# SIGNED AND UNSIGNED NUMBER

# UNSIGNED NUMBER

Unsigned numbers contain only magnitude of the number. They don’t have any sign. That means all unsigned binary numbers are positive. As in decimal number system, the placing of positive sign in front of the number is optional for representing positive numbers. Therefore, all positive numbers including zero can be treated as unsigned numbers if positive sign is not assigned in front of the number.

# REPRESENTATION OF UNSIGNED BINARY NUMBERS

The bits present in the un-signed binary number holds the magnitude of a number. That means, if the un-signed binary number contains ‘N’ bits, then all N bits represent the magnitude of the number, since it doesn’t have any sign bit.

**Example:**

Consider the decimal number 108. The binary equivalent of this number is 1101100. This is the representation of unsigned binary number.

$$108_{10} = 1101100_2$$

# SIGNED NUMBER

Signed numbers contain both sign and magnitude of the number. Generally, the sign is placed in front of number. So, we have to consider the positive sign for positive numbers and negative sign for negative numbers. Therefore, all numbers can be treated as signed numbers if the corresponding sign is assigned in front of the number.

If sign bit is zero, which indicates the binary number is positive. Similarly, if sign bit is one, which indicates the binary number is negative.

# REPRESENTATION OF SIGNED BINARY NUMBER

The Most Significant Bit (*MSB)* of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called as sign bit. The positive sign is represented by placing ‘0’ in the sign bit. Similarly, the negative sign is represented by placing ‘1’ in the sign bit.

If the signed binary number contains ‘N’ bits, then $N-1$ bits only represent the magnitude of the number since one bit MSB is reserved for representing sign of the number.There are three types of representations for signed binary numbers

- Sign-Magnitude form
- 1’s complement form
- 2’s complement form

Representation of a positive number in all these 3 forms is same. But, only the representation of negative number will differ in each form.

Example:

Consider the positive decimal number +108. The binary equivalent of magnitude of this number is **1101100**. These 7 bits represent the magnitude of the number 108. Since it is positive number, consider the sign bit as zero, which is placed on left most side of magnitude.

$$+108_{10} = 01101100_2$$

Therefore, the signed binary representation of positive decimal number +108 is **01101100**. So, the same representation is valid in sign-magnitude form, 1's complement form and 2's complement form for positive decimal number +108.

## Sign-Magnitude form

In sign-magnitude form, the MSB is used for representing sign of the number and the remaining bits represent the magnitude of the number. So, just include sign bit at the left most side of unsigned binary number. This representation is similar to the signed decimal numbers representation.

Example:

Consider the negative decimal number -108. The magnitude of this number is 108. We know the unsigned binary representation of 108 is 1101100. It is having 7 bits. All these bits represent the magnitude.

Since the given number is negative, consider the sign bit as one, which is placed on left most side of magnitude.

$$\mathbf{-108_{10} = 11101100_2}$$

Therefore, the sign-magnitude representation of -108 is 11101100.

## 1's complement form

The 1's complement of a number is obtained by complementing all the bits of signed binary number. So, 1's complement of positive number gives a negative number. Similarly, 1's complement of negative number gives a positive number.

That means, if you perform two times 1's complement of a binary number including sign bit, then you will get the original signed binary number.

Example:

Consider the negative decimal number -108. The magnitude of this number is 108. We know the signed binary representation of 108 is 01101100.

It is having 8 bits. The MSB of this number is zero, which indicates positive number. Complement of zero is one and vice-versa. So, replace zeros by ones and ones by zeros in order to get the negative number.

$$-108_{10} = 10010011_2$$

Therefore, the 1's complement of $108_{10}$ is $10010011_2$.

## 2’s complement form

The 2’s complement of a binary number is obtained by adding one to the 1’s complement of signed binary number. So, 2’s complement of positive number gives a negative number. Similarly, 2’s complement of negative number gives a positive number.

That means, if you perform two times 2’s complement of a binary number including sign bit, then you will get the original signed binary number.

Example:

Consider the negative decimal number -108.

We know the 1’s complement of $(108)_{10}$ is $(10010011)_2$

$$\textit{2's complement of } 108_{10} = \textit{1's complement of } 108_{10} + 1 = 10010011 + 1$$

$$= 10010100$$

Therefore, the 2’s complement of $108_{10}$ is $10010100_2$.

