

# Number Systems

## MA1S1

Tristan McLoughlin

November 27, 2013

[http://en.wikipedia.org/wiki/Binary\\_numeral\\_system](http://en.wikipedia.org/wiki/Binary_numeral_system)

<http://accu.org/index.php/articles/1558>

<http://www.binaryconvert.com>

<http://en.wikipedia.org/wiki/ASCII>

## Counting

Normally we use *decimal* or **base 10** when we count. That is we count by tens, hundreds = tens of tens, thousands = tens of hundreds, etc.



We see this in the SI units we are familiar with in science (kilometres =  $10^3$  metres, kilogrammes, centimetres =  $10^{-3}$  metres).

When we write the number 5678, we learned in the primary school that the 8 means 8 units, the 7 is 7 tens, while the remaining digits are 6 hundreds =  $6 \times 10^2$  and  $5 \times 10^3$ . So the number 5678 means

$$5 \times 10^3 + 6 \times 10^2 + 7 \times 10 + 8$$

## Counting in different bases

Although base 10 is the most common, we do see some traces of other bases in every day life.



For example, we normally buy eggs by dozens, and we can at least imagine shops buying eggs by the gross (meaning a dozen dozen or  $12^2 = 144$ ). So we use base 12 to some extent.

We can see some evidence of base 60 in angles and in time. In time units, 60 seconds is a minute and 60 minutes ( $= 60^2$  seconds) is an hour. Logically then we should have 60 hours in a day? Since we don't we stop using base 60.

# Binary

In *binary* or *base 2* we count by pairs. So we start with zero, then a single unit, but once we get to two units of any size we say that is a pair or a single group of 2.



So, when we count in base 2, we find:

- 1 is still 1
- 2 becomes a single group of 2 (a single pair)

Using positional notation as for decimal, we write this as 10. To make clear which base we are using, we may write a subscript 2 e.g.  $(10)_2$

- 3 is  $(11)_2 =$  one batch of 2 plus 1 unit.
- 4 is  $(100)_2 =$  one batch of  $2^2 + (0 \text{ batches of } 2) + (0 \text{ units})$

Using a more succinct format, we can explain how to count in binary as follows:

Decimal #	in binary	Formula for the binary format
1	$(1)_2$	$1$
2	$(10)_2$	$1 \times 2 + 0$
3	$(11)_2$	$1 \times 2 + 1$
4	$(100)_2$	$1 \times 2^2 + 0 \times 2 + 0$
5	$(101)_2$	$1 \times 2^2 + 0 \times 2 + 1$
6	$(110)_2$	$1 \times 2^2 + 1 \times 2 + 0$
7	$(111)_2$	$1 \times 2^2 + 1 \times 2 + 1$
8	$(1000)_2$	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 0$

So we can figure out what number we mean when we write something in binary by adding up the formula. Tedious, but the principle is not complicated.

At least for small numbers, there is a way to find the binary digits for a given number (*i.e.*, given in base 10) by repeatedly dividing by 2. For very small numbers, we can more or less do it by eye.

Say for the number **21**, we realise that it is more than  $16 = 2^4$  and not as big as  $32 = 2^5$ . In fact

$$21 = 16 + 5 = 16 + 4 + 1 = 2^4 + 2^2 + 1 = (10101)_2$$

We can in fact go from the other end...

Suppose we are starting with a positive integer number

$$n$$

(recall that an integer is a whole number, no fractional part). We want to know it in binary and in order to discuss what we are doing we will write down the units digit as  $n_0$ , the next digit from the right (multiples of 2) as  $n_1$ , etc.

That is we represent the number as

$$n = (n_k n_{k-1} \cdots n_2 n_1 n_0)_2 = n_k 2^k + n_{k-1} 2^{k-1} + \cdots + n_2 2^2 + n_1 2^1 + n_0$$

where the digits  $n_0, n_1, \dots, n_k$  are each either 0 or 1 and  $k$  is big enough so that  $2^k \leq n < 2^{k+1}$ .)

## From base 10 to base 2

We can give an algorithm to calculate the binary representation:

- If we divide  $n$  by 2 we get

$$\text{quotient} = \text{whole number part of } \frac{n}{2} = n_k 2^{k-1} + n_{k-1} 2^{k-2} + \cdots + n_2 2^1 + n_1$$

and remainder  $n_0$ . The remainder is 1 if  $n$  is odd and 0 if  $n$  is even.

- Now if we divide again by 2 we get remainder  $n_1$  and new quotient

$$\text{quotient} = n_k 2^{k-2} + n_{k-1} 2^{k-3} + \cdots + n_2$$

- If we divide again by 2 we get remainder  $n_2$  and new quotient

$$\text{quotient} = n_k 2^{k-3} + n_{k-1} 2^{k-4} + \cdots + n_3$$

and so on.

Thus at each step we can find the  $j$ -th digit of the binary representation as the remainder after dividing by 2.



## An Example

Look again at the case  $n = 21$  as an example.

