•Beginners Mac Assembly•


Chapter 2


68000 instructions.

In the last chapter, we introduced the addressing modes, or the operand syntax. Now we'll look at some of the 68000 instruction set. This is not an alphabetical listing, or even a complete one. These are the basic everyday bread and butter instructions that are used most commonly.

Note - this section is quite demanding. If you are having problems with it, skip to "Program layout and structure" and come back to it later.

The first instruction is move. This code accounts for over 70% of all program content.

The syntax is:

```
      move.s
```

The .s indicates the size of the move, for example .b, .w or .l for byte, word and long. The move instruction moves data from the source operand to the destination operand, for example:

```
      move.w      d0,d1
```

This moves a word from data register 0 to data register 1.
Each instruction supports various addressing modes for both the source operand and the destination operand. For a full list of these look in the 68000 instructions glossary.
If you get an addressing mode wrong, the assembler will tell you.

The addressing modes for the move instruction are as follows:

For the source operand you can use:
Dn, An, (An), (An)+, -(An), x(An), x(An,r.s), x.w, x.l, x(PC),x(PC,xr.s),#x

where:
n is a register number between 0 and 7.
x is a number
r is a register - a5,d1 etc
s is the size - .b .w .l

For the destination operand:
Dn, (An), (An)+, -(An), x(An), x(An,r.s), x.w, x.l

Why is An missing from the destination operand addressing modes list?
Because there is an instruction called MOVEA, which moves to an address register.
However, Fantasm allows you to use the An addressing mode in the move instruction so "move.l d0,a3" is quite legal in Fantasm, but may produce an error in other assemblers.

Examine the following instructions. Work out what they do, and see if you can spot the illegal ones:

```
1:    move.b      d0,d1
2:    move.l      a3,d2
3:    move.w      (a4),(a3)
4:    move.l      14(a0),10(a6)
5:    move.w      d0,10(pc)
6:    move.w      #$20,d3
7:    move.w      fred,d0
8:    move.l      (a0)+,-(a1)
9:    move.l      d0,#fred
10:   move.w      20(pc),d7
```

```
1:    move.b      d0,d1
```
This instruction moves the byte from d0 to d1.

If d0 contained 25, then 25 is moved to d1
If d0 contained $1FF then only the $FF is moved to d1. $FF is 8 bits, making a byte.
To move the $1FF, we'd need to change the size to word - move.w

What would be the outcome of:
```
      move.w      d0,d1
```
if d0 contained $7FFFF. Would $7FFFF be moved into d1? If not, what would be moved into d1? - Q1 - answers below.

```
2:    move.l      a3,d2
```

This moves the longword in a3 to d2.

Address registers hold the addresses of data.

If a3 contained 12345, and memory location 12345 contained 19, what would go into d2?
Would the contents of the memory location be moved, or the contents of the address register?

In this case its the contents of the address register - 12345, so d2 would contain 12345 after the instruction is complete. If we wanted to move the contents of memory location 12345, we'd need the (a3) addressing mode.

```
3:    move.w      (a4),(a3)
```
From example 2, you should be able to see that this instruction says:
"move the contents of the memory location at the address in a4 to the memory at the address in a3"

If a3 contained the address $AF000 and a4 contained the address $10000, then whatever data, as a word was in location $AF000 would be moved to location $10000.

```
4:    move.l      14(a0),10(a6)
```
Getting more tricky now.

This instruction says:
"move the contents of memory at the address in a0 plus 14 to memory at address a6 plus 10".
If a0 contained 1000, and a6 contained 2000, then the longword (32 bits) at 1014 would be moved to 2010.
If 1014 contained the long word $12345678, after the instruction has finished, $12345678 would be in address 2010.

```
5:    move.w      d0,10(pc)
```

This instruction is illegal. Examine the list of addressing modes above. Can you say why its illegal? Q2

```
6:    move.w      #$20,d3
```
This is one of the simplest instructions. It simply moves the number 20 as a word into d3.