# CHARACTER REPRESENTATION

# 04. CHARACTER REPRESENTATION

So far we have discussed how to represent numbers (0 and positive integers, negative and real numbers), but as well as numbers, computers often need to represent characters and textual information, such as text displayed on a screen or printer, word processor files, HTML, programming language source code, and much more.

- Data refers to the symbols that represent people, events, things, and ideas. Data can be a name, a number, the colors in a photograph, or the notes in a musical composition.
- As a result, all characters, whether they are letters, punctuation or digits, are stored as binary numbers. All of the characters that a computer can use are called a character set.
- Character representation is how numbers and letters are represented by a specific group of symbols. This process is called encoding.

# DECIMAL ENCODING

## BCD

In **Binary-Coded Decimal** (BCD), 4 bits are used to encode each decimal digit. The names 8-4-2-1, 4-2-2-1 and 7-4-2-1 come from the denary value represented by a binary 1 in the corresponding column.

| Decimal Digit | 8421 BCD | 4221 BCD | 7421 BCD |
|---|---|---|---|
| 0 | 0000 | 0000 | 0000 |
| 1 | 0001 | 0001 | 0001 |
| 2 | 0010 | 0100 or 0010 | 0010 |
| 3 | 0011 | 0101 or 0011 | 0011 |
| 4 | 0100 | 1000 or 0110 | 0100 |
| 5 | 0101 | 1001 or 0111 | 0101 |
| 6 | 0110 | 1100 or 1010 | 0110 |
| 7 | 0111 | 1101 or 1011 | 1000 or 0111 |
| 8 | 1000 | 1110 | 1001 |
| 9 | 1001 | 1111 | 1010 |

# CONVERSION OF DECIMAL NUMBER TO BCD

(i) $(17)_{10}$ = **0001 0111 BCD**

0001 0111

**$10001_2$**

(ii) $(156)_{10}$ = **0001 0101 0110 BCD**

# CONVERSION OF BCD TO DECIMAL

10100 BCD =

**0001 0100**

$\mathbf{14_{10}}$

# CHARACTER ENCODING

## ASCII

- Historically, and possibly still today, the most common standard for character encoding is ASCII (pronounced "**as key**") which stands for the **American Standard Code for Information Interchange**, which was originally created in 1963.
- ASCII was originally developed in the United States, it was subsequently adapted for use in many other countries. The original and primary variant is sometimes referred to as US-ASCII as it includes the characters needed US English and US typographical symbols.

# ASCII cont.

- ASCII is a 7-bit code, which means that 7 binary digits are used to represent each character. This allows for 128 different characters ($2^7$), from 0000000 binary (0 decimal, 0 hexadecimal) to 1111111 binary (127 decimal 7F hexadecimal).
- 95 of the 128 characters in ASCII are printable characters. In US-ASCII, this includes the letters A to Z in both upper and lower case, the digits 0 to 9, and a variety of punctuation and typographic symbols. As already mentioned, non-US variants of ASCII may substitute some country-specific characters, generally for some of the US punctuation characters or typographic symbols.
- 33 of the 128 characters are non-printable control codes that were originally intended for a variety of functions

