COMP253 — Lecture 2: **Number Systems**

(For reference books, see: Number Representation Book List *on the COMP253 webpage.)*

Number Systems

Modern humans are familiar using the decimal number system, or base 10. The numeric symbols that form the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, commonly referred to as "digits."

Why 10 numbers?

If we could somehow get the computer to internally store values in base 10, a lot of programming hassle would be eliminated. But this is not going to happen in the near future (someday perhaps) because of the physics of how computers work.

Internally, computers use the binary number system, or base 2. The numeric symbols that form the binary number system are 0 and 1, commonly referred to as "bits" (binary digits).

Why 2 numbers?

As a note of interest, the ancient Mayan, Egyptian, and Incan civilisations also used base 10 number systems (but with different symbols than the common Arabic used today. In contrast, the ancient Babylonians used a number system that had 60 digits (sexagesimal), and the Greeks described a unique number system based on their alphabet. The Romans used the Greek numbering techniques, and devised a simplified (less symbols) method, known today as "roman numerals."

The Binary System

Even though binary only has two numbers (0 and 1), it can still represent every number that is possible in decimal.

Example,

$5_{10} = 101_2 \qquad 27_{10} = 11011_2$

But before continuing with the mathematical aspects, observe the differences between the numeric symbols for each system, and the length of each number. (Also note the subscript number that indicates the base.)

Decimal makes sense for us (or at least it should), but how does binary work?

Consider how numbers in decimal are sequenced,

```
0->9, 10->19, 20->29, ...
     ...,100->109, 110->119, 120->129, ...
```

And now in binary,

```
0,1, 10,11, 100,101, 110,111, ...
```

To understand how both number systems progress, look at how numbers are valued in decimal,

$$\begin{aligned} 342_{10} &= 3*10^2 + 4*10^1 + 2*10^0 \\ &= 300 + 40 + 2 \\ &= 342_{10} \end{aligned} \qquad \begin{aligned} 110_2 &= 1*2^2 + 1*2^1 + 0*2^0 \\ &= 4 + 2 + 0 \\ &= 6_{10} \end{aligned}$$

The value of a binary number is defined in the same manner as decimal with a base (of 2) raised to a specific power.

decimal: $\frac{\_\_}{10^7}\ \frac{\_\_}{10^6}\ \frac{\_\_}{10^5}\ \frac{\_\_}{10^4}\ \frac{\_\_}{10^3}\ \frac{\_\_}{10^2}\ \frac{\_\_}{10^1}\ \frac{\_\_}{10^0}$ *binary*: $\frac{\_\_}{2^7}\ \frac{\_\_}{2^6}\ \frac{\_\_}{2^5}\ \frac{\_\_}{2^4}\ \frac{\_\_}{2^3}\ \frac{\_\_}{2^2}\ \frac{\_\_}{2^1}\ \frac{\_\_}{2^0}$

Using the ideas just presented, try the following,

$1_2 = \underline{\ ?\ }_{10} \quad 111_2 = \underline{\ ?\ }_{10} \quad 1011_2 = \underline{\ ?\ }_{10} \qquad 8_{10} = \underline{\ ?\ }_2 \quad 6_{10} = \underline{\ ?\ }_2 \quad 33_{10} = \underline{\ ?\ }_2$

Conversion From Decimal to Binary

You have already seen how to convert from binary to decimal in the previous lecture. Now the focus is placed on converting from decimal to binary.

Method 1: *Subtraction*

- subtract the highest possible bit-position value from the current decimal value.

ex:

$89_{10} = ?_2$

$$\begin{array}{lcccccccc} & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ = & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \end{array}$$

msb lsb

=> $\underline{89_{10} = 01011001_2}$

```
128 > 89       -> 7-bit = 0   msb
 89 – 64 = 25  -> 6-bit = 1
 32 > 25       -> 5-bit = 0
 25 – 16 = 9   -> 4-bit = 1
9 – 8 = 1      -> 3-bit = 1
4 > 1          -> 2-bit = 0
 2 > 1         -> 1-bit = 0
1 – 1 = 0      -> 0-bit = 1   lsb
      (stop)
```

This technique is not very efficient since the value of the 2^x bit-position value must be remembered, yet it makes more sense than the division method for most people.

Method 2: *Division*

ex: $89_{10} = ?_2$

```
89 / 2 = 44.5 (44 + remainder 1) : 1   lsb
44 / 2 = 22   (22 + remainder 0) : 0
22 / 2 = 11 + remainder 0        : 0
11 / 2 = 5 + remainder 1         : 1
 5 / 2 = 2 + remainder 1         : 1
 2 / 2 = 1 + remainder 0         : 0
 1 / 2 = 0 + remainder 1         : 1   msb
       (stop)
```

msb lsb

=> $\underline{89_{10} = 01011001_2}$

The division technique is much faster than subtraction and there is no need to memorise the 2^x bit-position values.

