•FANTASM V1.01•

•The Macintosh™ machine code•

•development system•

•©Lightsoft August 1994•

Lightsoft are Stuart Ball and Robert (Zed) Probin. Currently in the process of moving to Mac land from the Atari ST.

Lightsoft is involved in simulation, comms, games, graphics and development systems.

This document is layed out for A4 sized printing.

1/4/25 16:58:51 - Document Version 1.01 release #1

•Beginners Mac Assembly•

Chapter 1

What is machine code?

Machine code is the code that your computer actually runs. Processors don't understand English, Basic, C or anything else recognisable to humans. Only ones and noughts (binary) - machine code is ones and noughts.

When your computer is running, it continuously gets instructions from memory. These instructions tell the processor what to do now. On a 68000 based machine, the instructions are made up of sixteen binary digits (bits), or 1's and 0's. 16 bits together are called a word.

The pattern of the word will tell the processor how to execute the instruction. A certain word may add two numbers together, whilst another word could make the processor store a number back in memory, and so on.
A program consists of lots of instructions strung together, and some data for the instructions to operate on.

Programs can be written in many kinds of different languages, some common ones are BASIC (Beginners All purpose Symbolic Instruction Code), C, and Pascal. These are termed high level languages. High level languages have to translate the terms used by the programmers into machine code. This is fine, except that the high level languages only know a certain way of doing things, and sometimes have to string together significantly long sets of ones and noughts to get the desired end result. What this means in practise is that any compiled language, will only be as good as the compiler.

Machine code on the other hand is a low level language. The difference between C and machine code is that an intelligent being is the compiler. Because a human is writing the code and not a machine, the code can be written in the best way to achieve maximum speed, and use the minimum of memory. For this reason machine code is always a lot faster than any compiled language.

-oo-

RP: This always strikes me as strange - we always think of machines better than humans - yet here is an application where they are not.
SB: I never think of any machine as being better than a human, besides how can you call a compiler a machine, when its plainly a piece of software, written by some faceless boring (deleted) who wouldn't know creativity if it smacked him in the face with a wet fish. Now assembler writers are definitely creative, intelligent, smart, good looking.....

-oo-

It would be impossible to program a 68000 in 1's and 0's. It was possible (and indeed done) to program earlier processors, such as the Z80, in machine code, since they were less sophisticated. It's more sensible to assign a mnemonic to each machine code instruction and use a computer program to translate the mnemonics into machine code instructions.

What's a mnemonic? The dictionary defines it as "something to help the memory" - in this case its a word that represents a machine code instruction.

The 68000 processor understands 56 basic instructions, which means the 68000 assembly language programmer has 56 different mnemonics to remember. Compared to a high level language, which may have as many as 800 different instructions, this is a small number of instructions to memorise. In practise not all these instructions will be used, and you may find yourself using as few as 20 instructions regularly, so its not difficult to learn (contrary to popular belief!)

What does a mnemonic look like?
The 68000 has an instruction to add two numbers together. In ones and noughts it looks like 1101000000000000, which is a mouthful, so we use the mnemonic - add. Its as simple as that. Other

mnemonics include SUB (SUBtract), MULS (MULtiply Signed), DIVS (DIVide Signed) and MOVE. There are 56 of these instructions that can accomplish various set functions. Because the instructions are very simple, to accomplish anything complicated means stringing lots of instructions together, to form a machine code program. From now on we'll use the term machine code and assembly language to mean the same thing - mnemonics.

-oo-

RP - The reason for the incorrect technical usage is that programmers tend to use these words interchangeably because firstly, they know what they mean, and secondly, because by using assembly language you are (with an assemblers help) using machine code directly.

SB - What incorrect technical usage?!? I don't use these word interchangeably? I just say assembly language. Are you going to keep this up right through this document?

-oo-

To show you how easy it is, examine the following 68000 assembly language program.

```
add:        move.w      #10,d0
            move.w      #20,d1
            add.w       d1,d0
            move.w      d0,sum
            rts
sum:        ds.w        1
```

The first thing to notice is the layout. All assembler languages tend to be like this. The doing bits of the instructions (move, add, etc) - the actual instruction is in the 'second column'. Technically it is proceeded by white space - either tabs or space characters. This way the assembler knows they are instructions.

So what are the things up against the left hand edge of the page? Labels. They mark a position in the code. Notice they have a colon (:) after each one. When you are using

these labels in the code, you don't actually use the colon - for example in the line "move.w d0,sum".

The colon is not essential when you are defining a label in the first column,but it helps in two ways. Firstly it clearly identifies this thing as a label, both to the assembler AND humans. Secondly, it can be helpful when you are editing the program, and need to find a specific label. If you search for just "sum" you will reach all the places that access sum - but if you search for "sum:" you will go to the place where the label is defined.
After most of the instructions .w is tagged on. This basically defines the size - in this case words - to operate in - more in a while. Then some more white space. Then the operands. These are the things the instructions manipulate.

A note on layout - tabs are good because they line up the columns and make it look neat.

Any ideas about what this program does? It adds 20 and 10 and puts the result in sum. Here's the breakdown.

```
Line 1 - Add:           move.w      #10,d0
```

The 'Add' is a 'label'. Labels are used to reference lines of a program so we can change the program flow, by 'jumping' to labels. Think of it as a name for this part of the program.
Move.w means move a word. In this case it means move the word '10' (remember that a word is sixteen bits) into the processor.

Line 2 - move.w   #20,d1
This line has no label, and therefore cant be jumped to.
The instruction moves another number,in this case 20, into the processor.

Line 3 - add.w   d1,d0
This line instructs the processor to add the two numbers together.

Line 4 - Move.w   d0,sum
This instruction moves the result of the addition into memory. Where in memory? At sum.

Line 5 - rts
This instruction returns from this part of the program. if you like it means that's the end of this piece of code.

