table for op code starting with D

Add and ADDA are complicated

Table for opcode starting with E

|  |  |
| --- | --- |
| LSR | E0 |
| LSL,LSR | Depends on immediate value or rotation |
| ASR | Depends on immediate value or rotation |
| ASL | Depends on immediate value or rotation |
| ROL | Depends on immediate value or rotation |
| ROR | Depends on immediate value or rotation |

Feb 25th, Done: NOP and RTS

Looking at op code that start with 0

First Nibble is 0: Fine because the second nibble is C => CMPI

First nibble is 1: Move and movea with byte

First nibble is 2: Move and movea with long

First nibble is 3: Move and movea with word

First nibble is 4 => Notice 4E pattern => NOP,JSR, RTS

First nibble is 5: Notice that for other operations => size is 11 -> eliminate. Then check the last bit => ADDQ or SUBQ

First nibble is 6: Only 3 => check the second nibble => BRA and BSR. Others => Bcc

First nibble is 7: MOVEQ can only have 0 as the last bit in the second nibble => what happen if other invalid data also has it ?

First nibble is 8: Because of the addition of OR, SUBD now can’t be recognize

First nibble is 9: Can’t recognize SUB X if it happens

First nibble is A : No opcode => disqualified immediately

First nibble is B: Only CMP is possible (Does not have size 11 and the last bit of second nibble is 0)

First nibble is C: MULS and MULU only accpet 11 as the size . Differentiate based on the last bit of second nibble

Currenly caveat; Don’t know if an invalid instructions happen, what will be the data. So just print out 4 nibbles at a time => When decoding for op code, might run into cases where the data is decoded as an op code whhile it Is not