Team Little Indian and Big Friends

Duong Chau (EA)

Sai Badey (OP)

Sean Lai (IO)

1. **Program description**

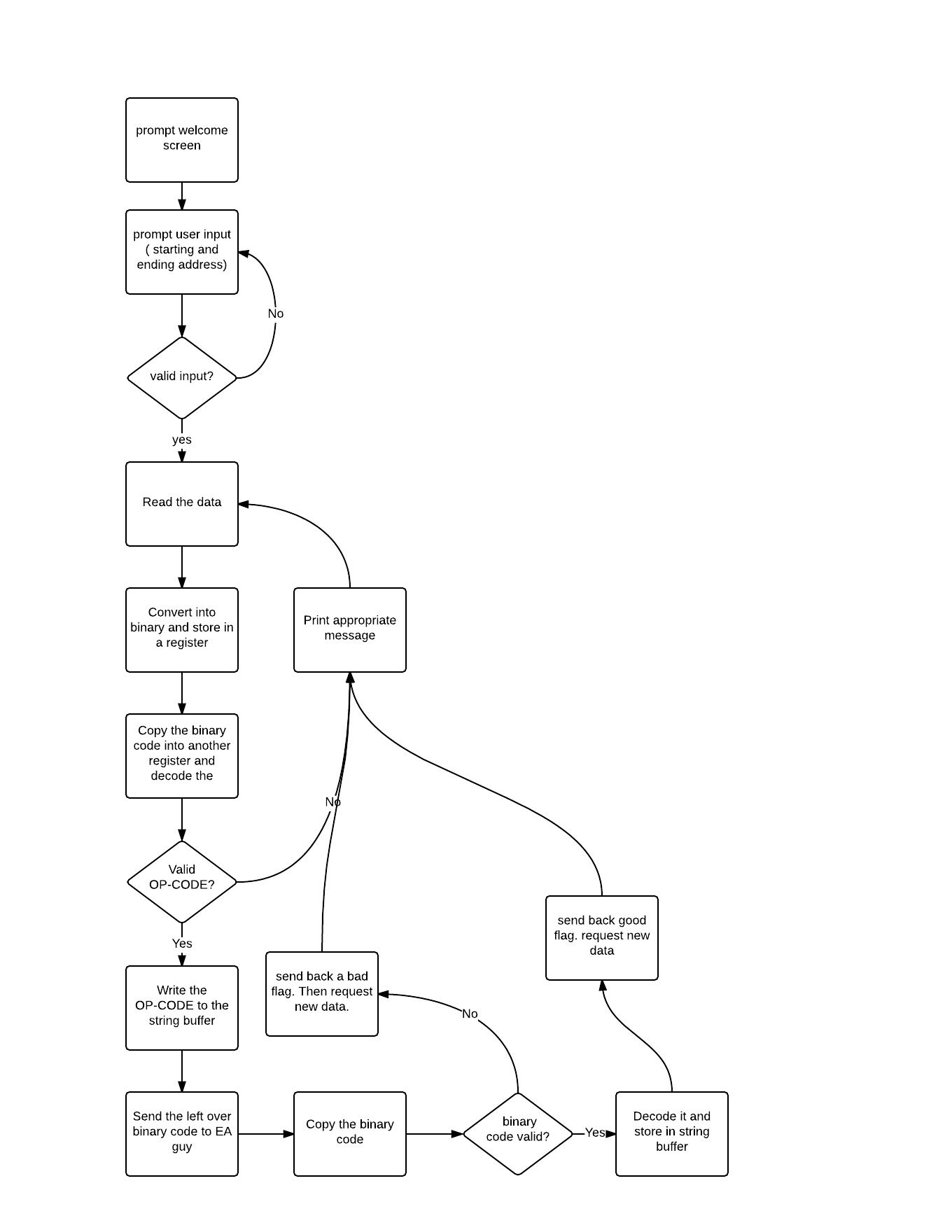
Our project is a disassembler written in 68K language using the Easy68K program. It will read series of memory image of instructions and decode data back to the 68K assembly language and finally prints the decoded output. The project is designed in a way that it’s split into three parts: I/O, Op-codes, and EA.

The general run through of the program will prompt the user to input an address range that they want the disassembler to decode, and the I/O will read each line of instructions to decode using the decode program ran by the OP and EA. If invalid input is detected, the program will prompt the user again until valid inputs are typed in.