If d3 contained $FFFFFFFF before the instruction, what does it contain after the instruction?
Is it just $20?
No, it contains $FFFF0020 - why? Q3

```
7:    move.w      $54321,d0
```
This instruction moves the contents of memory location $54321, as a word into d0.
So if $54321 contained 123, then d0 would contain 123 after this instruction.
How would we move the number $54321 into d0? Q4

```
8:    move.l      (a0)+,-(a1)
```

This instruction moves the contents of the address (memory location) in a0 to the address in a1.
Then, a0 is incremented by 4 and a1 is decremented by 4 - is this right?
Q5

```
9:    move.l      d0,#fred
```
This instruction is illegal - why? Q6

```
10:   move.w      20(pc),d7
```
This instruction moves the contents of the address 20 locations from here into d7 as a word.
Would the instruction
```
      move.w      37000(pc),d7
```
be legal or not? Q7

Here are two more instructions:

```
11:   move.b      11(a6),d0
12:   move.w      11(a6),d0
```

Are these both legal instructions, and if so would they work (hint - one of them will not). Q8

Answers.

Q1:   The instruction was move.w   d0,d1
and d0 contained $7FFFF.

After the instruction had executed d0 would only contain $FFFF, because $7FFFF is more than a word. The largest number 16 bits (a word) can hold is $FFFF or 65535 in decimal.
As $7FFFF is larger than this, only the word part of it will be moved - $FFFF.
To move $7FFFF the instruction would need changing to move.l.

Q2:   move.w   d0,10(pc)
This instruction is illegal because, from the list of destination addressing modes for the

move instruction, the “program counter with displacement” addressing mode is not allowed - i.e. the (pc) addressing mode is illegal as a destination mode.

```
Q3:   move.w      #$20,d3
```
If d3 contained $FFFFFFFF before the move, after the move it contains $FFFF0020 because a word was moved, and data registers are not sign extended when words are moved into them. If the destination register had been an address register, a3, then the word would have been extended to a long, that is bit 15 would have been copied to bits 16-31.

```
Q4:   By changing the instruction to:
      move.l      #$54321,d0
```
Note the size has changed, and the # is needed to signify a real number.

```
Q5:         move.l      (a0)+,-(a1)
```

“This instruction moves the contents of the address (memory location) in a0 to the address in a1. Then, a0 is incremented by 4 and a1 is decremented by 4 - is this right?”

No its not - the first thing that happens is a1 is decremented by the size of the data - in this case 4 because its a long. Then the contents of the address in a0 is moved into the new address in a1, and finally a0 is incremented by 4.

```
Q6:   move.l      d0,#fred
```
“This instruction is illegal - why?”
The # character in the destination operand specifies a real number. You can’t move data to a real number - a number is an imaginary thing.

```
Q7: “Would the instruction
      move.w      37000(pc),d7
      be legal or not?”
```

No it wouldn’t, because the displacement (37000) is greater than ±32767. Displacements are signed numbers. This means that the most significant bit is the sign bit. If its a
1 then the number is negative. We only have 15 bits in a word to specify the displacement, which gives 32767 as the largest displacement possible.

```
Q8:
11:   move.b      11(a6),d0
12:   move.w      11(a6),d0
```
Both of these instructions are legal, however instruction 12 tries to move a word from an odd address. 68000’s like their words and longs on even addresses. When it tries to get a word or long from an odd address, the processor will generate an address error, which has the effect of crashing the computer, or if you’ve installed a debugger (see chapter 3), the debugger will allow you to find out what went wrong.

This highlights an important point. With machine code you have total control over the machine. It will do exactly as you tell it. When it cant, then there are no safety mechanisms that will prevent the processor doing what it thinks is right.

The add and sub instructions.

The next major instruction, add simply does just that. Adds something to something else.

```
      add.w       #10,d0
```
Adds decimal 10 to d0.

All addressing modes except SR and CCR are allowed for the source operand.
For the destination operand, x(PC), x(PC,xr.s), SR and CCR are not allowed.

