Data representation

The present chapter is an extension of the previous chapter. As the arithmetic unit of a digital system recognizes only the binary states 0 and 1, a code is necessary to manipulate and transfer alphanumeric data (numbers, letters, special characters).

0.1 Binary Codes

0.1.1 Straight Binary

Straight Binary code is simply the radix 2 number system, It is used to represent natural numbers.

Decimal	Straight Binary Code
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Example 1. Going from $3 = 11_2$ to $4 = 100$, two bits change. This problem is solved by the following code.

0.1.2 Gray code

Gray code (or reflected binary code) is a non-weighted code, as it does not ascribe a specific weight to each bit position. It is not used for arithmetic calculations. The process of generation of higher-bit Gray codes using the reflect-and-prefix method is illustrated in Table 0.1.2 ; the columns of bits between those representing the Gray codes give the intermediate step of writing the code followed by the same written in reverse order.

Table 0.1.2 lists the binary and Gray code equivalents of decimal numbers $0 - 15$, an examination shows that the last and the first entry also differ by only 1 bit. This is known as the cyclic property of the Gray code.

TABLE 1 – Generation of higher-bit Gray code numbers

One-bit Gray code		Two-bit Gray code		Three-bit Gray code		Four-bit Gray code
0	0	00	00	000	000	0000
1	1	01	01	001	001	0001
	1	11	11	011	011	0011
	0	10	10	010	010	0010
			10	110	110	0110
			11	111	111	0111
			01	101	101	0101
			00	100	100	0100
					100	100
					101	1101
					111	1111
					110	1110
					010	1010
					011	1011
					001	1001
					000	1000

TABLE 2 – Gray code

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

Straight Binary-Gray code and Gray code-Straight Binary conversions

The conversion of a Straight Binary number to Gray code is carried out by making use of the following observations :

- — the most significant Gray code bit situated to the extreme left, is the same as the corresponding MSB for the Straight Binary number.
- — starting from the left, add, without taking into account the carry-out bit, each pair of adjacent bits to obtain the next bit in Gray code.

To convert Gray code to a Straight Binary number :

— the MSB of the Straight Binary number, located at the extreme left, is identical to the corresponding Gray code bit ;
— starting from the left, add each new bit of the Straight Binary code to the next bit of the Gray code, without taking into account any carry-out bit, to obtain the next bit of the Straight Binary code.

Example 2. 1. Convert the Straight Binary number $(101101)_2$ to Gray code.

$$\begin{array}{ccccccccccc} 1 & + & 0 & + & 1 & + & 1 & + & 0 & + & 1 \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ 1 & & 1 & & 1 & & 0 & & 1 & & 1 \end{array}$$

2. Convert the Gray code $(110011)_{GR}$ to a Straight Binary number.

$$\begin{array}{ccccccccccc} 1 & & 1 & & 0 & & 0 & & 1 & & 1 \\ \downarrow & \nearrow & \downarrow & \nearrow & \downarrow & \nearrow & \downarrow & \nearrow & \downarrow & \nearrow & \downarrow \\ 1 & & 0 & & 0 & & 0 & & 1 & & 0 \end{array}$$

0.1.3 Binary Coded Decimal

The binary coded decimal (BCD) is a type of binarry code used to represent a given decimal number in an aquivalent binary form. The BCD equivalent of a decimal number is written by replacing each decimal digit with its four-bit binary equivalent. As an example, the BCD equivalent of 425 is written as $(0100\ 0010\ 0101)_{BCD}$. Table 0.1.2 lists the BCD code.

TABLE 3 – **BCD code**

Decimal	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD-to-Binary Conversion

A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent.

Example 3. Find the binary equivalent of the BCD number $(0110\ 1000\ 0011\ 1001)_{BCD}$.

The corresponding decimal number is :6839, therefore

6839	2											
6838	3419	2										
1	3418	1709	2									
	1	1708	854	2								
		1	854	427	2							
			0	426	213	2						
				1	212	106	2					
					1	106	53	2				
						0	52	26	2			
							1	26	13	2		
								0	12	6	2	
									1	6	3	2
										0	2	1
											1	

$\longleftarrow$

$6839 = 1101010110111_2$

Binary-to-BCD Conversion

The process of binary-to-BCD conversion is the same as the process of BCD-to-binary conversion executed in reverse order.

Example 4. Find the BCD equivalent of the binary number 100001110011. The decimal equivalent of this binary number is 4323

0.1.4 Excess-3 Code

The excess-3 code is another important BCD code. The excess-3 for a given decimal number is determined by adding $'3'$ to each decimal digit in the given number and then replacing each digit of the newly found decimal number by its four-bit binary equivalent. Table 4 lists the Excess-3 code for the decimal numbers $0 - 9$.

Table 4 – **Excess-3 Code**

Decimal number	Excess-3 code	Decimal number	Excess-3 code
0	0011	5	1000
1	0100	6	1001
2	0101	7	1010
3	0110	8	1011
4	0111	9	1100

Example 5. Find the excess-3 code for the decimal number 541.

— The addition of $'3'$ to each digit yields the three new numbers $'8','7'$ and $'4'$.

— The corresponding four-bit binary equivalents are $1000, 0111$ and 0100 respectively.

— The excess-3 code for 541 is therefore given by : 100001110100_{XS-3}.

The equivalent decimal number to a given excess-3 code can be determined by first splitting the number into four-bit groups, starting from the right, and then subtracting 0011 from each four-bit group. The new number is the BCD equivalent of the given excess-3 code, which can subsequently be converted into the equivalent decimal number.