The decoded instruction and effective addresses will be printed back to the user to see in the terminal. The decode process will repeat it self until the program has reached the ending address. After reaching certain amounts of lines printed, the program will pause until the user hits the Enter key to display the next page of decoded instructions. After all the instructions are printed, the program will ask if the user wants to restart or quit.

Program Flow

1. The I/O person prompts the user with welcome screen, and inputs that specifies the starting and ending address the disassembler will read from.
2. The disassembler will check the input for errors such as out of bound address, odd Hexadecimal address etc. and length of address.
3. If valid addresses are passed in, the addresses in the memory are now being passed to the OP person.
4. The OP person will attempt to decode the valid instructions and pass the decoded instruction to the I/O person for him to print. OP person will also give the EA of the decoded instruction to the EA person. If the instruction is not valid, invalid message will be printed instead.
5. EA person will use the passed down EA code to decode the operand. If it can’t be decoded, it will be an invalid instruction.
6. Now it will loop back to the step 2 until the disassembler reaches the ending address inputted.
7. The program will ask the user if they want to restart or quit.



2) **Specification**: 1~2 pages.This is a simple list of what your program does.

1. This will be an inverse assembler that can take hex values within addresses and convert them back to instruction codes
2. The program will not be using the TRAP 60 Facility of the simulator. The disassembler will be using our own algorithm, and will not call Trap functions above 14.
3. The program is written from scratch using the Easy68k software, and not cross-pasted from any C++ IDE or any other software.
4. Program should be ORG'ed at $1000.
5. At startup, the program will display the welcome message, and prompt the user to input the start and the ending location of the code to be disassembled. The program will now scan the memory region and output the memory addresses of the instruction.
6. The display will show one screen at the time stopping at 25 lines. It will prompt the user to hit Enter to enter the next page.
7. When the program finds an illegal instruction, it will deal with it until it can find instructions again to decode. When the instruction cannot be decoded, the output should display as 1000 DATA $WXYZ. The program will never crash when it can’t decode an instruction.
8. Program will do a line by line disassembly, displaying the following columns: a- Memory location b- Op-code c- Operand
9. When it completes the disassembly, the program should prompt the user to disassemble another memory image, or prompt the user to quit.

3) **Test Plan**: 1~2 pages.This is a description of how you tested the program. It should also contain a description of your team’s coding standards. If you have, please include your testing files as well.

4) **Exception report**: This is your opportunity to describe problems that you’ve encountered but couldn't fix, or chose not to fix. Anything that you feel deviates from your intended program. Also, this is where you can describe what you were able to complete in the time allotted versus what the assignment asked for. This should definitely include the results of your testing if you found defects but didn't fix them. In an ideal situation, I should be able to just read your documentation and your source listing and give you a grade, without needing to run the program. Of course, I will run the program, but I hope that you get the idea.

5) **Team assignments and report**: A description of how you organized your team’s tasks. That is, "Who did what and how". You should specify the amount of the coding, as a percentage, the member did in the project. This information is VERY important as it will be a source for the separate grading.

We separated our work that based on our roles, but also worked together on a single code after we combined.

Sean Lai – I/O 33% Code

Sai Badey – OP 33% Code

Duong Chau – EA 34% Code

We split our task by having individually writing the code first and then meet up and combine our codes. After we combined our code in the early stage, we mostly worked on one single file with version checks, so we can minimize errors. We are all familiar of our own roles and also the team member’s roles very well, so it wasn't too specific on which part of the code we did individually since we worked together in a single program.

3) **Test Plan**:

We tested the program in 3 separate ways. The first way involved inserting the main opcodes into a test data register as we wrote our individual portions of the program. For example, in the OP code section, there was simply the jump tables and the op code uploader functions which called empty EA functions. At the beginning of this, OP codes that we needed to cover were uploaded into the register and we made sure that our program accurately handled it.

The second manner by which we tested was by uploading the hex codes for each of the test cases into view->memory when we executed the disassembler. We would move the memory to a specific location (such as 7000) and insert multiple different commands and data written in hex. Then, we ran the disassembler to see what it would catch. Note: we did not finish this section because we found a much more efficient way to test – test plan 3.

