Data Representation

Computers are electronic machines which process data. This means that the data must be represented inside the computer by electrical signals. All modern computers use only two types of electrical signal, symbolised by the binary digits 0 and 1. They usually represent two different voltages, say 3v and 5v for the sake of argument. It is much easier to distinguish between only two different voltages, rather than a whole range of them.

Character Codes

These are binary codes that are used for the transmission of text (alphanumeric characters). Each letter, digit, punctuation symbol and control character on the keyboard has been allocated a binary code. For example, the code for:

A	1000001	4	0110100	&	0100110
a	1100001	2	0110010	+	0101011

When a key is pressed, the appropriate character code is sent along the cable to the processor. When the processor wishes to display a word on the screen, it sends the appropriate character codes to the monitor.

If you wish to transmit data from one computer to another, it is important to ensure that both use the same set of character codes. All modern computers use ASCII codes to represent characters. (American Standard Code for Information Interchange). A list of ASCII codes for the standard character set is shown on the last page.

When we type in the number 42, the ASCII code for 4 (0110100) is sent to the processor, followed by the ASCII code for 2 (0110010). Character codes are fine for the transmission of the data. But if we want to do some sort of calculations with the data, they have to be converted into a different sort of binary code, a binary number code.

Parity Check Bits

You might have noticed that the standard set of ASCII codes are 7 bits long, whereas computer memory tends to be 8 bits per memory location (or 16, or some other multiple of 8). The extra bit may be used for one of two things. It may be used as a **parity check bit**. This is a way of seeing if a character has been transmitted correctly. The circuitry at the 'sending' end, will set the parity bit to 0 or 1 so that the total number of 1's in the code is an even number. (This is if even parity is being used. If odd parity is being used, the total number of 1's must be odd). When the code has been transmitted, the circuitry at the 'receiving' end will count the number of ones, and, if there are not an even number, it knows that the code has become corrupted, and it can send a 'Please retransmit' signal back to the sender.

However, most modern hardware is so reliable that there is little chance of character codes becoming corrupted over the short distance between the keyboard and the processor. So the extra bit is instead used to represent more characters. The standard 7 bit ASCII code allows us to represent $2^7 = 126$ different characters. With 8 bits, that is doubled to 256 characters. Unfortunately, these are rarely standardised, and can vary from computer to computer and often between a computer and its printer.

Number Codes

Column Values

- Decimal Numbers

10^3 or 1000	10^2 or 100	10^1 or 10	10^0 or 1
8	**4**	**0**	**4**

Decimal numbers are in base 10. That means that the column values are powers of 10: 1, 10, 100, 1000 and so on, and all numbers can be represented by using the digits 0 to 9.
The number above is 8000 + 400 + 0 + 4, in other words, 8404.
We need a number system that only uses the digits 0 and 1, so that we can represent numbers using electrical signals.

- Binary Numbers

2^4 or 16	2^3 or 8	2^2 or 4	2^1 or 2	2^0 or 1
0	**1**	**1**	**0**	**1**

Binary numbers are in base 2. That means that the column values are powers of 2: 1, 2, 4, 8, 16 and so on, and all numbers can be represented by using the digits 0 and 1.
The number above is 0 + 8 + 4 + 0 + 1, in other words 13.

Negative Numbers

A lot of computers have memory locations which are 8 bits (1 byte) wide.

128	64	32	16	8	4	2	1
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**

This means that normally, they can store numbers in the range 0 to 255. This isn't very useful for several reasons. The first is that we will want to save numbers greater than 255. This is easily solved by using 2 or even 4 memory locations to store one number. The second is that we could do with some way of representing negative numbers. There are several ways of doing this:

- Sign and Magnitude
  In this method, the most significant bit (the one at the left hand end) is used to represent the sign of the number. 0 is usually used for positive, 1 for negative.

+ / -	64	32	16	8	4	2	1
1	**0**	**1**	**1**	**0**	**1**	**0**	**0**

The example above represents the number -52.

- Twos Complement
  The sign and magnitude method has problems, and the one that is usually used is called Twos Complement. The most significant bit is still used to represent the sign of the number, but it also has a numerical value.

