Numbering System

Many number systems are in use in digital technology. The most common are the decimal, binary, octal, and hexadecimal systems. The decimal system is clearly the most familiar to us because it is a tool that we use every day. Examining some of its characteristics will help us to better understand the other systems. In the next few pages we shall introduce four numerical representation systems that are used in the digital system. There are other systems, which we will look at briefly.

- Decimal
- Binary
- Octal
- Hexadecimal

Decimal System

The decimal system is composed of 10 numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Using these symbols as digits of a number, we can express any quantity. The decimal system is also called the base-10 system because it has 10 digits.

10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
=1000	=100	=10	=1	.	=0.1	=0.01	=0.001
Most Significant Digit				Decimal point			Least Significant Digit

Even though the decimal system has only 10 symbols, any number of any magnitude can be expressed by using our system of positional weighting.

Decimal Examples

- 3.14_{10}
- 52_{10}
- 1024_{10}
- 64000_{10}

Binary System

In the binary system, there are only two symbols or possible digit values, 0 and 1. This base-2 system can be used to represent any quantity that can be represented in decimal or other base system.

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
=8	=4	=2	=1	.	=0.5	=0.25	=0.125
Most Significant Digit				Binary point			Least Significant Digit

Binary Counting

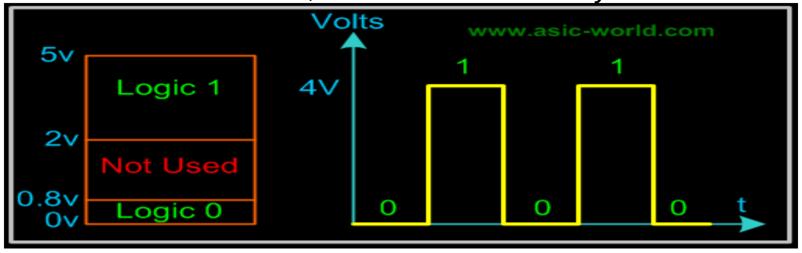
The Binary counting sequence is shown in the table:

2^3	2^2	2^1	2^0	Decimal
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8

1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Representing Binary Quantities
In digital systems the information that is being processed is usually presented in binary form. Binary quantities can be represented by any device that has only two operating states or possible conditions. E.g.. a switch is only open or closed. We arbitrarily (as we define them) let an open switch represent binary 0 and a closed switch represent binary 1. Thus we can represent any binary number by using series of switches.

Typical Voltage Assignment
Binary 1: Any voltage between 2V to 5V
Binary 0: Any voltage between 0V to 0.8V
Not used: Voltage between 0.8V to 2V in 5 Volt CMOS and TTL Logic, this may cause error in a digital circuit. Today's digital circuits works at 1.8 volts, so this statement may not hold true for all logic circuits.



We can see another significant difference between digital and analog systems. In digital systems, the exact voltage value is not important; eg, a voltage of 3.6V means the same as a voltage of 4.3V. In analog systems, the exact voltage value is important.
The binary number system is the most important one in digital systems, but several others are also important. The decimal system is important because it is universally used to represent quantities outside a digital system. This means that there will be situations where decimal values have to be converted to binary values before they are entered into the digital system.
In additional to binary and decimal, two other number systems find wide-spread applications in digital systems. The octal (base-8) and hexadecimal (base-16) number systems are both used for the same purpose- to provide an efficient means for representing large binary system.

Octal System
The octal number system has a base of eight, meaning that it has eight possible digits: 0,1,2,3,4,5,6,7.

8^3	**8^2**	**8^1**	**8^0**		**8^{-1}**	**8^{-2}**	**8^{-3}**
=512	=64	=8	=1	.	=1/8	=1/64	=1/512
Most Significant Digit				Octal point			Least Significant Digit

Octal to Decimal Conversion

- $237_8 = 2 \times (8^2) + 3 \times (8^1) + 7 \times (8^0) = 159_{10}$
- $24.6_8 = 2 \times (8^1) + 4 \times (8^0) + 6 \times (8^{-1}) = 20.75_{10}$
- $11.1_8 = 1 \times (8^1) + 1 \times (8^0) + 1 \times (8^{-1}) = 9.125_{10}$
- $12.3_8 = 1 \times (8^1) + 2 \times (8^0) + 3 \times (8^{-1}) = 10.375_{10}$

Hexadecimal System

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols.

16^3	16^2	16^1	16^0		16^{-1}	16^{-2}	16^{-3}
=4096	=256	=16	=1	.	=1/16	=1/256	=1/4096
Most Significant Digit				Hexa Decimal point			Least Significant Digit

Hexadecimal to Decimal Conversion

- $24.6_{16} = 2 \times (16^1) + 4 \times (16^0) + 6 \times (16^{-1}) = 36.375_{10}$
- $11.1_{16} = 1 \times (16^1) + 1 \times (16^0) + 1 \times (16^{-1}) = 17.0625_{10}$
- $12.3_{16} = 1 \times (16^1) + 2 \times (16^0) + 3 \times (16^{-1}) = 18.1875_{10}$

Binary Codes

Binary codes are codes which are represented in binary system with modification from the original ones. Below we will be seeing the following:

- Weighted Binary Systems
- Non Weighted Codes

Weighted Binary Systems

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	8421	2421	5211	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0010	0010	0011	0101
3	0011	0011	0101	0110
4	0100	0100	0111	0111
5	0101	1011	1000	1000
6	0110	1100	1010	1001
7	0111	1101	1100	1010
8	1000	1110	1110	1011
9	1001	1111	1111	1100

8421 Code/BCD Code

The BCD (Binary Coded Decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are 8,4,2,1.