The third testing method was by having a large list of most of all the test codes that we were required to get working. We created an assembly language program, executed it to obtain the list file, then used the list file to upload the data when we ran the disassembler. When we uploaded the data in this manner, we were able to test much faster than originally. The second method of testing involved creating the binary of almost every case, then converting each to hex values, then uploading them individually to the memory manually. With this method, we saved a lot of time and caught a lot of issues which crashed the program. By working with the tracing built into the emulator, we tracked down the bugs and we caught them.

**Coding Standards:**

The coding standards by which we did our assembler:

* We used capitals for all of our commands and our subroutines.
* When OP code transitions to EA, there is a subroutine for each command
* When there is a memory operation, we used version 1 on the subroutine calls
* When there is a register operation instead of memory, we used version 2
* We used the data registers as following
  + D0: used primarily for trap tasks (printing)
  + D1: reserved for temporary use by IO
  + D2: reserved for temporary use by IO
  + D3: final 16bit binary code
  + D4: temporary holder for bit shifting - used by OP
  + D5: is used to specify printing Word or Long Address/Data
  + D6: temporary holder for bit shifting - used by EA
  + D7: temporary holder for bit shifting - used by EA
  + A1: is used for temporary comparison and
  + A2: loading jump table
  + A3: the good buffer
  + A4: loading jump table
  + A5: starting address
  + A6: ending address
  + A7: the stack - we did not touch this
* We did not print anything out until we verified that it was legitimate code, if at any point we realized that it was invalid, we immediately jumped out of the deeper levels of the loop and printed the basic invalid string buffer.

TEST DATA

MOVE

|  |  |  |
| --- | --- | --- |
| 0001000000000000 | 1000 | MOVE.B   D0,D0 |
| 0001000001001000 | 1048 | MOVE.B   A0, A0  invalid |
| 0001000010010000 | 1090 | MOVE.B   (A0),(A0) |
| 0001000011011000 | 10D8 | MOVE.B   (A0)+,(A0)+ |
| 0001000100100000 | 1120 | MOVE.B   -(A0),-(A0) |
| 0001000101101000 | 1168 | MOVE.B   invalid |
| 0001000110110000 | 11B0 | MOVE.B   invalid |
| 0001000111111000 | 11F8 | MOVE.B   $ASDF, $ASDF |
| 0001001111111001 | 13F9 | MOVE.B   $ASDFASDF, $ASDFASDF |
| 0001100111111100 | 19FC | MOVE.B   invalid |
| 0001000000111100 | 103C | MOVE.B   #ADSF, D0 |
| 0011000000111100 | 303C | MOVE.W  #ASDF, D0 |
| 0010000000111100 | 203C | MOVE.L   #ASDFASDF,D0 |

MOVEA

|  |  |  |
| --- | --- | --- |
| 0001 000 001 000 000 | 1040 | MOVE.B   D0, A0 |
| 0001 000 001 001 000 | 1048 | MOVE.B   A0, A0 |
| 0001 000 001 010 000 | 1050 | MOVE.B   (A0), A0 |
| 0001 000 001 011 000 | 1058 | MOVE.B   (A0)+, A0 |
| 0001 000 001 100 000 | 1060 | MOVE.B   -(A0), A0 |
| 0001 000 001 101 000 | 1068 | MOVE.B   invalid |
| 0001 000 001 110 000 | 1070 | MOVE.B   invalid |
| 0001 000 001 111 000 | 1078 | MOVE.B   $ASDF, A0 |
| 0001 001 001 111 001 | 1279 | MOVE.B   $ASDFASDF, A0 |
| 0001 100 001 111 100 | 187C | MOVE.B   invalid |
| 0001 000 001 111 100 | 107C | MOVE.B   #ADSF, A0 |
| 0011 000 001 111 100 | 307C | MOVE.W  #ASDF, A0 |
| 0010 000 001 111 100 | 207C | MOVE.L   #ASDFASDF,A0 |

MOVEM\_0W

|  |  |  |
| --- | --- | --- |
| 0100100010 000 000 | 4880 | Invalid |
| 0100100010 001 000 | 4888 | Invalid |
| 0100100010 010 000 | 4890 | MOVEM.W  #ASDF,(A0) |
| 0100100010 011 000 | 4898 | Invalid |
| 0100100010 100 000 | 48A0 | MOVEM.W  #ASDF, -(A0) |
| 0100100010 101 000 | 48A8 | Invalid |
| 0100100010 110 000 | 48B0 | Invalid |
| 0100100010 111 000 | 48B8 | MOVEM.W  #ASDF, $ASDF |
| 0100100010 111 001 | 48B9 | MOVEM.W  #ASDF, $ASDFASDF |
| 0100100010 111 100 | 48BC | Invalid |

