Week 1: Data representation

In modern computers information is represented and managed in a digital format.

That is, in one way or another the information - photos, pictures, video, music, data - is translated to 0s and 1s, and manipulated that way. As we said before, each individual piece of binary information is called a "binary digit", or *bit* . Individual bits may be combined to form a binary form, the most common of these is an 8-digit binary quantity called a *byte*. Also, often bytes are combined to form a computer *word*.

> Another term you might hear now and then is *nibble* - this is 4 binary digits together (i.e. half a byte). The relevance of this will become clearer when we start talking about hexadecimal numbers.

A byte has 256 different combinations, which can be used to store characters such in the ASCII code to represent letters, numbers, symbols, and basic control information (e.g. new line, tab, escape, etc.) Bytes and words are used for storing numbers, both integer (whole numbers) and floating-point (numbers with decimals). Since a byte may only store up to 256 different elements, any of these representations take up more than one byte. Thus, some encoding schemes for floating-point numbers and other information may exceed 64 bits (8 bytes) in size.

Examples:

- a character can be represented A = 01000001, B = 01000010, ...
- colours on a gray scale can be represented by white = 000000 ... black = 111111
- colours for Web pages are often described using 3 bytes RGB (primary light colours):
  - Byte 1: red
  - Byte 2: green
  - Byte 3: blue
- a yes/no or true/false can be represented by yes = 1, no = 0
- an open switch may be a 1, a closed switch may be a 0

Note that the meaning of the sequence of 0s and 1s is a matter for interpretation. For example, the symbol 01000001 may be interpreted as the character A or the number 65. This means that at all times we and the computer have to know what type of information we are dealing with.

As a consequence of the success of the modern computers to be able to store and manage binary data, these representations are more common every day. In this course we are interested in the architecture and functioning of general purpose digital computers. From the software/programmer point of view, all information inside a computer is 0s and 1s. From the engineering point of view these are two voltages, 0 volts is interpreted as a 0, +5 volts is interpreted as a 1.

> A 5 volt signal is typical of a very old electronic engineering standard known as **TTL logic** (http://en.wikipedia.org/wiki/Transistor-transistor_logic) .
>
> The signal lines of modern computer systems have signalling schemes barely over a volt, in some cases.

A computer spends much of its time performing arithmetic operations. Our arithmetic is based on the positional system where the position of a digit within a number determines its actual value; the number 2 in 123 means twenty, but the number 2 in 2,345 means two thousands. Although this seems only natural, it hasn't been always like this: for example, Roman numerals don't work that way, and as a consequence operations with Roman numerals are very different. The table shows some equivalences between Roman and our decimal system:

Roman Numeral	Equivalent
VII	7
II	2
IX	9

Roman numerals are additive, in the sense that each number represents the sum of its numerals, as in 8 = V + I + I + I. If we try to add Roman Numerals in the normal decimal way, we can see the difference between the two systems, and how our operations such as sums are based on the positional representation. We have learnt this from school, and it seems natural, but our decimal representation is only one possible system.

In the decimal system, the positional value of each column, starting from the right is 1, 10, 100, 1000, etc. That is $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, etc. The decimal number system has a base (or *radix*) of ten, using ten different digits or symbols in the system 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. All integers are made up of a combination of these digits, in the positional system described before.

Although we are used to the decimal system (i.e. to the base 10) number systems can be based on any radix such as 2, 5, 8, 10, 16, 28, 112, etc. The arithmetic rules in any system are similar. Although it may seem surprising at first, the addition and multiplication rules are quite the same, only the tables change. For example, if we work to the base 5, we count: 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, ...

So the addition rules look different to the decimal ones, as in:

$1_5 + 2_5 = 3_5$
$2_5 + 3_5 = 10_5$
$3_5 + 3_5 = 11_5$
$4_5 + 3_5 = 12_5$

And the times tables also change:

$2_5 * 1_5 = 2_5$
$2_5 * 2_5 = 4_5$
$2_5 * 3_5 = 11_5$
$2_5 * 4_5 = 13_5$

What would be the 3 times table, **in base 5** ?

Operation	Result
$3_5 * 1_5$	3_5
$3_5 * 2_5$	
$3_5 * 3_5$	
$3_5 * 4_5$	

We can also perform other operations, quite similar to the standard decimal ones. For example, 3^2 in base 5 would have a result of 14_5.

Since inside a computer everything is represented with 0s and 1s, the most important number system for digital computers is the base 2. All operations in a computer system are reduced to *binary operations* - that is, operations to the base 2. Unfortunately, although these operations are very easy and efficient for a computer system, they are very hard for us humans to do. For example, the number 1,258 is easy to remember, but its equivalent binary 10010000110 is quite hard.

To make things easier for us, two other systems are in use, octal (base 8) and hexadecimal (base 16). These are selected because they are closely related to binary, and it is very easy to move from binary to any of these two systems and back.

- **Binary:** base 2. Digits are: 0,1
- **Octal:** base 8. Digits are: 0, 1, 2, 3, 4, 5, 6, 7
- **Hexadecimal:** base 16. Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

We shall see more of hexadecimal numbers, and we'll learn to use them and do some basic arithmetic in each of them.

A digital computer represents all information as strings of bits – numbers: integer, floating point (i.e. 'real') numbers etc – characters – program instructions. How can we represent integer (whole) numbers using bits?

- Consider decimal 105

1		0		5
$1 * 10^2$	+	$0 * 10^1$	+	$5 * 10^0$

• In decimal each place indicates a power of 10

• It is a base-10 number system