The sub instruction works the same way, with the same addressing modes.

```
      sub.w       #30,fred
```
will subtract decimal 30 from the data in fred, where fred is a memory address.

You may be asking "How on earth can fred be a memory address".
Labels - see the next section.

Program layout and structure.

Now that we've covered three instructions, its time to examine how assembly language programs are laid out, and how they are written.

All the mnemonics (instruction names) that go to make up a program as a whole are called source code. Source code is written using an editor - any editor will do, so long as it will handle tabs, and not try to format the text into paragraphs.

Examine the following program.

```
*******************************************************
*FILENAME:      EXAMPLE1.S                                      *
*DESCRIPTION:  PROGRAM TO ADD 3 NUMBERS TOGETHER        *
*                       AND PRINT THE RESULT ON THE SCREEN.     *
*******************************************************
ADD_3_NUMBS:
        BSR             INIT_CURSOR    *part of io_lib.s
        MOVE.L #10,D0           *FIRST NUMBER
        ADD.L           #20,D0          *ADD SECOND NUMBER
        ADD.L           #30,D0          *ADD THIRD NUMBER
        BSR             PRINTNUM                *part of io_lib.s
        RTS

        INCLUDE         IO_LIB.S
**********************END OF ADD THREE NUMBS*******************
```

As you can see, the program is split up into four columns or fields. Between each field is a tab or space. This enables us to read the program more easily, and allows the assembler to find each field.

If a line starts with a splat "*", then the line is deemed to be a comment line. That is, the line is there purely to remind the programmer what is going on. The first five lines are all comments, informing us of what this programs source is called and what it does. The importance of comments can not be stressed enough, they explain what the program is doing at this point. You

will forget what pieces of a program do. By using comments you can help yourself work out what a piece of code is doing, that you may have only written a week ago!

The next line has a "label" - ADD_THREE_NUMBS:
This tells us and the assembler the name of this piece of code. Note that instead of spaces in the label, we use the underline character, which is shift and the minus key on your keyboard. Because spaces are used to separate fields, if a space appears in a label, the assembler will get horribly confused. The label ends with a colon. This is not mandatory, but it does make it easier for the assembler and us to identify labels.

As labels can be used in instructions as well (with out the colon), putting a colon on when a label is defined makes the label easier to find in an editor, using the find command. Labels can be used to label parts of a program, or locations in memory.

```
      BSR   INIT_CURS
```

This instruction branches to a subroutine called init_curs. The first part of any program is usually the initialisation. During initialisation, any data that needs setting up before the program runs is set. In this case init_curs sets up the cursors coordinates to the top left of a window. Where is init_curs? The last line of the program "INCLUDE IO_LIB.S" loads in a file called io_lib.s. Init_curs is part of this code.

After the initialisation, the instruction 'MOVE.L' moves the decimal number 10 into d0.
Notice again that between the 'MOVE.L' and the #10 there is white space - either a tab or spaces. After the operands (#10,d0) there is a tab, and then a comment, to remind us of what this instruction is doing.
The next two lines add twenty and thirty to d0.

As printing is a fairly common operation for a program to perform, we write a special program to perform the printing for us, then whenever we want to print a number we pass the number to the printing program, more specifically called a subroutine. We branch to the subroutine, which actually prints the number, and when the subroutine is finished, control returns to us.

Hence we have "BSR   PRINTNUM". How does the processor know where printnum is? Exactly the same way it found init_curs. At the last line of the program there is an instruction to the assembler to include another file called "IO_LIB.S". Whilst the assembler is assembling our program, when it scans this instruction, it reads in the file, and assembles that, exactly as if the lines of code were typed in by hand. One of the labels in this file will be called "PRINTNUM", which will define the start of the code for printnum.

So now the processor branches off to printnum, which will print d0 as a decimal number. When printnum has finished, the last instruction in is an RTS which tells the processor to ReTurn from Subroutine. It returns to line 11 of our program which is also an RTS, so the processor will return from our program - in effect ending it.