Line 6 - sum: ds.w   1
This line is used to define the location of 'sum' in memory. Its not an instruction to the processor, as the processor has finished with this code in line 5. This line is used by Fantasm to set up 'sum' in memory ready for the program when it is run. The 'sum' tells Fantasm that this is the name we want to use. Again it is a label.
The ds.w means define space as words. Fantasm reserves the number of words needed for this label in memory. This is called a 'directive', meaning that it is a directive to the assembler (Fantasm), and not an mnemonic to be translated into machine code.
If you found that complicated, don't worry as we'll come back to directives later.

If you had trouble understanding that, read through it again. There's nothing devious or particularly clever about machine code programming - just common sense most of the time!

If after re-reading the above you still have difficulty understanding it, don't worry about it, just carry on with this text, as it was throwing you in at the deep end a bit!

Bits, bytes, words and longs.

As you may already know, all computers have 'memory'. Memory is a very wide term, that can be broken down as follows.

1. Long term memory. This type of memory is for long term storage of information.
Theoretically speaking it means any form of memory that doesn't forget when you switch the power off. This boils down to disks, both floppy and hard. Once the data is written to disk, it stays there until its either overwritten, or your four year old decides to open the case on one of your floppies and play frisbee with what's inside.

There are big differences between floppy disks, and hard disks. Floppy disks are removable, cheap and slow. If you slide the metal cover on one of a floppy you'll see a brown plastic disk inside the jacket. This is the actual floppy disk. Data is recorded onto the disk, in the same way that music is recorded onto a cassette, one bit at a time.

Because the disk is arranged in tracks (80 on a HD disk), and your Mac can select any one of these tracks at random, you don't have to fast forward over the whole disk to get at the data you want. The big disadvantage with floppy disks is that they are very slow compared to hard disks.

Hard disks are not (generally) removable, not cheap and not slow.
(They are slow in comparison to memory thou - RP)

A hard disk works in a similar way to floppy disks. They have tracks, and head(s), but because the disk spins a lot faster (3600 rpm) and the heads are a lot closer to the disk, you can store a lot more information on them.
Hard disks have a reputation for being fragile. This is not just paranoia about them, but a fact. In a hard disk, to get the heads as close as possible to the recording surface, the heads fly on a cushion of air created by the disk spinning. If the hard disk drive is knocked whilst it is in use, its quite possible for the heads to crash into the disk surface! This is a good reason why you should use the shutdown menu item in the finder, so the drive can move the heads away from the disk before the power goes off and the disk slows down.

All disks, whether hard or floppy segment the tracks up into sectors. This makes the drives more usable. For example if you have a high density floppy disk, capable of holding 1.4 megabytes (1 megabyte is 1 million bytes), and the floppy disk itself has 80 tracks, this means that each track can hold 0.0175 megabytes, or 17.5 kilobytes.
This means that the smallest amount of information that could be written to a floppy disk is 17.5K. If a 1k file was saved to disk, it would still take up 17.5K on the

floppy! If however, each track is further split up into sectors, and the drive knew where each of these sectors were, then small files would take up less space. For example if the track was split up into 20 sectors, then the smallest addressable unit on the floppy would be 17.5 K divided by 20, which is 875 bytes. Now a 1k file only takes up 2 sectors on disk amounting to 1.75K.

Getting data off a disk is a slow process - far too slow for the processor. The data in long term memory, such as a program, is transferred to short term memory before it can be run, or accessed by the processor.

The other type of long term memory, associated with Macintoshes, is pram, or parameter random access memory. This memory doesn't forget its data when a Mac is switched off.
It has a battery that keeps the memory running. In here the Mac stores vital information - its configuration, so that then next time its switched on, it can read the set up information from pram, and configure itself exactly as it was. Examples of the data stored in pram are the sound volume, how many flashes a menu bar makes, keyboard repeat speed etc.

(Other computers may have this type of memory. PC-compatibles use a type of memory that is usually referred to as CMOS memory. Using this name is a bit naughty, since it should be called "battery backed cmos memory" - cmos memory itself loses its contents without power. In PC's this stores things like hard drive type, floppy types, time and date - general set up in formation. - RP)

2. Short term memory - this is the memory the processor has direct access to. It is split into random access memory (RAM) and read only memory (ROM). Ram can be written to and read from, whereas ROM can only be read from.
The more ram you have, the more data can be stored inside the computer at any one time.
If the data is a program, then the more ram you have, the bigger the program you can run.

As was noted previously, the 68000 range of processors use 16 bit instructions. The basic unit of memory is a byte, which is 8 bits, so the 68000 needs to read two bytes for every basic instruction.

Ram and Rom. are fast enough for the processor to run programs from. For the processor to carry out one instruction can take a minimum of 4 clock cycles (ticks), so the fastest it can execute one instruction is .00000025 of a second, or 250 nanoseconds, assuming a 16 megahertz clock speed. This could mean that every 250 nanoseconds it needs an instruction from memory, so the memory in your Mac has to be faster than that. If it was slower then the processor would have to wait.

Its interesting to note here, that the speed quoted by simm makers, like 100ns is only the setup

time, not the total access time.

More on RAM.

As you know, computers these days are referred to as being digital. Digital means numerical, so it stands to reason that computers run by using numbers? Quite true, as a number is just a number, but can also be used to represent a code for either an operation, or a letter, or a sound volume etc.

So if a computer needs numbers to strut its stuff, what form are these numbers held in, do we just say 10, and it puts letter up on the screen? Not quite. The numbers the computer needs to carry out processing are held in the binary format. Binary is a number system in which a number can be one of two values, either 0 or 1, whereas denary means a number can have one of ten values, 0 through to 9.
How can we represent numbers larger than 1 in binary?
Lets look at our human denary system first.

