# 68k Disassembler

By Youssef Beltagy

This is a disassembler for Motorola's MC68000 Microprocessor. This microprocessor was used in the first Macintosh, so it is historically significant.

A disassembler is a program that reads memory and converts it back into assembly code. This program is written in 68k assembly language and made to disassemble the 68k language.

Though I would have been happy to make this a full-featured disassembler, the project requires me to only make the required op-codes and effective addressing modes. I will be penalized for implementing extra features.

This project, thus, only handles the required op-codes and effective addressing modes. Any unessential op-code or effective-addressing mode is treated as data.

This wiki explains the specifications, design, and testing of the disassembler.

To the best of my abilities, this program doesn't have bugs. So I don't know what to put in the exceptions part of the report. I put the challengers I faced in the challenges section.

I implemented this project alone. I did 100% of the coding in this project. There is an RTS subroutine that Jordan implemented, but I don't know if it counts. Since it is a copy of my NOP. I really don't know what you want me to put in the evaluations section of this report since I'm a one-man-team.

## Specifications

For this project, I was required to make a 68k disassembler that can disassemble 15 op-codes but only with a limited set of addressing modes.

I will be penalized for implementing any unessential op-code or effective addressing mode. So I only implemented what was required. Any unessential op-code is printed as data. Any op-code with an unessential effective addressing mode is printed as data even if the op-code itself is required.

The program must ask the user for two addresses; a starting address and an ending address. The addresses must be validated before the program starts disassembling.

### Required op-codes

1. [x] NOP
2. [x] MOVE
3. [x] MOVEM
4. [x] ADD
5. [x] SUB
6. [x] MULS
7. [ ] DIVU -- No longer required (Must not implement or I will lose points)
8. [x] LEA
9. [x] AND
10. [x] NOT
11. [x] LSL
12. [ ] LSR -- No longer required (Must not implement or I will lose points)

13. [ ] ASL -- No longer required (Must not implement or I will lose points)
14. [x] ASR
15. [x] Bcc     (BLT, BGE, BEQ)
16. [x] JSR
17. [x] RTS
18. [x] BRA

### Required Effective Addressing Modes

I had to handle the following effective addressing modes.

1. Data Register Direct
2. Address Register Direct
3. Address Register Indirect
4. Immediate Data
5. Address Register Indirect with Post incrementing
6. Address Register Indirect with Pre decrementing
7. Absolute Long Address
8. Absolute Word Address

## Design

To explain the design, I need to define an important term, signature.

### Signature

The design is built on the concept of op-code signatures. I define a signature as the common component in the 16-bit permutations that an op-code maps onto when it is assembled. While one op-code can map onto many permutations, each permutation can only map to one op-code.

The signature must be loose enough to include all permutations of an op-code, but must be tight enough that it doesn't contain another op-code's permutations.

The set of all permutations that contain the signature is equal to the set of permutations of an op-code.

These signatures are exclusive to one op-code. Two op-codes cannot share the same signature. If a signature is shared across op-codes, then it is wrong by definition.

### Program logic

Now that you know what I mean by signatures, I can explain the design. The design is a loop that checks if the data in the memory matches an op-code's signature.

