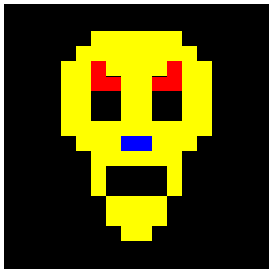
SECTION 1



1

DATA STORAGE

Before you can start writing programs in assembler you need to know a few things about how data is stored inside the computer and how that data can be accessed and changed. This chapter looks at the ways in which you can enter numbers from the keyboard and the notation which the computer uses to store these values in its memorv.

1.1 Hexadecimal notation

To most people it seems natural to use base ten when dealing with numbers. We have ten digits; 0,1, ... 8,9, and can use these to represent numbers as large as we please by making the value of a digit depend on which column it is in. Thus, when we consider the number 171 the first '1' represents 100, and the second '1' represents just one. Moving a digit one column to the left multiplies its value by ten; this is why our system is called base 10 or decimal.

When entering numbers into a computer you can still use base 10 if you wish, but another base -- base 16 -- is also available. For reasons which should become clear as you read through this chapter, base 16 (or hexadecimal) is far more suitable for working with computers. Hence it is advisable at this stage to spend some time becoming familiar with this number system.

In base 16 we need 16 different symbols to represent the 16 different ' hexadecimaldigits' .For convenience we retain the symbols 0 to 9, and use the letters A to F to represent the values of ten to fifteen.

Hexadecimal digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Another difference between base 10 and base 16 is what happens to a digit or hexadecimal digit when it is shifted one column to the left. Whereas we have seen that in base 10 this multiplies the value of the digit by ten, in base 16 it multiplies the value by sixteen. Hence 10 in hexadecimal represents the value sixteen.

Having two bases in which we can work can lead to confusion. Consider, for example, the number 10; as we have seen this can represent either of the values ten or sixteen depending on whether it is being interpreted as a decimal or hexadecimal number. We need a method of specifying whether a number is decimal or hexadecimal. Normally we do this by prefixing hexadecimal numbers with an ampersand (&), e.g.

&B1

The ' Bhas the value 16*11 because it is in the second column to the left, and the ' 1'represents 1 unit; the number therefore has the decimal value 176 + 1, i.e. 177.

&123

The ' 1'is in the third column to the left, so it has the value 16*16*1, the ' 2'has the value 16*2 and the ' 3' has the value 3. Adding these together produces 256 + 32 + 3, i.e. 291

There is no real need to learn how to convert between hexadecimal and decimal because the computer can do it for you, as shown below.

Converting hexadecimal to decimal
To print out the decimal value of a hexadecimal number, such as &123, type

```
PRINT &123
```

The answer, 291, is printed out.

Converting decimal to hexadecimal
To print, in hexadecimal, the value of a decimal number, type

```
PRINT ~&123
```

The answer, 7B, is printed out. The number printed will be in hexadecimal notation, but note that the computer doesn't use the symbol '&' when it is printing hexadecimal numbers. In this case it is obvious that the answer is a hexadecimal number but for an answer such as 79 you would need to know which base you requested the computer to use to be able to interpret the result correctly.

The symbol twiddle or, more accurately, tilde ~ means 'print in hexadecimal'; thus writing

```
PRINT~123
```

will print 123.

1.2 Binary notation and bits
Although the computer can accept numbers in either decimal or hexadecimal notation, it uses neither of these two systems for storing the numbers in its memory. The computer's memory consists of electronic circuits that can be put into one of two different states. The two states are normally represented as 0 and 1, but they are often referred to by different terms as listed below:

0	1
zero	one
low	high
clear	set
off	on
false	true

The circuits are said to be in a 'bistable state' i.e. they are always in one of two possible states. When the digits 0 and 1 are used to refer to these two states they are termed 'binary digits', or 'bits' for brevity.

With two bits, e.g. M and N, four different states can be represented:

M	N	
0	0	1
0	1	2
1	0	3
1	1	4

With a ' nibble' which is four bits, 16 different values can be represented (16 = 2^4). This means that a hexadecimal digit can be represented by a four-bit binary number. The hexadecimal digits and their binary equivalents are shown in the following table:

Decimal	Hexadecimal	Binary
0	0	0 0 0 0
1	1	0 0 0 1
2	2	0 0 1 0
3	3	0 0 1 1
4	4	0 1 0 0
5	5	0 1 0 1
6	6	0 1 1 0
7	7	0 1 1 1
8	8	1 0 0 0
9	9	1 0 0 1
10	A	1 0 1 0
11	B	1 0 1 1
12	C	1 1 0 0
13	D	1 1 0 1
14	E	1 1 1 0
15	F	1 1 1 1

Any hexadecimal number can be converted into its binary representation by the simple procedure of converting each hexadecimal digit into the corresponding four bits, for example

Thus the binary equivalent of &19 is 00011001, (or leaving out the leading zeros which are irrelevant, 11001).

1.3 Memory locations and bytes

The computer's memory is made up of a number of 'locations' each capable of holding a value. The size of each memory location is normally referred to as a 'byte'. Each byte can hold an eight-bit number, which means that it can store any one of 256 (2^8) different values; 0... 255.