Example: The bit assignment 1001, can be seen by its weights to represent the decimal 9 because:

$1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 9$

2421 Code

This is a weighted code, its weights are 2, 4, 2 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 2 + 4 + 2 + 1 = 9. Hence the 2421 code represents the decimal numbers from 0 to 9.

5211 Code

This is a weighted code, its weights are 5, 2, 1 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 5 + 2 + 1 + 1 = 9. Hence the 5211 code represents the decimal numbers from 0 to 9.

Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

Non Weighted Codes

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value.

Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Example: 1000 of 8421 = 1011 in Excess-3

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a special place.

Decimal Number	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Binary to Gray Conversion

- Gray Code MSB is binary code MSB.

- Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.
- MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

Error Detecting and Correction Codes
For reliable transmission and storage of digital data, error detection and correction is required. Below are a few examples of codes which permit error detection and error correction after detection.
Error Detecting Codes
When data is transmitted from one point to another, like in wireless transmission, or it is just stored, like in hard disks and memories, there are chances that data may get corrupted. To detect these data errors, we use special codes, which are error detection codes.

Parity
In parity codes, every data byte, or nibble (according to how user wants to use it) is checked if they have even number of ones or even number of zeros. Based on this information an additional bit is appended to the original data. Thus if we consider 8-bit data, adding the parity bit will make it 9 bit long.
At the receiver side, once again parity is calculated and matched with the received parity (bit 9), and if they match, data is ok, otherwise data is corrupt.
There are two types of parity:

- **Even parity:** Checks if there is an even number of ones; if so, parity bit is zero. When the number of ones is odd then parity bit is set to 1.
- **Odd Parity:** Checks if there is an odd number of ones; if so, parity bit is zero. When number of ones is even then parity bit is set to 1.

Check Sums
The parity method is calculated over byte, word or double word. But when errors need to be checked over 128 bytes or more (basically blocks of data), then calculating parity is not the right way. So we have checksum, which allows to check for errors on block of data. There are many variations of checksum.

- Adding all bytes
- CRC
- Fletcher's checksum
- Adler-32

The simplest form of checksum, which simply adds up the asserted bits in the data, cannot detect a number of types of errors. In particular, such a checksum is not changed by:

- Reordering of the bytes in the message
- Inserting or deleting zero-valued bytes
- Multiple errors which sum to zero

Example of Checksum : Given 4 bytes of data (can be done with any number of bytes): 25h, 62h, 3Fh, 52h

- Adding all bytes together gives 118h.
- Drop the Carry Nibble to give you 18h.
- Get the two's complement of the 18h to get E8h. This is the checksum byte.

To Test the Checksum byte simply add it to the original group of bytes. This should give you 200h.
Drop the carry nibble again giving 00h. Since it is 00h this means the checksum means the bytes were probably not changed.

Error-Correcting Codes
Error-correcting codes not only detect errors, but also correct them. This is used normally in Satellite communication, where turn-around delay is very high as is the probability of data getting corrupt.

ECC (Error correcting codes) are used also in memories, networking, Hard disk, CDROM, DVD etc. Normally in networking chips (ASIC), we have 2 Error detection bits and 1 Error correction bit.

Hamming Code

Hamming code adds a minimum number of bits to the data transmitted in a noisy channel, to be able to correct every possible one-bit error. It can detect (not correct) two-bits errors and cannot distinguish between 1-bit and 2-bits inconsistencies. It can't - in general - detect 3(or more)-bits errors.

The idea is that the failed bit position in an n-bit string (which we'll call X) can be represented in binary with $\log_2(n)$ bits, hence we'll try to get it adding just $\log_2(n)$ bits.

First, we set $m = n + \log_2(n)$ to the encoded string length and we number each bit position starting from 1 through m. Then we place these additional bits at power-of-two positions, that is 1, 2, 4, 8..., while remaining ones (3, 5, 6, 7...) hold the bit string in the original order.

Now we set each added bit to the parity of a group of bits. We group bits this way: we form a group for every parity bit, where the following relation holds:

position(bit) AND position(parity) = position(parity)

(Note that: AND is the bit-wise boolean AND; parity bits are included in the groups; each bit can belong to one or more groups.)

So bit 1 groups bits 1, 3, 5, 7... while bit 2 groups bits 2, 3, 6, 7, 10... , bit 4 groups bits 4, 5, 6, 7, 12, 13... and so on.

Thus, by definition, X (the failed bit position defined above) is the sum of the incorrect parity bits positions (0 for no errors).