# Decimal - Binary - Octal - Hex – ASCII Conversion Chart

| Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00000000 | 000 | 00 | NUL | 32 | 00100000 | 040 | 20 | SP | 64 | 01000000 | 100 | 40 | @ | 96 | 01100000 | 140 | 60 | ` |
| 1 | 00000001 | 001 | 01 | SOH | 33 | 00100001 | 041 | 21 | ! | 65 | 01000001 | 101 | 41 | A | 97 | 01100001 | 141 | 61 | a |
| 2 | 00000010 | 002 | 02 | STX | 34 | 00100010 | 042 | 22 | “ | 66 | 01000010 | 102 | 42 | B | 98 | 01100010 | 142 | 62 | b |
| 3 | 00000011 | 003 | 03 | ETX | 35 | 00100011 | 043 | 23 | # | 67 | 01000011 | 103 | 43 | C | 99 | 01100011 | 143 | 63 | c |
| 4 | 00000100 | 004 | 04 | EOT | 36 | 00100100 | 044 | 24 | $ | 68 | 01000100 | 104 | 44 | D | 100 | 01100100 | 144 | 64 | d |
| 5 | 00000101 | 005 | 05 | ENQ | 37 | 00100101 | 045 | 25 | % | 69 | 01000101 | 105 | 45 | E | 101 | 01100101 | 145 | 65 | e |
| 6 | 00000110 | 006 | 06 | ACK | 38 | 00100110 | 046 | 26 | & | 70 | 01000110 | 106 | 46 | F | 102 | 01100110 | 146 | 66 | f |
| 7 | 00000111 | 007 | 07 | BEL | 39 | 00100111 | 047 | 27 | ‘ | 71 | 01000111 | 107 | 47 | G | 103 | 01100111 | 147 | 67 | g |
| 8 | 00001000 | 010 | 08 | BS | 40 | 00101000 | 050 | 28 | ( | 72 | 01001000 | 110 | 48 | H | 104 | 01101000 | 150 | 68 | h |
| 9 | 00001001 | 011 | 09 | HT | 41 | 00101001 | 051 | 29 | ) | 73 | 01001001 | 111 | 49 | I | 105 | 01101001 | 151 | 69 | i |
| 10 | 00001010 | 012 | 0A | LF | 42 | 00101010 | 052 | 2A | * | 74 | 01001010 | 112 | 4A | J | 106 | 01101010 | 152 | 6A | j |
| 11 | 00001011 | 013 | 0B | VT | 43 | 00101011 | 053 | 2B | + | 75 | 01001011 | 113 | 4B | K | 107 | 01101011 | 153 | 6B | k |
| 12 | 00001100 | 014 | 0C | FF | 44 | 00101100 | 054 | 2C | , | 76 | 01001100 | 114 | 4C | L | 108 | 01101100 | 154 | 6C | l |
| 13 | 00001101 | 015 | 0D | CR | 45 | 00101101 | 055 | 2D | - | 77 | 01001101 | 115 | 4D | M | 109 | 01101101 | 155 | 6D | m |
| 14 | 00001110 | 016 | 0E | SO | 46 | 00101110 | 056 | 2E | . | 78 | 01001110 | 116 | 4E | N | 110 | 01101110 | 156 | 6E | n |
| 15 | 00001111 | 017 | 0F | SI | 47 | 00101111 | 057 | 2F | / | 79 | 01001111 | 117 | 4F | O | 111 | 01101111 | 157 | 6F | o |
| 16 | 00010000 | 020 | 10 | DLE | 48 | 00110000 | 060 | 30 | 0 | 80 | 01010000 | 120 | 50 | P | 112 | 01110000 | 160 | 70 | p |
| 17 | 00010001 | 021 | 11 | DC1 | 49 | 00110001 | 061 | 31 | 1 | 81 | 01010001 | 121 | 51 | Q | 113 | 01110001 | 161 | 71 | q |
| 18 | 00010010 | 022 | 12 | DC2 | 50 | 00110010 | 062 | 32 | 2 | 82 | 01010010 | 122 | 52 | R | 114 | 01110010 | 162 | 72 | r |
| 19 | 00010011 | 023 | 13 | DC3 | 51 | 00110011 | 063 | 33 | 3 | 83 | 01010011 | 123 | 53 | S | 115 | 01110011 | 163 | 73 | s |
| 20 | 00010100 | 024 | 14 | DC4 | 52 | 00110100 | 064 | 34 | 4 | 84 | 01010100 | 124 | 54 | T | 116 | 01110100 | 164 | 74 | t |
| 21 | 00010101 | 025 | 15 | NAK | 53 | 00110101 | 065 | 35 | 5 | 85 | 01010101 | 125 | 55 | U | 117 | 01110101 | 165 | 75 | u |
| 22 | 00010110 | 026 | 16 | SYN | 54 | 00110110 | 066 | 36 | 6 | 86 | 01010110 | 126 | 56 | V | 118 | 01110110 | 166 | 76 | v |
| 23 | 00010111 | 027 | 17 | ETB | 55 | 00110111 | 067 | 37 | 7 | 87 | 01010111 | 127 | 57 | W | 119 | 01110111 | 167 | 77 | w |
| 24 | 00011000 | 030 | 18 | CAN | 56 | 00111000 | 070 | 38 | 8 | 88 | 01011000 | 130 | 58 | X | 120 | 01111000 | 170 | 78 | x |
| 25 | 00011001 | 031 | 19 | EM | 57 | 00111001 | 071 | 39 | 9 | 89 | 01011001 | 131 | 59 | Y | 121 | 01111001 | 171 | 79 | y |
| 26 | 00011010 | 032 | 1A | SUB | 58 | 00111010 | 072 | 3A | : | 90 | 01011010 | 132 | 5A | Z | 122 | 01111010 | 172 | 7A | z |
| 27 | 00011011 | 033 | 1B | ESC | 59 | 00111011 | 073 | 3B | ; | 91 | 01011011 | 133 | 5B | [ | 123 | 01111011 | 173 | 7B | { |
| 28 | 00011100 | 034 | 1C | FS | 60 | 00111100 | 074 | 3C | < | 92 | 01011100 | 134 | 5C | \ | 124 | 01111100 | 174 | 7C | \| |
| 29 | 00011101 | 035 | 1D | GS | 61 | 00111101 | 075 | 3D | = | 93 | 01011101 | 135 | 5D | ] | 125 | 01111101 | 175 | 7D | } |
| 30 | 00011110 | 036 | 1E | RS | 62 | 00111110 | 076 | 3E | > | 94 | 01011110 | 136 | 5E | ^ | 126 | 01111110 | 176 | 7E | ~ |
| 31 | 00011111 | 037 | 1F | US | 63 | 00111111 | 077 | 3F | ? | 95 | 01011111 | 137 | 5F | _ | 127 | 01111111 | 177 | 7F | DEL |