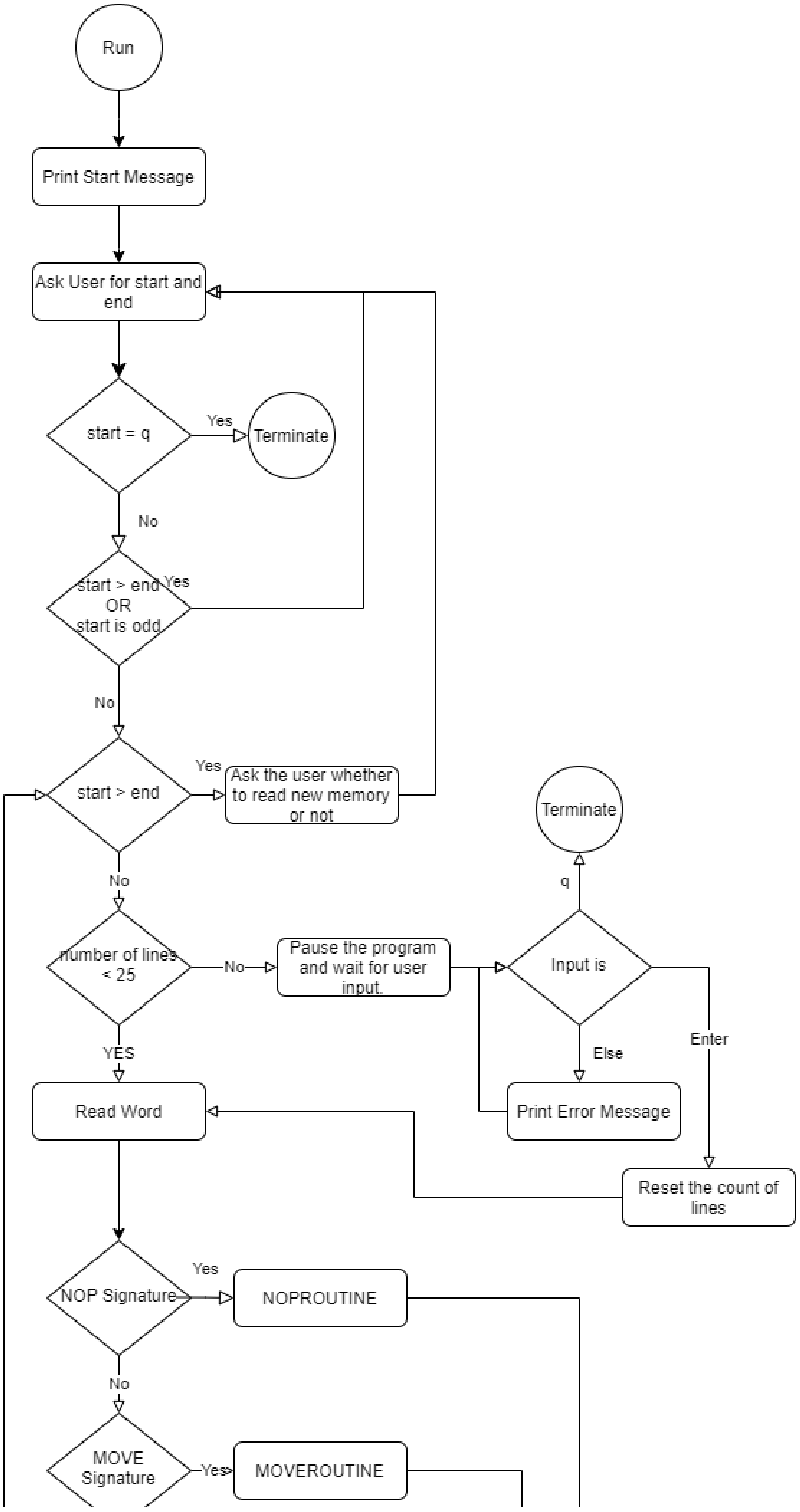
The program will ask the user for starting and ending memory addresses to disassemble. Then the program will loop through the memory, word-by-word, and disassemble each word.

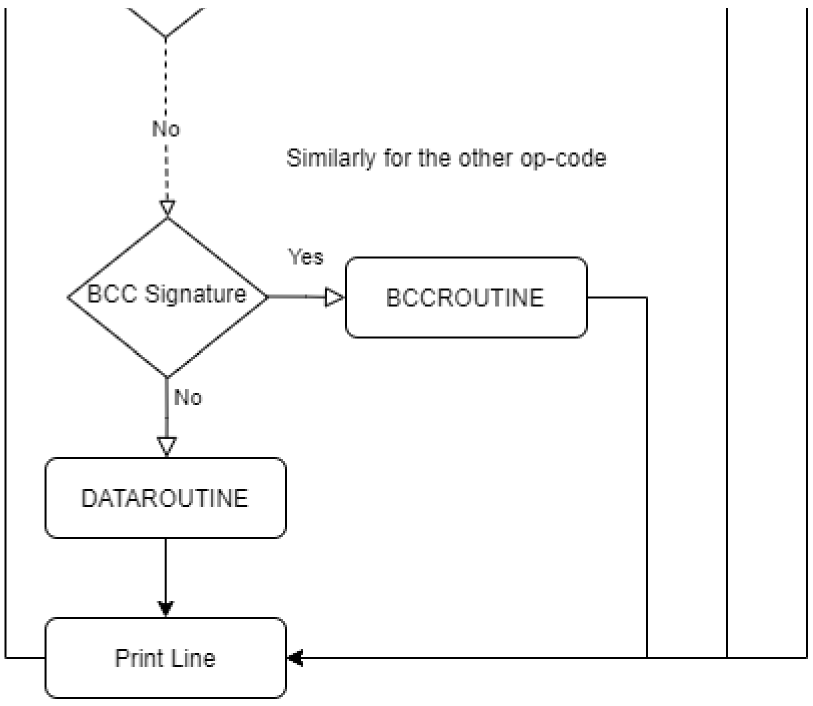
Then the program will check if the read word matches a specific op-code signature. The program checks the op-codes one by one.