If we start counting from zero upwards, we eventually get to 11, which means 10 plus 1, then when we get to 101, this means 1 hundred, no tens and 1 unit. If we term
tens, hundred's and thousands as multipliers, then any number can be expressed in terms of its multipliers - 1234 can be expressed as:
1 times 10^3
+
2 times 10^2
+
3 times 10^1
+
4 (note that in programming, * means multiply and ^ means to the power of
10^3=10*10*10).

As the computer uses binary and not denary, numbers can be expressed as powers of 2 instead of ten. Where in denary the multipliers go 1,10,100,1000,10000 etc, in binary the multipliers go 1,2,4,8,16,32,64,128,256 etc.
To express the number 13 in binary, break it down into powers of two.

13 is one 8, one 4, and 1 -
8 4 2 1
1 1 0 1

Thirteen is 1101 in binary. Now try 255.

Start with the multiplier above 255, 256. This is too great, so try 128.
255 divided by 128=1 remainder 127. So we have a 128.
127 divided by 64 =1 remainder 63.
63 divided by 32 = 1 remainder 31
31 divided by 16 = 1 remainder 15
15 divided by 8 = 1 remainder 7
7 divided by 4 = 1 remainder 3
3 divided by 2 = 1 remainder 1
1 divided by 1 = 1 remainder 0

Therefore 255 in binary is 11111111.

The last example is 471. 512 is too big, so:
471 div 255 = 1 remainder 215
215 div 128 = 1 remainder 89
89 div 64 = 1 remainder 25

25 div 32 = 0
25 div 16 = 1 remainder 9
9   div 8 = 1 remainder 1
1   div 4 = 0
1   div 2 = 0
1   div 1 = 1

471 in binary is:
111011001 - that's nine bits. I admit, it isn't easy, but here are some more examples expanded to 8 bits.

	128	64	32	16	8	4	2	1
25 =	0	0	0	1	1	0	0	1
129=	1	0	0	0	0	0	0	1
56 =	0	0	1	1	1	0	0	0
90 =	0	1	0	1	1	0	1	0

To show that a number is binary, we precede it with a percent sign - %1010101

Eight bits are termed a byte. One byte can hold 256 different values, so every conceivable letter and punctuation mark can be defined with in one byte, or alternatively, 256 different machine code instructions can be defined, or 256 different colours for a pixel.

As we know, the 68000 likes its instructions in 16 bit chunks, these are called words
- how many possible values are there with 16 bits?

2^16 = 65536 (easy with a calculator).

With 16 bits 65536 different values can be defined, so if for example you have a Mac that can generate 16 bit colour, then you know it can display over 65000 different colours.

With 16 bits making up a basic 68000 instruction, there are possibly over 65000 different instructions the processor could execute - in practise there are 56.

16 bits make up what's called a word. However 16 bits aren't really enough for big numbers. For example if your Mac as 4 megabytes of memory - that's over 4 million bytes, we cant locate them with only 16 bits of information, as that only allows us to count up to 65536.

So we have long words - two words strung together to form 32 bits. With 32 bits, there are 4,294,967,296 possible different values (4.2 gig). The 68000 has a 32 bit addressing capability, which means it can (theoretically) access over 4.2 gigabytes of data!

This is what it means when you switch on 32 bit addressing in the memory control panel - basically it allows the cpu to access more than 8 megabytes of memory, as with 32 bit addressing switched off, only 24 bits are used, which allows the cpu to access only 8 megabytes.

Hexadecimal notation.

Hex is just a number system, the same as decimal and binary. In decimal we use base 10, in binary we use base 2, and in hex we use base 16. How can we use base 16 if we only have ten digits (0-9). Good question. The digit set is extended to 16 by using the letters A-F.
If you can understand binary, hex isn't a big deal. Each hex digit represents 4 bits or a nibble.

An example is probably easiest to understand:

Here's an easy one 255 in binary is 11111111 (8 ones).

To get 255 in hex first convert to binary, then split it up into nibbles,
1111 1111.

Each hex digit is a nibble, so 1111 is 16 in decimal, or F in hexadecimal.

Therefore 255 is 11111111 in binary or FF in hex. To show this is a hex number we precede it with a dollar sign - $FF

To convert from hex to decimal, convert the hex to binary, then decimal:
convert $FACE to decimal -

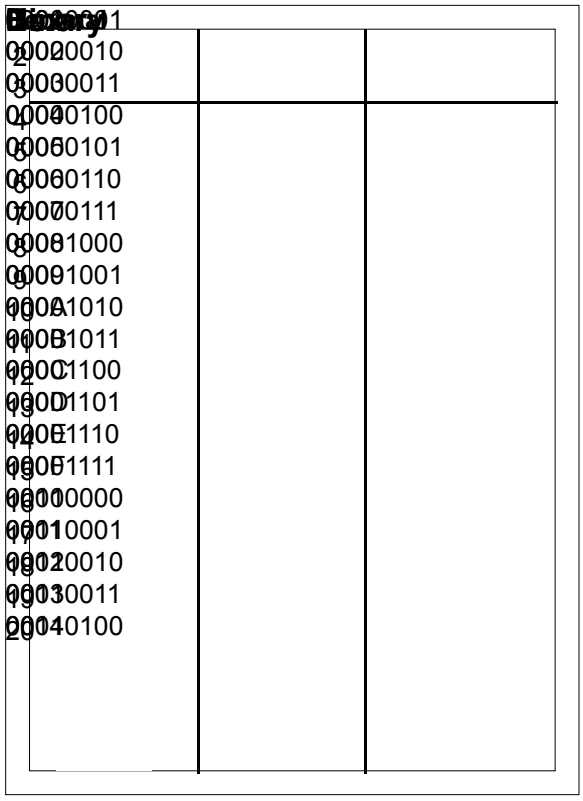
```
F    A    C    E
1111 1010 1100 1110

32768  16384  8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
  1      1     1    1    1    0    1   0  1   1  0  0 1 1 1 0
```