MOVEM\_0L

|  |  |  |
| --- | --- | --- |
| 0100100011 000 000 | 48C0 | Invalid |
| 0100100011 001 000 | 48C8 | Invalid |
| 0100100011 010 000 | 48D0 | MOVEM.W  #ASDFASDF,(A0) |
| 0100100011 011 000 | 48D8 | Invalid |
| 0100100011 100 000 | 48E0 | MOVEM.W  #ASDFASDF, -(A0) |
| 0100100011 101 000 | 48E8 | Invalid |
| 0100100011 110 000 | 48F0 | Invalid |
| 0100100011 111 000 | 48F8 | MOVEM.W  #ASDFASDF, $ASDF |
| 0100100011 111 001 | 48F9 | MOVEM.W  #ASDFASDF, $ASDFASDF |
| 0100100011 111 100 | 48FC | Invalid |

MOVEM\_1W

|  |  |  |
| --- | --- | --- |
| 0100110010 000 000 | 4C80 | Invalid |
| 0100110010 001 000 | 4C88 | Invalid |
| 0100110010 010 000 | 4C90 | MOVEM.W  #ASDF,(A0) |
| 0100110010 011 000 | 4C98 | MOVEM.W  #ASDF, (A0)+ |
| 0100110010 100 000 | 4CA0 | Invalid |
| 0100110010 101 000 | 4CA8 | Invalid |
| 0100110010 110 000 | 4CB0 | Invalid |
| 0100110010 111 000 | 4CB8 | MOVEM.W  #ASDF, $ASDF |
| 0100110010 111 001 | 4CB9 | MOVEM.W  #ASDF, $ASDFASDF |
| 0100110010 111 100 | 4CBC | Invalid |

MOVEM\_1L

|  |  |  |
| --- | --- | --- |
| 0100110011 000 000 | 4CC0 | Invalid |
| 0100110011 001 000 | 4CC8 | Invalid |
| 0100110011 010 000 | 4CD0 | MOVEM.W  #ASDFASDF,(A0) |
| 0100110011 011 000 | 4CD8 | MOVEM.W  #ASDFASDF, (A0)+ |
| 0100110011 100 000 | 4CE0 | Invalid |
| 0100110011 101 000 | 4CE8 | Invalid |
| 0100110011 110 000 | 4CF0 | Invalid |
| 0100110011 111 000 | 4CF8 | MOVEM.W  #ASDFASDF, $ASDF |
| 0100110011 111 001 | 4CF9 | MOVEM.W  #ASDFASDF, $ASDFASDF |
| 0100110011 111 100 | 4CFC | Invalid |

ADD\_v1

|  |  |  |
| --- | --- | --- |
| 1101 000 001 000 000 | D040 | ADD.W D0,D0 |
| 1101 000 001 001 000 | D048 | ADD.W A0,D0 |
| 1101 000 001 010 000 | D050 | ADD.W (A0),D0 |
| 1101 000 001 011 000 | D058 | ADD.W (A0)+,D0 |
| 1101 000 001 100 000 | D060 | ADD.W –(A0),D0 |
| 1101 000 001 101 000 | D068 | Invalid |
| 1101 000 001 110 000 | D070 | Invalid |
| 1101 000 001 111 000 | D078 | ADD.W $ASDF,D0 |
| 1101 000 001 111 001 | D079 | ADD.W $ASDFASDF,D0 |
| 1101 000 001 111 100 | D07C | ADD.W #ASDF,D0 |

ADD\_v2

|  |  |  |
| --- | --- | --- |
| 1101 000 101 000 000 | D140 | Invalid |
| 1101 000 101 001 000 | D148 | Invalid |
| 1101 000 101 010 000 | D150 | ADD.W  D0, (A0) |
| 1101 000 101 011 000 | D158 | ADD.W  D0, (A0)+ |
| 1101 000 101 100 000 | D160 | ADD.W  D0, -(A0) |
| 1101 000 101 101 000 | D168 | Invalid |
| 1101 000 101 110 000 | D170 | Invalid |
| 1101 000 101 111 000 | D178 | ADD.W  D0, $ASDF |
| 1101 000 101 111 001 | D179 | ADD.W  D0, $ASDFASDF |
| 1101 000 101 111 100 | D17C | Invalid |