If the word matches an op-code, the subroutine for that specific op-code is called. If the word didn't match a subroutine, the word will be treated as data, and the program will call the data subroutine.

The program will print the op-code or data until the end of the output console or the end memory address. If the program reached the end of the output console, the program will wait for the user to press "enter" before it disassembles more. If the program reached the end of the memory the user entered, the program will restart and ask the user for a new starting and ending addresses.

## Flow-Chart


Run
Print Start Message
Ask User for start and end
start = q
Yes
Terminate
No
start > end OR start is odd
Yes
No
start > end
Yes
Ask the user whether to read new memory or not
Terminate
q
No
number of lines < 25
No
Pause the program and wait for user input.
Input is
Else
Enter
YES
Read Word
Print Error Message
Reset the count of lines
NOP Signature
Yes
NOPROUTINE
No
MOVE Signature
Yes
MOVEROUTINE



## Guidelines

This is a long project. So I defined some guidelines to follow when writing the code to ensure high-quality and easy integration.

### Test-Driven Development

I test my code and document these tests. I follow the guidelines of Test-Driven Development.

1. Write a unit test.
2. Run the test and ensure it fails before you implement.
3. write the minimum code for the test to pass.
4. Run the test and ensure it succeeds.
5. Refactor the code until it is simple.
6. Repeat and accumulate unit tests

I Test that the code works when it should and fails when it shouldn't. I don't want to print an op-code when I am given wrong input.

### Style Guidelines

I write my code in small letters.

I write my labels in capital letters.

I follow a standard naming process for my opcode-subroutines. I prepend the name of the op-code to its subroutine. For example, I name the subroutine that handles move, MOVEROUTINE.

For labels inside subroutines, I prepend the subroutine label. For example, if I'm writing a loop inside MOVEROUTINE, I name the label for the loop MOVEROUTINE_LOOP. This allows me to avoid accidentally duplicating a label and moving the program control haphazardly.

### Git Rules

For every feature, I make a branch. I name the branch with that feature.

When I am finished implementing a feature, I make a merge request and review it one last time.

Again, I follow the principles of test-driven development. I make tests. I ensure they fail. I make the tests pass, and I record the tests in merge requests.

## Implementation

In here, I discuss important implementation and execution details.

The program requires two inputs; a starting address and an ending address. The inputs must be typed as 8 hex-digit characters. The program is case-insensitive so it doesn't matter if you use capital letters or small ones. The addresses are validated by ensuring that all the digits are hex-digits, the most significant byte of both addresses is 0, the starting address is even, and the ending address is bigger than or equal to the starting address. Notice that the ending address doesn't have to be even.

While the starting address is less than or equal to the ending address, the program disassembles op-codes. The program prints 25 lines at a time and then pauses. When the program pauses, you can type enter to continue disassembling or type "q" to exit. Notice that "q" must be lower case.

When the program finishes execution, it asks for the starting and ending addresses again. If you want to exit, again type "q". You can exit when the program asks for the starting input, but you can't exit when the program asks for the ending input.

Unessential op-codes or op-codes with unessential effective addressing modes are printed as data.

I'm proud to say that my program is well commented and documented. It is modular and reusable. In fact, most of the core functions of the program is implemented in reusable subroutines that I used again and again.

I distinguish between two types of subroutines. Utility subroutines are routines that handle general program logic and are to be used in multiple op-codes. Op-code Subroutines are subroutines that handle a single op-code from the beginning to the end.

Every subroutine starts with comments that explains what it does, its inputs, and its outputs.