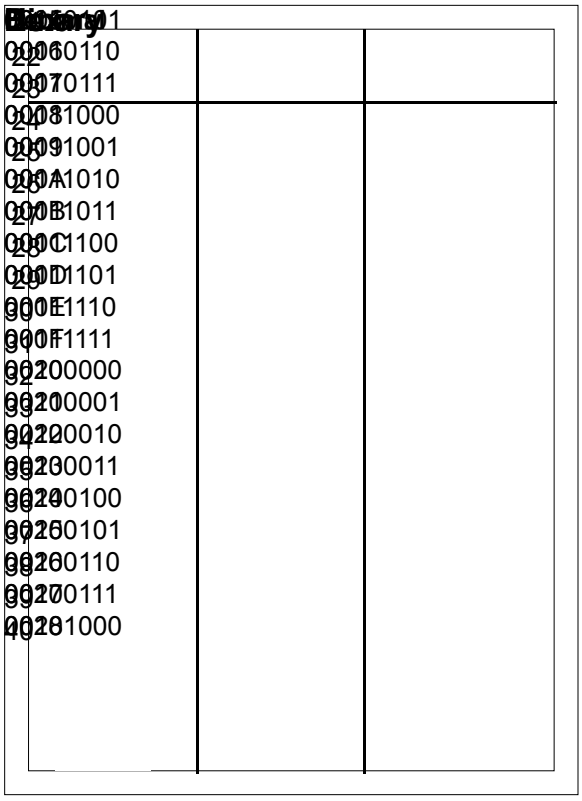
then add together all the ones:
32768+16384+8192+4096+2048+512+128+64+8+4+2 = 64206

Now try $9276
You should get 37494. Of course the quickest way is to use a the computer to do it for you. Normally I just drop into macsbug! (a debugger).

(How about a list of numbers, say 1 to 20, with binary, hex, decimal in? - RP)

Damn fine idea - here we go:





The 68000 family can manipulate numbers as either bits, bytes, words or longwords.

Now that we know a little of how the computer uses numbers, how are they stored in memory? Because the numbers are represented in binary, which is a string of ones and noughts, its easy to go from theory to practise. In memory a 1 is represented by a voltage, whilst a nought is represented by no voltage. The memory is laid out in bytes (8 bits), so for the processor to get one instruction it needs to read two bytes.
Two bytes make up one word, so it stands to reason that words must live in even locations in memory, if the memory starts at location 0 (which it does).

A quick summary:
The number system used in computer is binary. Binary means one of two values. Either a digit is a one or a nought. When counting in binary, the power of two is applied as a multiplier. A bit is one binary digit, either a 1 or 0. A byte is 8 binary digits that can hold 256 possible values. A word is 16 bits that can hold 65536 possible values,
and a long word is 32 bits that can hold very big values.

Addressing.

How does the processor know where the data is and how does the data get from the memory to the processor and back again?

All computers have busses. Highways for information. Typically there are a data bus and an address bus. The address bus tells the memory where the processor wants to read or write to. To get a word from memory location 1000, the processor puts 1000 on the address bus, then tells the memory to "read". The memory will put the data on the data bus, and the processor can read it in. From now on, instead of the word "location", we'll use the word "address" to mean a location that the

processor can access.

Inside the processor the address is stored in a "register". The 68000 has 8 of these address registers, each being 32 bits wide. Programs can modify these registers, so that the processor can keep temporary pointers to memory locations.
A register is like a small piece of memory that is internal to the processor, and hence very fast. Just as we have 8 address registers, we also have 8 data registers. The difference between address registers and data registers is that address registers are used for referencing memory locations, whilst data registers are used for holding and modifying data, for example dividing can be carried out on a data register, but not on address register. Address registers are called A0 through A7 and data registers are called D0 through D7 (surprise!).

-oo-

(Hold on matey, you cant throw in a word like "referencing" with me noticing! - Explainez vous - RP)
I knew you'd pick that up! Ok, so referencing means pointing to something - in the case of an address register, it means it points to a memory address. Good enough?

-oo-

An important point to realise is that its not only memory and the processor that can access the address and data busses. Most peripherals, such as disk controllers, keyboards, screen driver hardware also access these busses.
These peripherals are normally given a memory address that's well out of the way of the memory, so if the computers main memory ends at address $1000000, the peripheral hardware addresses may start at $80000000. If the video driver hardware lives at $80000000, the processor can send and read data by reading and writing to this address.
We'll come back to peripherals later.

Program counter.

There is a special address register called the program counter. This one keeps track of where in a program the processor is. Normally it increments by the size of each instruction, as each instruction is read in.

-oo-

(What does "increments" mean then? RP in his best beginners voice)
Oh dear, increments means it increases - something is added to it. In this case the size of the last instruction.

-oo-

Example.
If the program starts at address 1000, and the first instruction is 4 bytes long, after the processor has read in the instruction, the program counter will be pointing to location 1004.

How, if the 68000 uses 16 bit instructions, can the instruction be four bytes long?????

Well the basic instruction is two bytes, but there may be data associated with the instruction. Suppose we have an add instruction, that performs the addition of 5000 to data register 1 (D1). The first word of the instruction (two bytes) says "add to D1",
and the next word contains the 5000.

68000 instructions come in varying lengths, from 2 to 14 bytes. The shorter the instruction, the quicker it runs! Its useful to know the instruction lengths, as sometimes its very possible to change one instruction for another if you need the speed. This is sometimes called optimising - I always do it after a program is written.

As an example, by optimising Fantasm, it was speeded up by fifteen percent.