Hexadecimal and Octal (see Appendix D in the textbook)

As much as binary describes the fundamental number system for the computer, it is cumbersome for humans to work with. Hardware and software designers needed something with the *quality of binary*, but was more familiar and less bulky (they wanted to design and code in binary, but without using 0's and 1's).

Hence, the introduction of the hexadecimal (base 16) and octal (base 8) numbers systems. In the following discussion, consider hex and octal as compressed binary.

Hexadecimal (Hex)

Base 16 (2^4). Numeric symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C(12), D(13), E(14), F(15)

ex:

$27_{10} = 1b_{16}$ $5_{10} = 5_{16}$ $244_{10} = F4_{16}$

Numeric values:

16^3 16^2 16^1 16^0
= 4096 256 16 1

ex:

$$F4_{16} = F(15)*16^1 + 4*16^0 = 15*16 + 4*1 = 240 + 4 = 244_{10}$$

Conversion (decimal to hexadecimal)

(**Note**: The Subtraction Method can be used, but it is quite difficult for hexadecimal. The Division Method is far more practical.)

ex: $89_{10} = ?_{16}$

```
89 / 16 = 5 9/16 (5 + remainder 9) : 9   ls
 5 / 16 = 0 + remainder 5          : 5   ms
       (stop)
```

ms ls
=> $89_{10} = 59_{16}$

ex:

$75_{10} = ?_{16}$

```
75 / 16 = 4 11/16 (4 + remainder 11) : B(11)   ls
 4 / 16 = 0 + remainder 4            : 4       ms
       (stop)
```

ms ls
=> $75_{10} = 4B_{16}$

Octal

Base 8 (2^4). Numeric symbols: 0, 1, 2, 3, 4, 5, 6, 7

ex:

$27_{10} = 33_8$ $5_{10} = 5_8$ $244_{10} = 364_8$

Numeric values:

8^3 8^2 8^1 8^0
= 512 64 8 1

ex:

$$364_8 = 3*8^2 + 6*8^1 + 4*8^0 = 3*64 + 6*8 + 4*1 = 192 + 48 + 4 = 244_{10}$$

Conversion (decimal to octal)

(**Note**: As with hexadecimal, the Subtraction Method can be used, but it is quite difficult.)

ex:

$89_{10} = ?_{8}$

```
89 / 8 = 11 1/8 (11 + remainder 1) : 1   ls
11 / 8 =  1 + remainder 3          : 3
 1 / 8 =  0 + remainder 1          : 1   ms
       (stop)
```

```
      ms   ls
```
=> $\underline{89_{10} = 131_{8}}$

ex:

$75_{10} = ?_{8}$

```
75 / 8 = 9 + remainder 3      : 3   ls
 9 / 8 = 1 + remainder 1      : 1
 1 / 8 = 0 + remainder 1      : 1   ms
       (stop)
```

```
      ms   ls
```
=> $\underline{75_{10} = 113_{8}}$

Why use Hexadecimal and Octal?

(Consult the Numeric Conversion Chart, available on COMP253 webpage)

These number systems are convenient (easy to convert from Binary) when programming while being far less bulky than binary. Essentially, hexadecimal and octal can represent binary numbers without having to deal with the large number of 0's and 1's .

Consider the following,

$89_{10} = 1011001_{2} = 59_{16} = 131_{8}$

and look at the respective Binary Tables,

```
Hex:     __5__   __9__   16
Bin: 0 0 1 0 1 1 0 0 1   2
Oct:   1     3     1     8
```

A binary number can have its bits grouped so that the hexadecimal or octal number can be read directly.

This also means that once a decimal number is converted to hexadecimal, octal, or binary, it can be quickly converted to the rest.

ex:

$$75_{10} = 4B_{16} \quad \left| \begin{array}{l} 4_{16} = 0100_{2} \\ \\ B_{16} = 1011_{2} \end{array} \right| \quad 75_{10} = 0100\ 1011_{2}$$

$$75_{10} = 113_{8} \quad \left| \begin{array}{l} 1_{8} = 001_{2} \\ 1_{8} = 001_{2} \\ 3_{8} = 011_{2} \end{array} \right| \quad 75_{10} = 001\ 001\ 011_{2}$$

Beyond just quick conversions, hexadecimal and octal make life easy for lower-level language programmers (assembly and machine language) because the bits in memory make binary numbers, which are represented hexadecimal or octal numbers with no effort on the programmer's part.

(The above comment also applies to network programmers that must deal with flowing groups of bits.)