We have seen already that each hexadecimal digit requires four bits to specify it. A byte, since it contains eight bits, can therefore represent any hexadecimal number between 0 and &FF.

The bits in a byte are usually numbered for convenience, as follows:

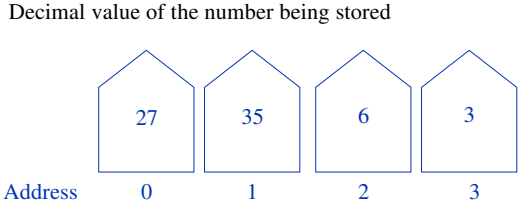
bit number	7	6	5	4	3	2	1	0
byte	0	0	0	1	1	0	0	1

Bit 0 is often referred to as the 'low-order bit' or 'least-significant bit', and bit 7 as the 'high-order bit' or most-significant bit'.

1.4 More about memory locations

Somehow it must be possible to distinguish between one location and another. Houses in a town are distinguished by each of them having a unique address. Even when the occupants of a house change, the address of the house remains the same. Similarly, each location in a computer has a unique 'address' consisting of a number which remains unchanged even when the contents of the memory location are altered. Thus we can speak of the 'contents of location 100' as being the number found in the location whose address is 100. The memory locations start from

address 0 and could look something like this:



An address can be one or two bytes long. This means that addresses can cover the range 0 to &FFFF. For a detailed look at which part of the memory each address corresponds to see the memory maps in Appendix A.

Examining memory locations

We can look at the contents of some memory locations in the computer using the query (?) operator. The reference ' ?X'means use the value of X as the address of the location under consideration. Hence the reference ' ?&FFEE'means that we are concerned with the location whose address is &FFEE. To look at this location type

```
PRINT ?&FFEE
```

This prints out the value found at the location specified, which in this case should be the number 108. Any memory location can be examined in this way and all of them will contain a number between 0 and 255.

It is often convenient to look at several memory locations in a row; for example, to list the contents of the 32 memory locations from &70 upwards, type

```
FOR N = 0 TO 31 : PRINT ?(N+&70); : NEXT N
```

An alternative way of writing this is

```
FOR N=0 TO 31 : PRINT N?&70; : NEXT N
```

This method is tidier than the other and gives identical results; i.e. for each of the values of N between 0 and 31, N

is added to the number &70 to give the address of the location whose contents are to be printed out. This should result in the contents of 32 memory locations being listed on the screen.

Changing memory locations
It is possible to change the number stored at a particular memory location by assigning a new value to it. As an example try changing the contents of &70. First, print the contents of this address; the value there will be whatever was in the memory when the computer was switched on since the computer does not use this location for storing any of the variables it is working with. To change the contents to 7, type

```
?&70=7
```

To verify the change, type

```
PRINT ?&70
```

Try setting the contents of this memory location to other numbers. Setting the contents to a number greater than 255 or &FF will result in the number entered modulo 256 being stored there, for example

```
?&70=600
PRINT ?&70
```

This will print out

```
88 (600 MOD 256)
```

A word of warning: Before you change the contents of any other memory locations be sure that you know what you are doing. Although it is quite safe to look at almost any memory location in the computer, care must be exercised when changing any of them. The example given here uses a specific location which is not used by the computer; if you change any other location you may lose any program you have in memory or confuse the computer to such an extent that it proves necessary to reset it by pressing BREAK to make it accept any further commands.

1.5 Negative numbers - two's complement
Although the values stored in the memory locations are between 0 and 255 these can be used to represent

both positive and negative numbers. To do this two's complement representation is used. To represent a number using this system we first have to consider what its positive counterpart is in binary notation. For example to find out how -5 would be stored consider the number

```
+5 = 00000101
```

We then find the complement of this, i.e. change each 0 into a 1 and each 1 into a 0, e.g.

```
complement of +5 = 1111010
```

Finally we add one:

```
  11111010
  +       1

  11111011
```

This gives us the two's complement representation of -5.

We can now try adding together +5 and -5 to see if they give us 0.

```
      00000101
      11111011

(1)   00000000
```

Ignoring the 1 which has overflowed gives us the result, zero, which we were expecting.

Note that when representing numbers using two's complement notation a single byte can represent any number between -128 and +127. The left-hand bit is 1 if the number is negative and 0 otherwise. Zero is classed as a non-negative number.

1.6 Storing text

If locations can only hold numbers between 0 and 255, how is text stored in the computer's memory? The answer is that numbers are used to represent the different characters. Hence text is stored simply as a sequence of numbers in successive memory locations. The computer does not become confused about

whether a number is representing an actual number or a character since the context will always make it clear how it should be interpreted.

The unique number corresponding to each character is given by its ASCII code (American Standard Code for Information Interchange). To find the ASCII code of a given character the ASC function can be used, for example type

```
PRINT ASC "A"
```

and the number 65 will be printed out. This means that the character ' A'is represented internally by the number 65. If you try repeating this process for B C D ..... you will notice that there is a certain regularity. The same is true for a b c d ... and the sequence 1 2 3 4 ....

A full table of the ASCII codes used to represent all the characters is given in Appendix A.