### Utility Subroutines

```
INPUT_OR_EXIT:
*Description:
*Ensures that INPUT_START and INPUT_END
*contain valid starting and ending addresses
*or returns -1 if the program should terminate.
*The addresses are valid if INPUT_END >= INPUT_START
*and INPUT_START is even.
*sets d7 to 0 for valid input and to -1 for exit.
*Nothing changes other than INPUT_START, INPUT_END, and d7
*Input: nothing
*Output: d7.l, INPUT_START, INPUT_END



CONTINUE_OR_EXIT:
```

```
*Description:
*Ensures that MAIN_LOOP_COUNTER is reset or that
*d7.l = -1 to indicate that the program should terminate.
*nothing changes other than MAIN_LOOP_COUNTER or d7.l
*Input: nothing
*Output: d7.l



LONG_FROM_STRING:
*Description:
*Given a string at a1 and its size at d1.w, returns a hex number at
*d6 and 0 at d7.l, or -1 at d7.l to represent an error.
*nothing other than d7 and d6 will change
*Input: a1, d1.w
*Output: d7.l, d6.l


STRING_FROM_NIBBLE:
*Description:
*Given the lower nibble at d2.b, will convert that into a char in the
*memory pointed to by a1.
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to


STRING_FROM_BYTE:
*Description:
*Given a byte at d2.b, will convert that into a string of hex
*digits pointed to by a1
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to


STRING_FROM_WORD:
*Description:
*Given a word at d2.w, will convert that into a string of hex
*digits pointed to by a1
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.w
*Output: a1 and the memory it points to


STRING_FROM_LONG:
*Description:
*Given a long at d2.l, will convert that into a string of hex
*digits pointed to by a1
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.l
*Output: a1 and the memory it points to


COPY_STRING_A2_TO_A1:
*Description:
*Given a null terminated string at a2, will
*copy the string to a1 except the terminating null.
```

```
*Nothing other than a1 will change
*Input: a1, a2
*Output: a1


IS_EA_VALID:
*Description:
*Determines if the EA of the current op-code at (a6) is valid or not.
*If the effective address is not valid, then calls DATAROUTINE
*and returns -1 in d7.l.
*If the effective address is valid, returns 0 in d7.l
*Nothing should be affected other than a6, d7.l, a1,
*and the memory a1 points to.
*Input: a6, a1
*Output: a6, d7.l, a1, and the memory pointed to by a1.


GET_LIGHT_PURPLE_SIZE:
*Description:
*Given an op-code at (a6) that uses
* the light purple size in http://goldencrystal.free.fr/M68kOpcodes-v2.3.pdf.
* This subroutine will print the approperiate size (B|W|L) in the value pointed
to by a1.
* If the input is Invalid, prints ERROR_STRING.
* returns the size in TEMP_VARIABLE.b.
*Input: a6
*Output: a1, TEMP_VARIABLE.b


GET_A_REG_DIRECT:
*Description:
*Given a byte at d2.b, will print "A" then the number of the
*address register that is specified in d2.b.
*Assumes the address register number (d2.b) is valid [0,7].
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to


GET_D_REG_DIRECT:
*Description:
*Given a byte at d2.b, will print "D" then the number of the
*address register that is specified in d2.b.
*Assumes the data register number (d2.b) is valid [0,7].
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to


GET_A_REG_INDIRECT:
*Description:
*Given a byte at d2.b, will print "(A" then the number of the
*address register that is specified in d2.b and ")".
*Assumes the address register number (d2.b) is valid [0,7].
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to
```

```
GET_A_REG_INDIRECT_POST:
*Description:
*Given a byte at d2.b, will print "(A" then the number of the
*address register that is specified in d2.b and ")+".
*Assumes the address register number (d2.b) is valid [0,7].
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to


GET_A_REG_INDIRECT_PRE:
*Description:
*Given a byte at d2.b, will print "-(A" then the number of the
*address register that is specified in d2.b and ")".
*Assumes the address register number (d2.b) is valid [0,7].
*nothing other than a1 and the memory it points to will change
*Input: a1, d2.b
*Output: a1 and the memory it points to


GET_INVALID_ADDRESSING_MODE:
*Description:
*Loads "IAM" into the memory pointed to by a1
*nothing other than a1 and the memory it points to will change
*Input: a1
*Output: a1 and the memory it points to


GET_EA: * Get the effective address.
*Description:
*Requires d3.b to contain the size of the instruction in case it is immediate.
*You must set d3.b to a value in the range of [0,2]. 0 for byte. 1 for word. 2
for long.
*If d3.b contains anything aside from [0,2] and the EA was immediate,
ERROR_STRING will be printed.
*Requires d2.w to contain the instruction.
*Requires a1 to point to buffer.
*Requires a6 to point to the next insturction or the memory of
*the data of this EA.
*Again, Requires a6 to point to the memory of the data of this EA or the
*next instruction.
*By the end of this subroutine, a6 will point to the next
*instruction.
*This subroutine is really powerful. But it needs to make a lot
*of assumptions. It is YOUR responsibility to ensure these
*prerequisites are correct!
*It is YOUR responsibility to print anything you need to print before
*or after GET_EA.
*Nothing other than a1 and the value it points to will change.
*input: a6, d2.w, d3.b, a1
*Output: a6, a1, and the value pointed to by a1


LIKE_ANDROUTINE:
*Description:
*For op-codes that are structured like and,
*loads the size and the effective addressing modes into
```

```
*buffer and makes a6 point to the next op-code.
*Requires a1 to point to buffer.
*Assumes the effective addressing mode is valid.
*Assumes the op-code is printed.
*input: a6, a1
*Output: a6, a1, and the value pointed to by a1


SHIFTROUTINE:
*Description:
*A routine to handle shift op-codes like ASR/LSL and any more if necessary.
*Requires a1 to point to buffer.
*Requires a2 to point to the string of the shift ("ASR" for example).
*Calls COPY_STRING_A2_TO_A1 and handles the size and EA of the op-code.
*If the EA is invalid, calls DATAROUTINE.
*input: a6, a2
*Output: a6, a1, the value pointed to by a1
```