# Extended ASCIIs

- Also known as ASCII 8
- As we have already mentioned ASCII is a 7-bit character-encoding, but the many computers are based around 8-bit bytes.
- Using 8-bits per character allows up to 256 different characters, from 0000000 binary (0 denary, 0 hexadecimal) to 1111111 binary (127 decimal, 7F hexadecimal).

**NOTE**

- The character codes are grouped and run in sequence; i.e. If **A is 65** then **C must be 67**.
- The pattern applies to other groupings such as digits and lowercase letters, so you can say that since **7 is 55**, **9 must be 57**.

- Notice that the ASCII code value for 5 **(0011 0101)** is different from the pure binary value for **5 (0000 0101)**. That's why you cannot calculate with numbers which are input as strings.
- Another example, the ASCII value 0011 0100 will print the character 4, the binary value is actually equal to the decimal number 52.

# EBCDIC

- The character encoding that IBM developed was called EBCDIC (pronounced "**ebb-see-dik**") which stands for **Extended Binary-Coded Decimal Interchange Code**.
- EBCDIC is an 8-bit code, which means that 8 binary digits are used to represent each character. This potentially allows for 256 different characters, from 0000000 binary (0 decimal, 0 hexadecimal) to 11111111 binary (255 decimal, FF hexadecimal). However, in EBCDIC not all combinations are considered valid or used.
- As is the case with ASCII, EBCDIC includes both printable characters and non-printable control characters.

- The ordering of the characters in EBCDIC can be somewhat surprising. For example, in ASCII the 26 letters of the alphabet have consecutive character codes (from 65 denary to 90 denary for uppercase letters, and from 97 denary to 122 denary for lowercase letters). In contrast in EBCDIC, the letters do not have consecutive codes.
- The EBCDIC character set does not include several of the characters commonly used in programming and network communications such as curly braces { }.

| Char | EBCDIC Code | | Hex |
|---|---|---|---|
| | Zone | Digit | |
| A | 1100 | 0001 | C1 |
| B | 1100 | 0010 | C2 |
| C | 1100 | 0011 | C3 |
| D | 1100 | 0100 | C4 |
| E | 1100 | 0101 | C5 |
| F | 1100 | 0110 | C6 |
| G | 1100 | 0111 | C7 |
| H | 1100 | 1000 | C8 |
| I | 1100 | 1001 | C9 |
| J | 1101 | 0001 | D1 |
| K | 1101 | 0010 | D2 |
| L | 1101 | 0011 | D3 |
| M | 1101 | 0100 | D4 |

| Char | EBCDIC Code | | Hex |
|---|---|---|---|
| | Zone | Digit | |
| N | 1101 | 0101 | D5 |
| O | 1101 | 0110 | D6 |
| P | 1101 | 0111 | D7 |
| Q | 1101 | 1000 | D8 |
| R | 1101 | 1001 | D9 |
| S | 1110 | 0010 | E2 |
| T | 1110 | 0011 | E3 |
| U | 1110 | 0100 | E4 |
| V | 1110 | 0101 | E5 |
| W | 1110 | 0110 | E6 |
| X | 1110 | 0111 | E7 |
| Y | 1110 | 1000 | E8 |
| Z | 1110 | 1001 | E9 |