-128	64	32	16	8	4	2	1
1	**0**	**1**	**1**	**0**	**1**	**0**	**0**

The example above represents the number -76. (-128+32+16+4).
This method is preferred because it is very easy to change a negative number into it's positive equivalent (and vice versa). All you have to do is invert the bits (ones become zeros and zeros become ones), and then add 1. Taking the representation of -76, shown above, as an example:

Invert bits: 01001011
Add one: 01001100

-128	64	32	16	8	4	2	1
0	**1**	**0**	**0**	**1**	**1**	**0**	**0**

is +76.

So subtraction can be done by negating one of the numbers and then using the addition circuits to add them together. This is cheaper than building a separate set of circuits for subtraction.

There is a third method of representing negative numbers which is called Ones Complement, but it is rarely used. It is the same as Twos complement, except that the most significant bit is worth one less (-127, in the example above.) It is rarely used because zero can be represented by two different binary codes, and the range of numbers it can represent is slightly less than Twos complement.

Binary Fractions

Fractions can be can be stored by extending the column headings to the right, past the binary point. If we look back to decimal numbers:

1000	100	10	1		1 ---- 10	1 ---- 100	1 ---- 1000	1 ---- 10000
2	**6**	**4**	**2**	**•**	**6**	**1**	**2**	**0**

we can see that we represent fractions by having column headings which have the values of 1, divided by the column headings on the left hand side of the decimal point. It is the same principle for binary fractions:

8	4	2	1		1 ---- 2	1 ---- 4	1 ---- 8	1 ---- 16
0	**1**	**0**	**0**	**•**	**1**	**1**	**0**	**0**

This is one way of representing the number 4¾ (four and three quarters). In actual fact, it is rare to find the whole part and the fractional part of the number stored in the same memory location. They would normally be stored as floating point numbers.

Negative fractions can be stored by adapting the Twos complement principle:

-1	1 ---- 2	1 ---- 4	1 ---- 8	1 ---- 16
1	**1**	**1**	**0**	**0**

This represents the number -1/4. Notice that the binary point is implied, but not actually stored. The software will have been written on the assumption that if this type of value is used, the binary point can always be assumed to lie between the most significant bit and the one on its immediate left.

Binary Coded Decimal

This has been left until now because it is a combination of both a number code and a character code. There are times when the computer needs to store a sequence of decimal digits. One example is the display of a pocket calculator, or a digital watch. No calculations are going to be done, so normally one would use ASCII character codes. The only problem with this is that each digit requires 7 bits, or 8 bits if you include the parity bit.

Binary Coded Decimal is used to save memory space, when it is known that only decimal digits are going to be displayed. Each of the ten digits are represented by a 4 bit character code, using the standard 8 4 2 1 column values.

Digit	BCD
0	**0000**
1	**0001**
2	**0010**
3	**0011**
4	**0100**
5	**0101**
6	**0110**
7	**0111**
8	**1000**
9	**1001**

This means that you can fit two decimal digits into one byte of memory space.

Hexadecimal Codes

Often, you will see binary codes expressed as hexadecimal numbers. These are numbers in base 16. The column headings for hexadecimal are powers of 16:

16^4 or 65536	16^3 or 4096	16^2 or 256	16^1 or 16	16^0 or 1
0	**0**	**1**	**7**	**9**

So the number 179 hex is equal to the decimal number (1 x 256) + (7 x 16) + (9 x 1) = 377.

There is a problem though. It is only when we get up to 16 (decimal) that we carry a one over into the second column of a hex number. (16 decimal = 10 hex). This means that all of the numbers up to 15 (decimal) must be represented by a single digit, in the units column of a hexadecimal number. Up to 9, we just use the normal decimal digits. But what about 10, 11, 12, 13, 14 and 15? What we do is use the letters A, B, C, D, E and F. So, for example:

16^4 or 65536	16^3 or 4096	16^2 or 256	16^1 or 16	16^0 or 1
0	**0**	**2**	**F**	**A**

The number 2FA (hex) is equal to (2 x 256) + (15 x 16) + (10 x 1) = 762 (decimal)

Why on earth do people prefer to write binary numbers using this complex system? One reason is that it allows us to write large numbers using relatively few digits. But decimal allows us to do that too.

Hexadecimal is preferred because, due to a mathematical trick, it is very easy to convert a binary code into a hexadecimal code, and vice versa. Simply divide an 8 bit binary code into two groups of 4 bits, and treat each group of 4 as if it had the column values 8 4 2 1. Then convert the 4 bit binary code into its hexadecimal equivalent. Thus an 8 bit binary code can be represented by 2 hexadecimal digits, rather than a varying number of decimal digits.