How does the processor know where to return to, when it executes an RTS instruction?
When a Branch to SubRoutine takes place, the address of the next instruction is pushed onto the hardware stack, so in this case its the address of the RTS at line 11.
The processor then branches to the subroutine, which carries out its task. Eventually the subroutine finishes with an RTS instruction. This instruction takes the address from the top of the stack, and puts it in the PC. The processor gets the instruction from the PC and execution continues from there - hey presto.

Educationalists have for years been promoting learning through practical experience, so.....

Load Fantasm. From the project menu select “Set main file name”. With the file selector open the folder named “Source”. In here select “Example1.s”, either by double clicking it or highlighting it and then clicking on “open”. The selector will go away, and you will be returned to the empty editor window. Nothing has happened, but now Fantasm knows the name of the file you will want to assemble. If you want to see the file, open it from the file menu - it will be loaded into the editor. Note that in Fantasm, the assembler and the editor are independent of each other. The reasons for this are explained in the Fantasm manual.

Now we need to tell Fantasm how to assemble this file - open the project menu again. You’ll see the file name of your program, and underneath this four output options, one of which will have a tick by the side of it. These output options tell Fantasm how you want the program built. The main two you’ll be interested in are “Output to Mac ram” and “Output as Mac app”.

“Output to Mac Ram” builds the program in memory, instead of on disk. This is handy for immediately checking code you write. As soon as Fantasm has finished assembling your program, it’ll ask you to press any key to run it. Your program will run, and when it finishes, Fantasm will regain control. If your program crashes, then Fantasm will also be terminated.

“Output as Mac app” creates as stand alone program on disk. The name of the program will be the name of the main source file you select in the project menu, but instead of having the extension “.s” it will end in “.app”. To run the program just double click it from the desktop.

In this case, for speed, we’ll run it in ram, so select “Output to Mac ram” from the project menu.

Now its time to assemble our program - from the Fantasm menu select “Assemble”.
A new window will open and Fantasm will assemble “example1.s” into memory.
Once it is finished Fantasm will ask you to “Press any key to begin execution.”
Press any key on the keyboard, or click the mouse button, and your program will run.

When the program is finished you’ll be asked to “Press any key to return to Fantasm”.
Before you do this, examine carefully what happened. What do we expect to happen.
The number 60 should be printed on the screen right? Well sort of. What's actually happened is that the number 60 has been printed in at the top left of Fantasm’s assembler window.

Because the print routine the program called, “Printnum”, uses the Mac to print characters to a window, Printnum has printed to the current window - in this case fantasm’s assembler window. What would happen if there wasn’t a window open?
On any Mac there is always a window open. If you don’t believe me try this.

From the project menu, select “Output Mac App”. Now click on assemble again in the Fantasm menu. The assembler will create an application called example1.app on your disk. When the assembler has finished, click on the desktop and look in the examples folder. Double click example1.app. What happens - where does the 60 appear?

What we really need is our own window for our program to run in.

This is example 2.

```
*********************************************************
*FILENAME:      EXAMPLE2.S                                              *
*DESCRIPTION:  PROGRAM TO ADD 3 NUMBERS TOGETHER          *
*                        AND PRINT THE RESULT IN A WINDOW.      *
*********************************************************

EXAMPLE2:       BSR             INIT_CURSOR    *part of io_lib.s
                LEA             WINDOW_TITLE,A0
                BSR             OPEN_WINDOW *PART OF IO_LIB.S
                MOVE.L A0,WINDOW_HANDLE     *OPEN_WINDOW RETURNS A HANDLE TO THE WINDOW

                MOVE.L #10,D0           *ADD 10 AND 20 AND 30
                ADD.L           #20,D0
                ADD.L           #30,D0

                BSR             PRINTNUM                *part of io_lib.s - PRINTS D0
                MOVE.L WINDOW_HANDLE,A0    *GET THE HANDLE FOR THE WINDOW
                BSR             CLOSE_WINDOW        *CLOSE THE WINDOW
                RTS
                INCLUDE         IO_LIB.S
****VARIABLES
WINDOW_TITLE:DC.B     “EXAMPLE 2”,0
WINDOW_HANDLE:      DS.L     1
**********************END OF EXAMPLE2********************
```