ADDA

|  |  |  |
| --- | --- | --- |
| 1101 000 011 000 000 | D0C0 | ADDA.W  D0,A0 |
| 1101 000 011 001 000 | D0C8 | ADDA.W  A0.A0 |
| 1101 000 011 010 000 | D0D0 | ADDA.W  (A0),A0 |
| 1101 000 011 011 000 | D0D8 | ADDA.W  (A0)+,A0 |
| 1101 000 011 100 000 | D0E0 | ADDA.W  -(A0),A0 |
| 1101 000 011 101 000 | D0E8 | INVALID |
| 1101 000 011 110 000 | D0F0 | INVALID |
| 1101 000 011 111 000 | D0F8 | ADDA.W  $ASDF,A0 |
| 1101 000 011 111 001 | D0F9 | ADDA.W  $ASDFASDF,A0 |
| 1101 000 011 111 100 | D0FC | ADDA.W  #ASDF,A0 |
| 1101 000 111 111 100 | D1FC | ADDA.L   #ASDFASDF,A0 |

ADDQ

|  |  |  |
| --- | --- | --- |
| 0101 000 0 00 000 000 | 5000 | ADDQ.B  #0,D0 |
| 0101 000 0 00 001 000 | 5008 | ADDQ.B  #0,A0 |
| 0101 000 0 00 010 000 | 5010 | ADDQ.B  #0,(A0) |
| 0101 000 0 00 011 000 | 5018 | ADDQ.B  #0,(A0)+ |
| 0101 000 0 00 100 000 | 5020 | ADDQ.B  #0,-(A0) |
| 0101 000 0 00 101 000 | 5028 | INVALID |
| 0101 000 0 00 110 000 | 5030 | INVALID |
| 0101 000 0 00 111 000 | 5038 | ADDQ.B  #0,$ASDF |
| 0101 000 0 00 111 001 | 5039 | ADDQ.B  #0,$ASDFASDF |
| 0101 000 0 00 111 100 | 503C | INVALID |

SUBBI

|  |  |  |
| --- | --- | --- |
| 00000100 00 000 000 | 0400 | SUBI.B #ASDF,D0 |
| 00000100 10 000 000 | 0480 | SUBI.L #ASDFASDF,D0 |
| 00000100 00 001 000 | 0408 | Invalid |
| 00000100 00 010 000 | 0410 | SUBI.B #ASDF,(A0) |
| 00000100 00 011 000 | 0418 | SUBI.B #ASDF,(A0)+ |
| 00000100 00 100 000 | 0420 | SUBI.B #ASDF,-(A0) |
| 00000100 00 101 000 | 0428 | INVALID |
| 00000100 00 110 000 | 0430 | INVALID |
| 00000100 00 111 000 | 0438 | SUBI.B #ASDF,$ASDF |
| 00000100 00 111 001 | 0439 | SUBI.B #ASDF,$ASDFASDF |
| 00000100 00 111 100 | 043C | INVALID |

ASd\_v1   memory shift

|  |  |  |
| --- | --- | --- |
| 1110000011 000 000 | E0C0 | Invalid |
| 1110000011 001 000 | E0C8 | Invalid |
| 1110000011 010 000 | E0D0 | ASL  (A0) |
| 1110000011 011 000 | E0D8 | ASL  (A0)+ |
| 1110000011 100 000 | E0E0 | ASL  -(A0) |
| 1110000011 101 000 | E0E8 | Invalid |
| 1110000011 110 000 | E0F0 | Invalid |
| 1110000011 111 000 | E0F8 | ASL  $ASDF |
| 1110000011 111 001 | E0F9 | ASL  $ASDFASDF |
| 1110000011 111 100 | E0FC | Invalid |

ASd\_v2   register shift

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

LEA

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

ORI

|  |  |  |
| --- | --- | --- |
| 0000 0000 00 000 000 |  | ORI.B #4,D0 |
| 0000 0000 00 001 000 |  |  |
| 0000 0000 00 010 000 |  |  |
| 0000 0000 00 011 000 |  |  |
| 0000 0000 00 100 000 |  |  |
| 0000 0000 00 111 000 |  |  |
| 0000 0000 00 111 001 |  |  |
| 0000 0000 01 000 000 |  | ORI.W #4,D0 |
| 0000 0000 10 000 000 |  | ORI.L #4,D0 |
|  |  |  |