Normally, the program counter (PC) is incremented by the length of the current instruction, to point at the next instruction. However programs need a way of making decisions, and going off to do something else if need be. This is called branching or jumping. As an example consider a program that accepts names from the keyboard until
ten names have been entered.
The program could go something like this:

```
step 1:     get name 1 from keyboard
step 2:     print the name on the screen
step 3:     get name 2 from keyboard
step 4:     print the name on screen
step 5:     get name 3 from the keyboard
step 6:     print name....
step 7:     get name 4....
step 8:     and so on....
```

As you can see the program is a repetition of steps 1 and 2. What would be
nice is if we had a way of using steps 1 and 2 ten times over. By using a conditional check and a counter we can:

```
step 1:     set counter to 1
step 2:     get name from keyboard
step 3:     print name on screen
step 4:     add 1 to the counter
step 5:     is the counter equal to 10?   This is the conditional check
step 6:     if no, then go to step 2
step 7:     end the program
```

Step 6 is a conditional branch - it is taken if the condition is met - if the counter doesn't equal 10 then branch to step 2.
The processor would have to scrap the contents of the program counter and replace it with the memory location for the instruction at step 2.
To be able to perform conditional branching, or jumping the processor has to have a method of flagging the result of operations.
In the above program the processor needs to know if the counter had reached 10.
How does it do this? It compares the value of the counter to 10. If it equals 10 the processor sets a flag in the "condition code" register. At step 6 this flag is checked.
if the flag isn't set, then the program can branch back to step 2

Stack pointers

As was stated earlier, the 68000 range has 8 address registers numbered 0 to 7. One of these, address register 7 (A7), is referred to as the "stack pointer".
Stacks are a very important concept, that need to be at least partially understood.
A stack is an area of memory, where temporary data can be stored. Data is moved onto a stack, and is also moved off. When an item is moved onto the stack, the stack pointer moves down by the length of the data. If a long word is pushed onto the stack, then the stack pointer moves down 4 bytes, if a word is pushed on, then the stack pointer (SP) moves down by 2 bytes.
When data is moved off the stack, the stack pointer moves up by the length of the data.
This means that a stack operates as a last in, first out buffer. If two word A and B are moved

onto the stack (in that order), B will be on top of the stack. When a move from the stack occurs, it will be word B. The next move from the stack will produce word A.

Assume the SP is pointing at 1000 in memory, and D0 contains 55. If we tell the processor to move the word in d0 onto the stack, memory location 1000 gets 55 put into, and the stack pointer now points at memory location 998. When we get the data back off the stack, the SP will now contain 1000 again. Any of the address registers can be used as a stack, but A7 is the hardware stack pointer. Whenever the processor needs to put something on the stack, it will use A7. If the abbreviation SP is used then A7 is used by Fantasm.

As an example (see if you can follow this through, I know I haven't introduced the various instructions yet...)

```
1:    move.l      d0,-(sp)
2:    bsr         do_something
3:    move.l      (sp)+,d0
```

This example shows how to preserve the value of D0, as a long, whilst the program branches to do_something. do_something is what is called a subroutine. Generally used to carry out common tasks needed by a program, and to make a program a lot more readable! When the subroutine is finished, control will return to line 3. As the subroutine may well have altered the value of D0, it is restored by moving its contents back from the stack.
Its very important to note, that if data is put on the stack, then it must also be removed, in the right order.

How does the processor know where to go when the subroutine (do_something) finishes?
When the processor comes across a BSR instruction, it moves the address of the next instruction onto the stack, in this case the address of instruction 3. Then it branches to the subroutine.

The subroutine will end with the instruction RTS - return from subroutine. This will cause the address on the stack (line 3 in this case) to be moved into the program counter, and the processor will continue from there. The subroutine would look something like this:

```
do_something:     do some processing
                  .......
                  rts
```

We'll come back to subroutines and stacks later.

Before we dive in...

These are some of the terms that will crop up pretty soon:

Operand:
Data on which an instruction operates. For example
move.w d0,d1 - d0 is the first operand, d1 is the second operand.

Sign extension:
How does one specify whether or not a number is negative or positive in binary numbers?

The most significant bit (31 for a long, 15 for a word, 7 for a byte) is deemed to be the sign bit. If its a 1 then the number is negative, if its a 0 then the number is positive. Sign extension simply copies the sign bit. For example sign extending a word copies bit 15 into bits 16 to 31 of the long word

68000 architecture.

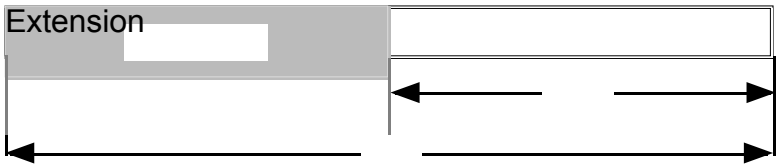
Now that we've covered the basic terms and buzz words, its time to introduce the beasty that powers your Mac, from a programmers point of view.
One of the first things most programmers want to know is how many registers does the processor have, are they interchangeable, and what size are they?
The number of registers is important, as register computations are faster and use less code than memory computations. The more registers a machine has, the better.
As was stated earlier the 68000 series has both data and address registers. Generally the are not interchangeable. Data registers hold data, and address registers hold memory addresses.

Address registers.

There are 8 address registers, numbered A0 to A7. Address registers normally hold a 32 bit quantity, but can be used as 16 bit quantities. When a word is loaded into an address register, it is sign extended to 32 bits. This means that bit 15 is copied into bits 16-31. Bits are always numbered from 0 upwards, thus bit 0 is the least significant bit (LSB) and bit 31 is the most significant bit (MSB).
The notation A0.W is used to indicate the word part of an address register. The suffix .B (for byte) is not allowed for address registers.