There are five new lines. The first line initialises the cursor, as in example one.
The next 2 lines call a subroutine - open_window - that displays our window on the screen.

```
            LEA         WINDOW_TITLE,A0
```

Open_window has to know the title of the window. It expects the address of the title string in A0. At the bottom of the program are the variables. Here WINDOW_TITLE is defined by using the dc.b (define constant bytes) directive. This tells Fantasm to store this string in memory, and give it the name “WINDOW_TITLE”. When we want to use this string in our program, we can simply use its name. Note that the string is terminated in a zero byte, and the actual characters of the string are enclosed in speech marks.

The mnemonic LEA stands for Load Effective Address. It loads an address register with an address, in this case the address of WINDOW_TITLE. It cannot be used with a data register.

Handles.

The next line calls a subroutine from io_lib.s to open a window. The Mac has to have a way of knowing which window is the target of any printing or drawing. For this reason when a window is opened, the mac, or to be more precise, the operating system, returns a “handle” for this window. The handle contains a unique identifier for this window. OPEN_WINDOW returns this handle in A0. As A0 could possibly be used later in the program we save the handle in the long variable WINDOW_HANDLE, which is defined at the bottom of the program with the directive “ds.l 1”. This tells Fantasm to reserve space in memory for 1 long word and call it “WINDOW_HANDLE”.

The rest of the program is the same as example one, apart from the branch to CLOSE_WINDOW, which is a subroutine in io_lib.s. It needs to know which window to close - there could be lots of windows open on screen - it cant just pick one at random. Like open_window, it expects the handle to be in A0, so we copy the handle from

window_handle to A0 with the line

```
            MOVE.L      WINDOW_HANDLE,A0
```

then we can call close_window.

Using Fantasm to modify the code to see what happens. The first thing to try is modifying the title name, to prove this is really the window title. Then you could try removing the "BSR CLOSE_WINDOW" instruction - what should happen and what does happen? Try creating an application with the close window call removed, and examining if and when the window ever disappears, and if it does, can you get it back again?

Position independant code, speed and size.

In Example 2, we used instructions like "MOVE.L WINDOW_HANDLE,A0". This is not the best way of moving data about. Why not? Firstly, window_handle is an absolute address. When the processor executes this instruction it has to read in a long to find the address of window_handle, so theres 2 bytes for the move instruction, and four for the address - six bytes in total. It would be faster if it only had to read four bytes.

Secondly, when a program is loaded from disk into memory, it doesnt know where its going to go. Because of this, Fantasm assembles everything starting at address zero.
When Fantasm has finished, window_handle will have been assigned an address of 20 (for example).

When the Mac loads the program, it has millions of addresses to choose from - the program will certainly not be loaded at address 0. Fantasm uses a special technique
to make a note of these addresses, and modify them accordingly before the program is run, but after it is loaded - this is called "relocating". However, if you are using Fantasm to output code to be merged into another program generated by a C compiler, then this code will crash because the address of window_handle wont be relocated.

The main consideration is speed. Machine code is about speed - call it pride.

As stated above - MOVE.L WINDOW_HANDLE,A0 requires 6 bytes of memory. What if we change the instruction to MOVE.l WINDOW_HANDLE(PC),A0. Does this alter the outcome of the instruction? No, but it does reduce the size of the instruction from 6 bytes to 4, which means it will execute faster.

Unfortunately, we cant do it the other way round! For example, the instruction
MOVE.W D0,FRED(PC) is illegal, The reason for this is when Motorola were designing the 68000 series they ommitted this addressing mode to stop people writing self modifying code - that is, code that alters itself. So how do we write this instruction without having to use an absolute long to define FRED?