To understand why it is so, let's call X_n the nth bit of X in binary representation. Now consider that each parity bit is tied to a bit of X: parity1 -> X_1, parity2 -> X_2, parity4 -> X_3, parity8 -> X_4 and so on - for programmers: they are the respective AND masks -. By construction, the failed bit makes fail only the parity bits which correspond to the 1s in X, so each bit of X is 1 if the corresponding parity is wrong and 0 if it is correct.

Note that the longer the string, the higher the throughput n/m and the lower the probability that no more than one bit fails. So the string to be sent should be broken into blocks whose length depends on the transmission channel quality (the cleaner the channel, the bigger the block). Also, unless it's guaranteed that at most one bit per block fails, a checksum or some other form of data integrity check should be added.

Alphanumeric Codes

The binary codes that can be used to represent all the letters of the alphabet, numbers and mathematical symbols, punctuation marks, are known as alphanumeric codes or character codes. These codes enable us to interface the input-output devices like the keyboard, printers, video displays with the computer.

ASCII Code

ASCII stands for American Standard Code for Information Interchange. It has become a world standard alphanumeric code for microcomputers and computers. It is a 7-bit code representing $2^7 = 128$ different characters. These characters represent 26 upper case letters (A to Z), 26 lowercase letters (a to z), 10 numbers (0 to 9), 33 special characters and symbols and 33 control characters.

The 7-bit code is divided into two portions, The leftmost 3 bits portion is called zone bits and the 4-bit portion on the right is called numeric bits.

An 8-bit version of ASCII code is known as USACC-II 8 or ASCII-8. The 8-bit version can represent a maximum of 256 characters.

EBCDIC Code

EBCDIC stands for Extended Binary Coded Decimal Interchange. It is mainly used with large computer systems like mainframes. EBCDIC is an 8-bit code and thus accomodates up to 256 characters. An EBCDIC code is divided into two portions: 4 zone bits (on the left) and 4 numeric bits (on the right).

Floating Point Numbers

A real number or floating point number is a number which has both an integer and a fractional part. Examples for real real decimal numbers are 123.45, 0.1234, -0.12345, etc. Examples for real binary numbers are 1100.1100, 0.1001, -1.001, etc. In general, floating point numbers are expressed in exponential notation.
For example the decimal number

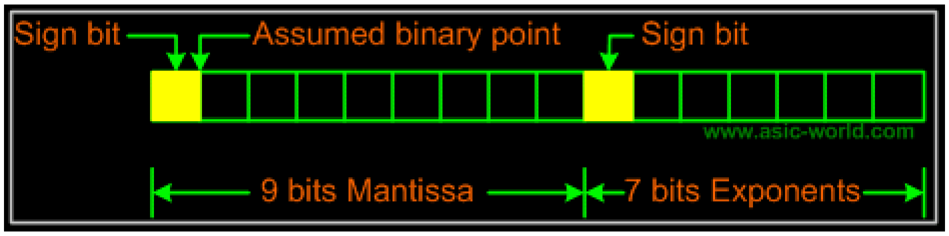
- 30000.0 can be written as 3×10^4.
- 312.45 can be written as 3.1245×10^2.

Similarly, the binary number 1010.001 can be written as 1.010001×10^3.
The general form of a number N can be expressed as

$$N = \pm m \times b^{\pm e}.$$

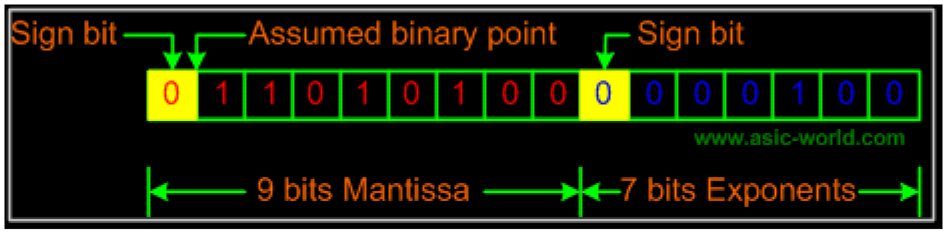
Where m is mantissa, b is the base of number system and e is the exponent. A floating point number is represented by two parts. The number first part, called mantissa, is a signed fixed point number and the second part, called exponent, specifies the decimal or binary position.

Binary Representation of Floating Point Numbers
A floating point binary number is also represented as in the case of decimal numbers. It means that mantissa and exponent are expressed using signed magnitude notation in which one bit is reserved for sign bit.
Consider a 16-bit word used to store the floating point numbers; assume that 9 bits are reserved for mantissa and 7 bits for exponent and also assume that the mantissa part is represented in fraction system. This implies the assumed binary point is at the mantissa sign bit immediate right.



Example
A binary number 1101.01 is represented as
Mantissa = 110101 = $(1101.01)_2 = 0.110101 \times 2^4$
Exponent = $(4)_{10}$
Expanding mantissa to 8 bits we get 11010100
Binary representation of exponent $(4)_{10}$ = 000100

The required representation is


Sign bit
Assumed binary point
Sign bit
0 1 1 0 1 0 1 0 0 0 0 0 0 1 0 0
www.asic-world.com
9 bits Mantissa
7 bits Exponents