Examples:

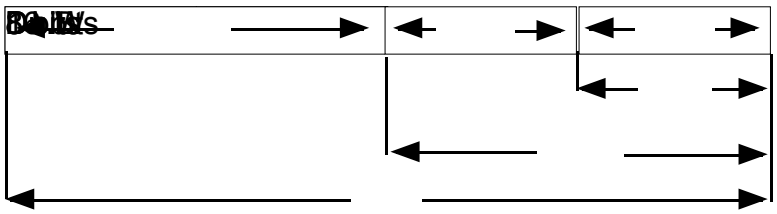
```
      move.l      d0,a2       - this moves 32 bits from data register 0 to address reg 2.
      move.w      d1,a3       - this moves 16 bits from D1 to A3. Once in A3 bit 15 is
                              copied to bits 16-31
      move.b      d2,a1       - this is illegal because bytes aren't allowed in address
                                regs.
```



Address register layout.

Data registers

The 68000 family has 8 data registers umber d0-d8. A data register can be used as an 8 bit, 16 bit or 32 bit quantity. Unlike address registers, loading a data register with less than 32 bits (a byte or a word) does not cause sign extension to take place. The remaining bits are unchanged.
Data registers cannot be used to address memory in an instruction.



Data register layout

Program counter

A 32 bit register is used to control the execution of a program running in memory. It always points to the address of the next instruction to be executed. As the processor executes each instruction, the PC is advanced to the start of the next instruction. However certain instructions cause the PC to be altered.

These are:

- Instructions that cause the PC to be altered unconditionally. These are called branches or jumps. For example:

```
bra   fred
```

causes the PC to be loaded with the address of fred, and execution will continue from there. Note that branches are limited to a range of ± 32767. If the range is greater than this, then a jump must be used. Jumps are slower as the branch instruction specifies a 16 bit offset, whereas a jmp must specify a 32 bit address.

- Instructions that alter the value of the PC depending on the result of a previous calculation. These are called conditional branches or jumps. For example:

```
      beq         stu
```

causes the PC to be loaded with the address of stu, only if the result of a previous calculation was equal to zero. Conditional branches allow parts of a program to be skipped, or loops to be executed if a condition is satisfied.

- Instructions that cause a given portion of code to be repeated a specific number of times, or until a condition is satisfied. For example:

```
            move.l      #9,d0             *move the long number 9 to d0
loop:       sub.l       #2,d1             *subtract the long number 2 from d1
            dbne        d0,loop           *decrement d0 and branch if not equal to zero
```

The dbne means decrement d0 and branch to loop if not equal to zero.

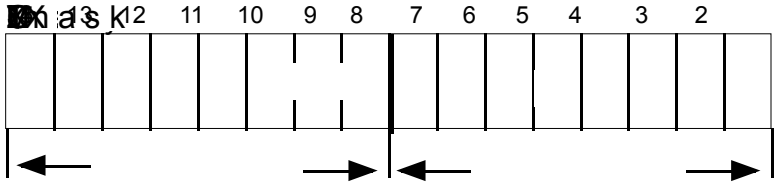
- Instructions that are used to branch to a subroutine, and then return to the instruction following the original branch. Example:

```
      bsr         getkey
      tst.w       d0
```

Branches to the subroutine called getkey. When getkey finishes, the processor continues with the tst.w (test) instruction.

Status register.

Previously we've mentioned conditional operations. That is to say, something will only happen if a certain condition exists. These conditions are checked for by examining the status register. The status register stores information about the state of the processor. This is the register used by conditional branches to find out the result of the previous instruction.



Status register layout

System byte.

Bits 15 to 8 of the status register are called the system byte.

- Bit 15 is called the trace bit. After every instruction, an exception occurs which allows debuggers to regain control, if the trace bit is set to a 1.

- Bit 13 is called the supervisor mode bit. The 68000 can operate in two modes. User and supervisor. In user mode certain instructions that could affect system integrity are not allowed. In supervisor mode (S bit set to 1) these instructions are allowed. When in user mode access to the system byte is prohibited, so that user mode programs can't switch into supervisor mode.

- Bits 10 to 8 are called the interrupt mask. These will not be dealt with here.

User byte.

The user byte carries 5 bits known as the condition codes, which are flags used to record the outcome of the last arithmetic operation performed.

- The C flag is the carry flag. If two 8 bits numbers are added, the C bit is the ninth bit of the result.

- The V flag is the overflow flag. If a result overflows, then the v flag will be set. Overflow means that the result could not be properly represented. For example, suppose we add two bytes together. The MSB of each byte is the sign bit.

  If both numbers were positive, then bit 7 of both numbers would be clear. The numbers are 127 and 2. The biggest positive number in a byte is 127, so adding 2 to it makes the answer = 129. 129 means bit 7 is set, which makes the number negative. The 68000 recognises this and sets the V flag to indicate that the result overflowed.

- The Z flag is set if the result is zero.

- The X bit is is a copy of the carry bit, but is not affected by all the instructions that affect the carry bit. The main use for the X bit is in multiprecision instructions, whereby data larger than 32 bits is being operated on.

The lower half of the status register is sometimes called the condition code register (CCR) because all it contains is the condition codes.

68000 data types.

The 68K can work with several different data types: binary, binary coded decimal (BCD) and floating point. The binary types we've already discussed - bytes, words and longs.
BCD is a form where numbers are defined in blocks of 4 bits each. For example $99 would be 153 in decimal normally, but in BCD it is 99 decimal. The 68k can add and subtract BCD numbers in this format. Its handy to know the facility is there should you ever need it! Floating point numbers wont be discussed in this document.

ASCII

Maybe you've heard of this term before? It stands for American Standard Code for Information Interchange. This code assigns a value for each character. Characters are simply codes that

can be printed - EG $41 is an "A", whereas 13 is a carriage return, and 10 is a linefeed. All ASCII characters can be represented in a byte. A sequence of ASCII bytes is called a string. The 68K has no specific instructions for dealing with strings, they must be dealt with by byte manipulations. There are three common ways of storing strings.

- Use a fixed amount of storage for each string. This has the advantage that all strings are the same length, but wastes memory as every string takes up the same amount of space as the largest string. EG,
  FRED0000000000

- Precede the string with a byte count. Before the string is processed the first byte is read, which indicates the length of the string. This is commonly used in Pascal and some Basics. EG.
  4FRED
  The Macintosh operating system (system 7) uses this format for strings.