- We have  $\frac{21}{2} = 10 +$  remainder 1. So the last binary digit is 1 = that remainder.
- Now  $10/2 = 5 +$  no remainder. That makes the digit in the 2's place 0.
- Continuing we have  $5/2 = 2 +$  remainder 1. So there third digit is 1.  
 $2/2 = 1 +$  no remainder and the fourth digit is 0 and finally  $1/2 = 0 +$  remainder 1 so the fifth digit is 1 i.e.  $(10101)_2$

*Each time* (even when the remainder is zero) we discover the binary digits one at a time from the units place up.

One thing to notice about binary is that we only ever need two digits, 0 and 1. We never need the digit 2 because that always gets 'carried' or moved to the next place to the left.

## Octal

In *octal* or *base 8* we count by 8's so we need 8 digits now: 0, 1, 2, 3, 4, 5, 6 and 7. Now zero is still 0 in octal, 1 is 1, 2 is 2, *etc.* 7 is still 7 in octal, but eight becomes  $(10)_8$ . In base 8  $(10)_8$  means  $1 \times 8 + 0$ .

Using a layout similar to the one used before we can explain how to count in octal as follows:

Decimal #	in octal	Formula for the octal format
1	$(1)_8$	1
2	$(2)_8$	2
7	$(7)_8$	7
8	$(10)_8$	$1 \times 8 = 0$
9	$(11)_8$	$1 \times 8 + 1$
10	$(12)_8$	$1 \times 8 + 2$
16	$(20)_8$	$2 \times 8 + 0$
17	$(21)_8$	$2 \times 8 + 1$

# Hexadecimal

Another base that is commonly used in computer systems is base 16, also called *hexadecimal*.

We can go ahead as we did before, just counting in groups and batches of 16. However, we run into a problem with the notation caused by the fact that the (decimal) number 10, 11, 12 13, 14 and 15 are normally written using two adjacent symbols. If we write 11 in hexadecimal, should we mean ordinary eleven or  $1 \times 16 + 1$ ?

To get around this difficulty we need new symbols for the numbers ten, eleven, ..., fifteen. What we do is use letters  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$  and  $f$  (or sometimes the capital letters A, B, C, D, E and F).

Thus the number ten becomes a single digit number  $(a)_{16}$  in hexadecimal. Eleven becomes  $(b)_{16}$ , and so on. But sixteen becomes  $(10)_{16}$ .

## Hexadecimal

Using a layout similar to the one used before we can explain how to count in hex as follows:

Decimal #	in hex	Formula for the hexadecimal format
1	$(1)_{16}$	1
9	$(9)_{16}$	9
10	$(a)_{16}$	10
15	$(f)_{16}$	15
16	$(10)_{16}$	$1 \times 16 = 0$
17	$(11)_{16}$	$1 \times 16 + 1$
26	$(1a)_{16}$	$1 \times 16 + 10$
32	$(20)_{16}$	$2 \times 16 + 0$
165	$(a5)_{16}$	$10 \times 16 + 5$
256	$(100)_{16}$	$1 \times 16^2 + 0 \times 16 + 0$

## Converting Octal or Hex to binary

We did already discussed how to convert between base 10 integers and base 2 between different bases using repeated division and keeping track of remainders. We can also use this to convert from decimal to octal, to hex, or to binary.

Alternatively if we write out the formula corresponding to a number in binary, octal or hex, we can compute the number in decimal by evaluating the formula.

These methods involve quite a bit of work, especially if the number is large. However there is a very simple way to convert between octal and binary. It is based on the fact that  $8 = 2^3$  is a power of 2 and so it is very easy to convert base 8 to base 2.

$$\begin{aligned}(541)_8 &= 5 \times 8^2 + 4 \times 8 + 1 \\&= (1 \times 2^2 + 0 \times 2 + 1) \times 2^6 + (1 \times 2^2) \times 2^3 + 1 \\&= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^6 + 1 \times 2^5 + 1 \\&= (101100001)_2\end{aligned}$$

## Converting Octal or Hex to binary

If we look at how this works, we see that we can convert from octal to binary by converting each octal digit to binary separately *but* we must write each digit as a 3 digit binary number.

Redoing the above example that way we have  $5 = (101)_2$  (uses 3 digits anyhow),  $4 = (100)_2$  (again uses 3 digits) and  $1 = (1)_2 = (001)_2$  (here we have to force ourselves to use up 3 digits) and we can say

$$(541)_8 = (101\ 100\ 001)_2 = (101100001)_2$$

This method works with any number of octal digits and we never have to really convert anything but the 8 digits 0-7 to binary. In reverse we can convert any binary number to octal very quickly if we just group the digits in 3's starting from the units. For example

$$(1111010100001011)_2 = (001\ 111\ 010\ 100\ 001\ 011)_2 = (172413)_8$$

## Converting Octal or Hex to binary

A similar method works for converting between binary and hex, except that now the rule is “4 binary digits for each hex digit”. It all works because  $16 = 2^4$ .