| Character | EBCDIC Code | | Hexadecimal Equivalent |
| --- | --- | --- | --- |
| | Zone | Digit | |
| 0 | 1111 | 0000 | F0 |
| 1 | 1111 | 0001 | F1 |
| 2 | 1111 | 0010 | F2 |
| 3 | 1111 | 0011 | F3 |
| 4 | 1111 | 0100 | F4 |
| 5 | 1111 | 0101 | F5 |
| 6 | 1111 | 0110 | F6 |
| 7 | 1111 | 0111 | F7 |
| 8 | 1111 | 1000 | F8 |
| 9 | 1111 | 1001 | F9 |

## UNICODE

The biggest limitation with ASCII and EBCDIC is the relatively small number of different characters available. While enough characters are available for English text, neither has nearly enough characters available to support the full range of widely used typographical and mathematical symbols, let alone all accented Latin characters, national currency symbols, and the various alphabets and symbols used in languages such as Chinese, Korean, Japanese, Greek,Russian, Arabic and Hebrew.

The solution to this issue was the development of Unicode (also known as the **Universal Coded Character Standard or UCS** which potentially allows for up to 1,114,112 different characters (although only 149,186 have been defined as of September 2022 in the Unicode 15.0.0 standard)

- Provides a consistent way of encoding multilingual text
- Defines codes for characters used in all major languages of the world
- Defines codes for special characters, mathematical symbols, technical symbols etc.
- Capacity to encode as many as a million characters
- Assigns each character a unique numeric
- Affords simplicity and consistency of ASCII, even corresponding characters have same code
- Some Unicode character encodings include: UTF-8, UTF-16, UTF-32

# ERROR DETECTION & CORRECTION

# ERROR DETECTION

In the case of digital systems set up, error occurrence is a common phenomenon. And for that the first step is to detect the error and after that errors are corrected.The most common cause for errors is that the noise creeps into the bit stream during the course of transmission from the transmitter to the receiver. And if these errors are not detected and corrected the result could be disastrous as the digital systems are very much sensitive to errors and will malfunction due to the slightest of errors in transmitted codes.

We will discuss the use Parity checking method for **error detection and correction** such as the addition of extra bits which are also called check bits, sometimes they are also called redundant bits as they don’t have any information in them.

# PARITY CHECKING

## PARITY BIT

A parity bit is added to the transmitted strings of bits during transmission from transmitters to detect any error in the data when they are received at the receiver end. Basically, a parity code is nothing but an extra bit added to the string of data. Now there are two types of parity these are **even parity** and **odd parity.**

In **even parity**, the total number of bits which are set to 1 in the data is counted: if the total is odd then the parity bit is set to 1, but if the total is even then the parity bit is set to 0. Thus, in even parity, the total number of bits set to 1 (including both the data and the parity bit itself) will always be even.

Here are some examples of even parity:

If we need to generate a parity bit, using an even parity protocol, for the data **1010011**, the parity bit would be 0 (as there are already four 1s, an even number of 1s, in the data). The byte would hold **01010011**.

If we need to generate a parity bit, using an even parity protocol, for the data **1010010**, the parity bit would be 1 (as there are three 1s, an odd number of 1s, in the data, and we need one more 1 to get an even number of 1s). The byte would hold **11010010**.

In **odd parity,** the total number of bits which are set to 1 in the data is counted: if the total is odd then the parity bit is set to 0, but if the total is even then the parity bit is set to 1. Thus, in odd parity, the total number of bits set to 1 (including both the data and the parity bit itself) will always be odd.

Here are some examples of odd parity:

If we need to generate a parity bit, using an odd parity protocol, for the data **1010011**, the parity bit would be 1 (as there are four 1s, an even number of 1s, in the data, and we need one more 1 to get an odd number of 1s). The byte would hold **11010011**.

If we need to generate a parity bit, using an odd parity protocol, for the data **1010010**, the parity bit would be 0 (as there are already three 1s, an odd number of 1s, in the data). The byte would hold **01010010**.