- Terminate the string with some unique character, that wouldn't appear in the string - normally a zero byte. This is the convention used in C, and is my personal preferred method. EG.
  FRED0

What are addressing modes?

A computer instruction has to specify two things.

1. What operation to perform - i.e. add, subtract etc
2. On what data to perform the operation.

The op code specifies the operation - move, add, sub etc. The operand specifies the data.

```
move.b      d1,d7
op code   operands
```

For a computer to be useful it has to be able to access data in a variety of ways.
In the example above d1 and d7 are both registers. If the instruction was:

```
move.w      d1,$1000          *move the word in d1 to memory address $1000
```

then the source operand is d1 and the destination operand is address $1000. The way the operands are constructed is called the addressing mode. Instructions allow data to be accessed in different ways, and these different ways need names so we can work with them.
Don't worry about trying to remember the names of the addressing modes, just try to get a hang of the syntax of the operands.

Data register direct addressing.

This is the simplest form of addressing. Here the operand is a data register.

```
add.l       d0,d1             *adds d0 to d1.
```

Both the source operand (d0) and the destination operands (d1) use data register direct addressing.

Address register direct addressing.

Exactly the same as data register direct addressing, except this time the register is an address register.

```
      sub.l       d7,a1             *subtracts d7 from a1
```

The source operand uses data register direct addressing, and the destination operand uses address register direct addressing.

Address register indirect addressing.

This method allows access to the data in the memory location held in an address register.

```
      move.l      (a2),d0           *move the contents of the address in a2 to d0.
```

If a2 contained $1234, which because its an address register, is a memory location.
In address $1234 is the longword data $55555555. This instruction causes d0 to be loaded with the $5555555 from address $1234.

Another example...

```
      move.l      #$2001,a0         *move the long number $2001 into a0


      move.w      #1,d0             *move the word 1 into d0
      move.w      d0,(a0)           *move d0 into memory location $2001
```

Address register indirect addressing with post increment.

What a mouthful. This is a great addressing mode, one of the ones that makes the 68000 a powerful processor. Its basically the same as the previous mode, except that after the data has been moved, the address register is incremented by the size of the data.
This makes it very handy for moving strings, lists or blocks of data.

```
      move.l      d0,(a3)+          *move the long in d0 to the memory location in a3
                                    *then add 4 to a3 (because it was a long data item)
```

Another example:
Assume a0 is pointing to a string - hello mother,0 - and a1 is pointing to empty memory.

```
      move.b      (a0)+,(a1)+       *move one byte from the memory location in a0 to the
                                    *memory location in a1, then add 1 to a0 and add 1 to
```

*a1.
This instruction has moved the “h” in the string “hello mother” to the empty memory pointed to by a1, then added 1 (because a byte was moved) to both a0, and a1. a0 will now be pointing at the “e”, and a1 will be pointing at an empty byte in memory.
This instruction routinely forms the core of code that moves chunks of data about.
Note that a special case occurs when a byte is moved onto the stack using this addressing mode. In this case the stack pointer will be incremented by 2 to ensure the stack pointer always points to an even address.

Address register indirect with pre decrement.

The counter part to post increment. Here the address register is decremented by the size of the data before the data is moved.

```
      move.l      d0,-(sp)          *decrement the stack pointer (a7) by 4, then move d0 to
                                *the address held in the stack pointer

      move.b      d7,-(a1)          *decrement a1 by 1, then move the word in d0 into the
                                *address held in a1
```

A special case occurs when a byte is moved onto the stack, in that the SP is decremented by 2, so the stack pointer remains on an even address, as words and long words have to be moved to an even address, and you may try to move a word onto the stack after moving a byte onto it.

Address register indirect addressing with displacement.

```
            move.w      d0,4(a1)    *move the word in d0 to the memory location at a1+4
```

If a1 contains $7000 and d0 contains 23, this instruction moves 23 to memory address $7004.

```
            move.l      -2(a0),d5   *move the longword at a0-2 to d5
```

If a0 contains 2000, then this instruction moves the longword at address 1998 to d5.

The offset is a 16 bit word. The word is signed so the possible range offsets lie between -32767 and +32768.

```
      move.w      60000(a0),d5      *this is illegal as the displacement (60000) is too
                                great.
```

Address register indirect with index.

Example:

```
      move.w      4(a0,d1.l),d0
```

Move the data word from address a0+d1+4 into d0. This is the same as the previous mode, but with the addition that another register can be added in as well.
Note that the displacement here must fit in a byte. It must be greater than -127 and less than 128.

```
      move.l      20(a0,a1),d0
```

Move the long data from a0+a1+20 into d0

And this is possibly the most complicated form of 68000 instruction:

```
      move.l      10(a0,d0.l),30(a1.d1.l)
```

This instruction shows how powerful the 68000 is.
Move the longword data from the address a0+d0+10 to the address a1+d1+30.

Absolute short addressing.

This mode is used to address the lower 32K of memory, as only 16 bits are specified for the address - hence the short.

Just specify an address in the lower 32K - ie

```
      move.w      32,d0             *move the word at address 32 into d0
```

Absolute long addressing.

Whenever a memory address, above 32767 is used, this is the addressing mode.

```
      move.w      d0,$50000         *move a word from d0 to the address $50000.
```

Program counter with displacement.

This mode means that the displacement is added to the PC to get the actual address.
Using this mode to access memory allows access to data via a relative address, rather than an absolute address.
"So what?"
Well this is a good thing for two reasons:
1. You're program will not know where its going to be running in memory, or where the data will be living, so by using offsets, it doesn't matter.
2. Your program will be faster because a long address requires 32 bits to specify it, whereas an offset is just 16 bits.

```
      move.w      14(pc),d0         *move the word, 14 bytes from here, into d0
```

Note that the displacement must fit in a word i.e. ± 32767

Offsets would be a pain, if the assembler didn't calculate them for you :

```
      move.w      total(pc),d0      *move the word held in total to d0.
```

You needed worry about the offset here, as Fantasm will calculate it for you and substitute the offset for total.