## Op-Code Subroutines

Each op-code has its own subroutine as mentioned in the guidelines. The label for the subroutine is the op-code name in all capitals with "ROUTINE" appended at the end (MOVEROUTINE).

Every op-code subroutine only handle that op-code. It reads a bytes from (a6) and loads what should be printed into A1. It increments a6 with the number of bytes it read (so a6 points to the next op-code by the end). All other registers and memory should remain unchanged.

Every op-code should start with comments that explain it. It should also start by backing up the registers it uses. At the end of a subroutine, the subroutine should restore the states of the registers.

Here is an example of a subroutine op-code:

```
LSLROUTINE:
*Description:
*Requires a1 to point to buffer.
*Loads "LSL" into BUFFER and handles its size and EA.
*If the EA is invalid, calls DATAROUTINE.
*input: a6
*Output: a6, a1, and the value pointed to by a1
    movem.l     a2,-(sp)

    lea         LSL_OPCODE,a2
    jsr         SHIFTROUTINE

    movem.l     (sp)+,a2
    rts

* End of LSLROUTINE subroutine
```

# Testing

I followed the principles of Test-Driven development as I mentioned in the guidelines.

At the beginning, I tested my I0 by printing it and by ensuring the program error-checks the input as it should.

The first two subroutines that I implemented are the NOPROUTINE and the DATAROUTINE. I implemented NOPROUTINE because it is easy. I implemented DATAROUTINE so I can test the other subroutines.

While developing, I had my tests at the top of my program. I had the program skip executing the tests, but at least they were assembled. For every op-code, I would write my tests. I ensured that the op-code is treated as data before I implemented its subroutine. After implementing the subroutine, I made sure that the new test and old tests succeed.

For important, key op-codes that represent other op-codes like AND (ADD and SUB are just like AND), I made a test file. I also implemented the effective addressing modes when I made AND, so I used that file to test the effective addressing modes.

## Challenges

Because the program required me to implement a limited set of op-codes, the program doesn't disassemble other opcodes. This causes my disassembler to fail in chases were an invalid op-code has an operand that can be confused as another op-code. These false positives can't be solved without implementing a full-featured disassembler.

It is annoying that there is no jsr with conditions.

The disassembler's DS directive can start a long from an odd index. It is implemented incorrectly. This causes the program to break if I try to access the long inside a variable because it is illegal to access a word with an odd index.

It was surprising to learn that the address BCC/BRA point to is the address of BCC/BRA + 2 + the sign-extended displacement.

I was horrified to learn that if I jmp to a line below the current line, jmp is three words. If I jmp to a line above the current line, jmp can be two words. This messes up with jmp tables.

I learned to be careful with LEA because it makes the address registers point to a memory. I had problems where I would load a string into a1. Then print the string. Then read input, only to discover later that the string was modified because a1 was still pointing to it. I learned to use a buffer.