For example

$$(a539)_{16} = (1010\ 0101\ 0011\ 1001)_2 = (1010010100111001)_2$$

Or going in reverse

$$(1111010100001011)_2 = (1111\ 0101\ 0000\ 1011)_2 = (f50b)_{16}$$

We can use these ideas to convert octal to hex or *vice versa* by going via binary. We never actually have to convert any number bigger than 15.

If we wanted to convert a number such as 5071 to binary, it may be easier to find the octal representation (by repeatedly dividing by 8 and keeping track of all remainders) and then converting to binary at the end via the “3 binary digits for one octal” rule.

$$\begin{aligned}\frac{5071}{8} &= 633 + \text{remainder } 7 \\ \frac{633}{8} &= 79 + \text{remainder } 1 \\ \frac{79}{8} &= 9 + \text{remainder } 7 \\ \frac{9}{8} &= 1 + \text{remainder } 1 \\ \frac{1}{8} &= 0 + \text{remainder } 1 \\ (5071)_{10} &= (11717)_8 \\ &= (001\,001\,111\,001\,111)_2 \\ &= (001001111001111)_2 \\ &= (1001111001111)_2\end{aligned}$$



## Relation with computers



Although computers are very sophisticated the basic works are essentially many rows of on/off switches. Clearly a single on/off switch has only 2 possible settings of or states, but a row of 2 such switches has 4 possible states.

on	on
----	----

on	off
----	-----

off	on
-----	----

off	off
-----	-----

A row of 3 switches has twice as many possible setting because the third switch can be either on or off for each of the 4 possibilities for the first two. So  $2^3$  possibilities for 3 switches. In general  $2^8 = 256$  possibilities for 8 switches,  $2^n$  possible settings for a row of  $n$  switches.

Computers generally work with groups of 32 switches (also called 32 *bits*, where a ‘bit’ is the official name for the position that can be either on or off) and sometimes now with groups of 64. With 32 bits we have a total of  $2^{32}$  possible settings.

How big is  $2^{32}$ ?

We could work out with a calculator that it is  $4294967296 = 4.294967296 \times 10^9$  but there is a fairly simple trick for finding out approximately how large a power of 2 is. It is based on the fact that

$$2^{10} = 1024 \cong 10^3$$

Thus

$$2^{32} = 2^2 \times 2^{30} = 4 \times (2^{10})^3 \cong 4 \times (10^3)^3 = 4 \times 10^9$$

You can see that the answer is only approximate, but the method is fairly painless (if you are able to manipulate exponents).

## Integer format storage

Computers use binary to store everything, including numbers.

In general modern computers will use 32 bits to store each integer.

(Sometimes, they use 64 but we will concentrate on a 32 bit system.) How are the bits used? Take a simple example like 9. First write that in binary

$$9 = (1001)_2$$

and that only has 4 digits. For this number 9 we can pad it out by putting zeros in front

$$9 = (1001)_2 = (00 \dots 001001)_2$$

and then we end up filling our row of 32 bits like this:

9	0	0	...	0	0	1	0	0	1
Bit position:	1	2	...	27	28	29	30	31	32

One practical aspect of this system is that it places a limit on the maximum size of the integers we can store.

Since we allocate 32 bits we have a total of  $2^{32} \cong 4 \times 10^9$  different settings and so we have room for only that many different integers.

So we could fit in all the integers from 0 to  $2^{32} - 1$ , but that is usually not such a good strategy because we may also want to allow room for negative integers. If we don't especially favour positive integers over negative ones, that leaves us with space to store the integers from about  $-2^{31}$  to  $2^{31}$ . To be precise, that would be  $2 \times 2^{31} + 1 = 2^{32} + 1$  numbers if we include zero and so we would have to leave out either  $\pm 2^{31}$ .

Notice that  $2^{31} \cong 2 \times 10^9$  is not by any means a huge number. In a big company, there would be more Euros passing through the accounts than that in a year. In astronomy, the number of kilometres between stars would usually be bigger than that.

Computers are not actually limited to dealing with numbers less than  $2 \times 10^9$ , but they often are limited to dealing in this range for exact integer calculations. We will return to another method for dealing with numbers that have fractional parts and it allows for numbers with much larger magnitudes. However, this is done at the expense of less accuracy. When dealing with integers (that are within the range allowed) we can do exact calculations.

Returning to integers, we should explain about how to deal with negative integers. One way would be to allocate one bit to be a sign bit. So bit number 1 on could mean a minus sign. In this way we could store

$$-9 = -(1001)_2 = -(0 \dots 001001)_2$$

by just turning on the first bit. However, if you ask your calculator to tell you  $-9$  in binary, you will get a different answer. The reason is that computers generally do something more complicated with negative integers. This extra complication is not so important for us, but just briefly the idea is that the method used saves having to ever do subtraction.

So  $-1$  is actually stored as all ones:

-1	1	1	...	1	1	1	1	1	1
1	0	0	...	0	0	0	0	0	1
Bit position:	1	2	...	27	28	29	30	31	32

If you add 1 to that in binary, you will have to carry all the time.

Eventually you will get zeros in all 32 allowable places and you will have to carry the last 1 past the end. Since, only 32 places are allowed, this final carried 1 just disappears and we get 32 zeros, or 0.