This form of addressing uses two bytes for the offset, whereas an absolute address uses 4 bytes:

```
      move.w      fred(pc),d0       *this instruction is 4 bytes
      move.w      fred,d0           *this instruction is 6 bytes
```

Note that normally on a Mac

```
      move.w      fred,d0
```

would be illegal, as all data has to be accessed as offsets. However Fantasm uses a special loader in the application it creates to relocate any absolute long references.

Program counter with index.

This is the same as program counter with displacement, but with the added bonus of being able to add in a register (either address or data reg) as well.

```
      move.l      12(pc,d0.l),d1          * move the data at 12+here+d0.l into d1
```

Note that the displacement here must fit into a byte - i.e. ± 127

Immediate mode.

Immediate mode means a real number, not an address. It is indicated by preceding the number by a # symbol.

```
      move.w      #1,d0             *moves 1 into d0
      move.l      #123,a0           *moves 123 into a0
```

Compare these two instructions

```
1:    move.l      #100000,d5
2:    move.l      100000,d5
```

Instruction 1 moves the number 100000 into d5
Instruction 2 moves the data held in address 100000 into d5.
Be very careful not to forget your hashes!

Status register addressing.

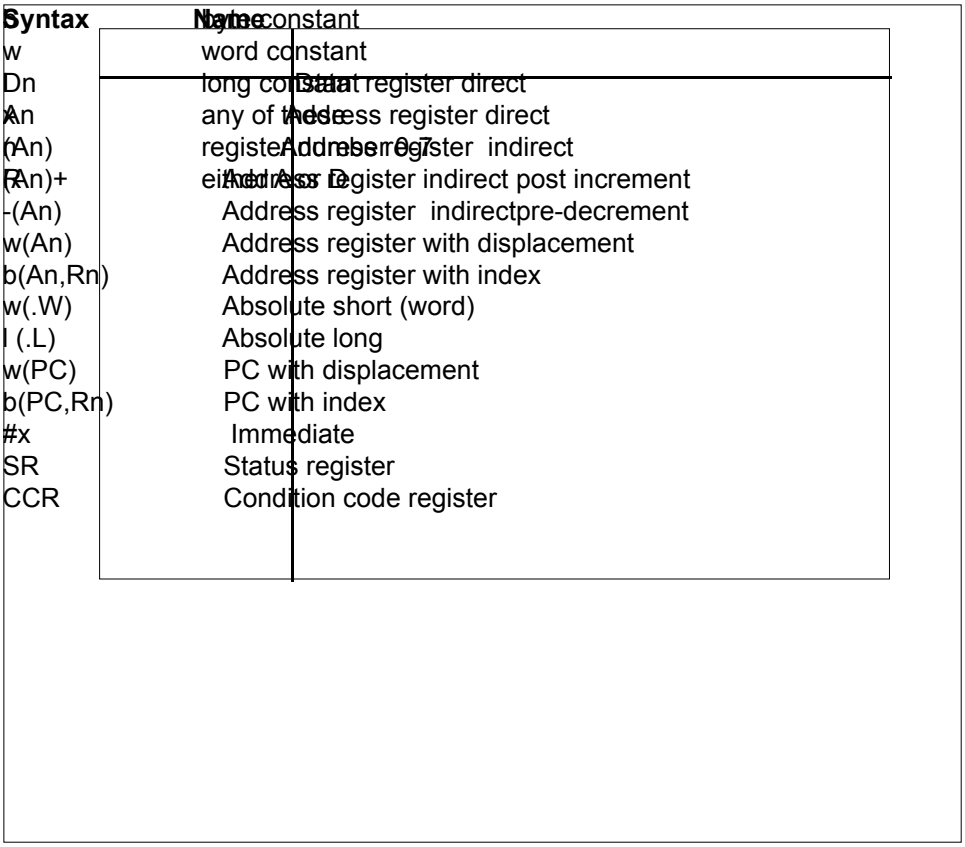
Either SR or CCR.

```
      move.w      #$2700,sr         *moves the number $2700 into the status register
```

*note that to affect the status register a word must be moved.
If you were in user mode, could you execute this instruction?
If not (no you couldn't) why not?

```
      move.b      #0,ccr            *clear the condition code register
```

*note that a move.w would be illegal here, as the CCR is only the lower 8 bits of the SR.



Syntax	Name
b	byte constant
w	word constant
Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect post increment
-(An)	Address register indirectpre-decrement
w(An)	Address register with displacement
b(An,Rn)	Address register with index
w(.W)	Absolute short (word)
l (.L)	Absolute long
w(PC)	PC with displacement
b(PC,Rn)	PC with index
#x	Immediate
SR	Status register
CCR	Condition code register

Addressing modes summary

Summary.

The important points that we have covered so far:

- The 68000 series has sixteen registers. 8 data registers and 8 address registers. Data registers maybe used for bytes, words or long words. Address registers may be used for word or long words only. If a word is moved into an address register, then it is sign extended to a long word.

- There are two special registers. The program counter (PC) and the status register (SR). The PC contains the address of the next instruction to be executed. The SR contains the state of the processor. The upper 8 bits of the SR are called the system byte, and cannot be accessed by programs running in “user mode”.
  The lower 8 bytes are called the condition code register (CCR). This byte contains bits that reflect the result of the last instruction.

- There are 14 distinct methods of specifying data in an instruction - these are called addressing modes.

- Stacks are organized in a last in first out structure. The register A7 is the hardware stack pointer. Software stacks can be implemented by using any other address register.