We use an address register to hold the address of the start of our variables, and access the variables as an offset from the address register.

```
            LEA         VARIABLES,A6
            MOVE.W      D0,FRED(A6)
            MOVE.L D1,HARRY(A6)
            MOVE.B#2,GEORGE(A6)

FRED:       EQU         0           FRED IS A WORD
HARRY: EQU      2           HARRY IS A LONG
GEORGE:     EQU         6           GEORGE IS A BYTE
VARIABLES:  DS.B        7           MEMORY FOR FRED, HARRY AND GEORGE.
            EVEN
```

In the above, we use A6 to point to the memory used to hold FRED and HARRY. We use the "EQU" directive ("equates to") to define the offsets for FRED and HARRY. So FRED: EQU 0 means "from now on, FRED is the same as 0" and HARRY: EQU 2 means "from now on, HARRY is the same as 2. Why 2? Because FRED is a word. Hence GEORGE is 6 because HARRY is a long (4

bytes).
Note also the even directive at the end. This is necessary because VARIABLES is 7 bytes, which is an odd number. Any code after this must be on an even address, so we tell the assembler to make sure the address is even by using the "EVEN" directive.

Why is VARIABLES 7 bytes?
Fred is a word, which is 2 bytes. Harry is a long, which is 4 bytes. GEORGE is a byte, which is 1 byte. So 2+4+1 = 7.

You may find that when you write a large program, the number of offsets becomes quite large. Its easier if they live in a seperate include file, with good comments detailing the size of each offset and what the offset means.

The other option is to use the RS directives. These automatically calculate the size of the offset, depending on the size. An internal counter keeps track of the offsets, so the label is simply equated to the counter. For example:

```
fred:       rs.l        1
stu:        rs.b        1
cath:       rs.w        1
```

If these are the offsets for the variables that are accessed off a6, then
fred(a6) is the same as 0(a6), stu(a6) is 4(a6) and cath(a6) is 6(a6), not 5 because words and longs are always placed on word boundaries. For more information see the Fantasm user manual.

Here is example 2 re written to use position independant code:

```
*******************************************************
*FILENAME:      EXAMPLE3.S                                      *
*DESCRIPTION:  PROGRAM TO ADD 3 NUMBERS TOGETHER         *
*                       AND PRINT THE RESULT IN A WINDOW.      *
*                       WRITTEN IN POSITION INDEPENDANT CODE   *
*******************************************************

EXAMPLE3:       LEA             EX3_VARS(PC),A6         *A6 POINTS TO OUR VARIABLES
                BSR             INIT_CURSOR     *part of io_lib.s
                LEA             WINDOW_TITLE(PC),A0
                BSR             OPEN_WINDOW *PART OF IO_LIB.S
                MOVE.L A0,WINDOW_HANDLE(A6)         *OPEN_WINDOW RETURNS A HANDLE TO THE WINDOW

                MOVE.L #10,D0           *ADD 10 AND 20 AND 30
                ADD.L           #20,D0
                ADD.L           #30,D0

                BSR             PRINTNUM                *part of io_lib.s - PRINTS D0
                MOVE.L WINDOW_HANDLE(A6),A0         *GET THE HANDLE FOR THE WINDOW
                BSR             CLOSE_WINDOW        *CLOSE THE WINDOW
                RTS
                INCLUDE         IO_LIB.S
****VARIABLES
WINDOW_HANDLE:      EQU     0                       *WINDOW HANDLE .L
EX3_VARS:           DS.B    4                       *SPACE FOR 1 LONG
```

```
WINDOW_TITLE:DC.B     "EXAMPLE 3",0

**********************END OF EXAMPLE3*******************
```

When Fantasm assembles your code, it will tell you how many absolute long references it found. If there are none, then the code is position independant. In example 2, there are 3 absolute long references, can you spot them?

```
line 2 -          LEA             WINDOW_TITLE,A0
line 4 -          MOVE.L A0,WINDOW_HANDLE    *OPEN_WINDOW RETURNS A HANDLE TO THE WINDOW
line 9 -          MOVE.L WINDOW_HANDLE,A0    *GET THE HANDLE FOR THE WINDOW
```

In the next chapter, we'll have a quick look at all the basic 68000 instructions.