NEG

|  |  |  |
| --- | --- | --- |
| 0100 0100 00 000 000 |  | NEG D0,D0 |
| 0100 0100 00 001 000 |  | INVALID |
| 0100 0100 00 010 000 |  | MULS (A0),D0 |
| 0100 0100 00 011 000 |  | MULS (A0)+,D0 |
| 0100 0100 00 100 000 |  | MULS -(A0),D0 |
| 0100 0100 00 101 000 |  | INVALID |
| 0100 0100 00 110 000 |  | INVALID |
| 0100 0100 00 111 000 |  | grab 4 |
| 0100 0100 00 000 001 |  | grab 8 |

AND\_v1

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

AND\_v2

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

EOR

|  |  |  |
| --- | --- | --- |
| 1011 000 100 000 000 |  | EOR.B D0,D0 |
| 1011 000 101 011 000 |  | EOR.W D0, (An)+ |
| 1011 000 110 010 000 |  | EOR.L D0, (An) |
| 1011 000 101 001 000 |  | INVALID |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

EORI

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

NOT

|  |  |  |
| --- | --- | --- |
| 0100 0110 00 000 000 |  | NOT.B D0 |
| 0100 0110 00 001 000 |  | INVALID |
| 0100 0110 00 010 000 |  | NOT.B (A0) |
| 0100 0110 00 011 000 |  | NOT.B (A0)+ |
| 0100 0110 00 100 000 |  | NOT.B -(A0) |
| 0100 0110 00 111 000 |  | grab 4 |
| 0100 0110 00 111 000 |  | grab 8 |
| 0100 0110 01 000 000 |  | NOT.W D0 |
| 0100 0110 10 000 000 |  | NOT.L D0 |
| 0100 0110 11 000 000 |  | INVALID |

BTST\_v1   Dn to <EA>

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

BTST\_v2   #<DATA> to <EA>

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

MULS

|  |  |  |
| --- | --- | --- |
| 1100 000 111 000 000 | C1C0 | MULS D0,D0 |
| 1100 000 111 001 000 | C1C8 | INVALID |
| 1100 000 111 010 000 | C1D0 | MULS (A0),D0 |
| 1100 000 111 011 000 | C1D8 | MULS (A0)+,D0 |
| 1100 000 111 100 000 | C1E0 | MULS -(A0),D0 |
| 1100 000 111 101 000 | C1E8 | INVALID |
| 1100 000 111 110 000 | C1F0 | INVALID |
| 1100 000 111 111 000 | C1F8 | should grab 4 bytes of data |
| 1100 000 111 111 001 | C1F9 | should grab 8 bytes of data |
| 1100 000 111 111 100 | CIFC | should grab 4 bytes of data |
| 1100 000 111 111 010 | C1FA | INVALID |
| 1100 000 111 111 011 | CIFB | INVALID |
| 1100 000 111 111 110 | CIFE | INVALID |
| 1100 000 111 111 111 | CIFF | INVALID |
| 1100 000 111 111 101 | CIFD | INVALID |

DIVS

|  |  |  |
| --- | --- | --- |
| 1000 000 111 000 000 |  | DIVS D0,D0 |
| 1000 000 111 001 000 |  | INVALID |
| 1000 000 111 010 000 |  | DIVS (A0),D0 |
| 1000 000 111 011 000 |  | DIVS (A0)+,D0 |
| 1000 000 111 100 000 |  | DIVS -(A0),D0 |
| 1000 000 111 101 000 |  | INVALID |
| 1000 000 111 110 000 |  | INVALID |
| 1000 000 111 111 000 |  | grab 4 |
| 1000 000 111 111 001 |  | grab 8 |
| 1000 000 111 111 100 |  | grab immediate data 4? |
| 1000 000 111 111 010 |  | INVALID |
| 1000 000 111 111 011 |  | INVALID |

4) **Exception report**:

We ran into an issue with the following bugs and we do not know why it is the way it is:

1. SUBI.B #$02,(A6) has the hex value of 5516 which does not match the syntax of SUBI. Therefore this will result in a error and print out the default Error Message instead of having the command decoded
2. ADD.L #$1234,D3 this will turn the code to ADDI.L by default and will mess up the command list. A couple of the following command in the list will print out the default Error Message.