Example 6. Find the decimal equivalent of the excess-3 number $(010111000011)_{XS-3}$.

Subtracting 0011 from each four-bit group, we obtain the BCD number code 0011 1001 0000, so the decimal equivalent is : 390.

0.2 Alphanumeric Codes

Alphanumeric codes, also called UTF character codes, are binary codes used to represent alphanumeric data. The codes write alphanumeric data, including letters of the alphabet, numbers, mathematical symbols and punctuation marks, in a form that is understandable and processable by a computer. These codes enable us to interface input-output devices such as keyboards, printers, VDUs, etc, with the computer. Two widely used alphanumeric codes include the ASCII and EBCDIC codes but they have a limitation in terms of the number of characters they can encode, so they not permit multilingual computer processing. Unicode, developed jointly by the Unicode Consortium and the International Standards Organization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by compters.

0.2.1 ASCII code

The ASCII (American Standard Code for Information Interchange), pronounced 'ask-ee', is strictly a seven-bit code based on the English alphabet, ASCII codes are used to represent alphanumeric data in computers, communications equipment and other devices. It is a seven-bit code, it can at the most represent 128 characters. It currently defines 95 printable characters including 26 upper-case letters (A to Z), 26 lower-case letters (a to z), 10 numerals (0 to 9) and 33 special characters including mathematical symbols, punctuation marks and space character. It defines codes for 33 nonprinting, mostly obsolete control characters that affect how text is processed. Table lists the ASCII codes for all 128 characters. When the ASCII code was introduced, many computers dealt with eight-bit groups (or bytes) as the smallest unit of information. The eighth bit was commonly used as a parity bit for error detection on communication lines and other device-specific functions. Machines that did not use the parity bit typically set the eighth bit to $'0'$.

Example 7. Represent YES in ASCII code (hexadecimal). From Table5 ; we have Y :59, E :45, S :53. Therefore YES is coded by 59 45 53.

0.2.2 EBCDIC code

The EBCDIC (Extended Binary Coded Decimal Interchange Code), pronounced 'eb-si-dik', is another widely used alphanumeric code, mainly popular with larger systems. The code was created by IBM to extend the binary coded decimal that existed at that time. All IBM mainframe computer peripherals and operating systems use EBCDIC code, and their operating systems provide ASCII and Unicode modes to allow translation between different encodings. It is an eight-bit code and thus can accommodate up to 256 characters. A single byte in EBCDIC is divided into two nibbles (four-bit groups)

Example 8. 'K' is coded in EBCDIC by $D2$ in hexadecimal and $\underbrace{1101}_{zone}\ \underbrace{0010}_{digit}$; 'zone' represents the category and 'digit' identifies the specific character.

0.2.3 Unicode

As briefly mentioned in the earlier sections, encodings such as ASCII, EBCDIC and their variants do not have a sufficient number of chracters to be able to encode alphanumeric data of all forms, scripts and languages. Two different encodings may use the same number for two different characters or different numbers for the same characters. For example, 4B (in hex) represents the upper-case letter 'K' in ASCII code and the point '.' in the EBCDIC code.

Unicode developed jointly by the Unicode Consortium and the International Organization for Standardization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. Different characters in Unicode are represented by a hexadecimal number preceded by 'U+'.

Example 9. 'T' is coded by $U + 0054$ and 't' is coded by $U + 021B$.

UTF Code

The Unicode Standard provides three distinct encoding forms for Unicode characters, using 8-bit, 16-bit and 32-bit units. These are named UTF-8, UTF-16 and UTF-32, respectively. The "UTF" is a carryover from earlier terminology meaning Unicode Transformation Format. Each of these three encoding forms is an equally legitimate mechanism for representing Unicode characters, each has advantages in different environements.

To meet the requirement of byte-oriented, ASCII-based systems, one of the third encoding form specified by the Unicode Standard is UTF-8, we use one byte for characters in ASCII (7bits), and two, three or four bytes for the other characters. It is more space-efficient and more compatible with ASCII.

From Unicode to UTF-8

For encoding character in UTF-8 we follow the following steps.

— The number of each character is provided by the Unicode standard.

— Characters with numbers from 0 to 127 are encoded in one byte, with the most significant bit always being zero.

— Characters with numbers higher than 127 are encoded using multiple bytes. In this case, the most significant bits of the first byte form a sequence of 1s of a length equal to the number of bytes used to encode the character, with the following bytes having 10 as their most significant bits.

Binary UTF-8 representation	Meaning
$0xxxxxxx$ (Ascii)	For 1 to 7 significant bits (1 byte)
$110xxxxx\ 10xxxxxx$	For 8 to 11 significant bits (2 bytes)
$1110xxxx\ 10xxxxxx\ 10xxxxxx$	For 12 to 16 significant bits (3 bytes)
$11110xxx\ 10xxxxxx\ 10xxxxxx\ 10xxxxxx$	For 17 to 21 significant bits (4 bytes)

Example 10. Let us write the UTF-8 code of the symbol € coding in Unicode by U+20AC

1. Write 20AC in binary code : 0010000010101100.
2. We have 14 significant bites : 10000010101100.
3. We encode the symbol in 3 bytes :11100010 10000010 10101100
4. We convert in hexadecimal : $E282AC$.

0.3 Representation of numbers

0.3.1 Integers

Unsigned representation

We can easily proof that the maximal positif integer representable in binary code with n digits ; is $2^n - 1$. Suppose N be the maximal positif integer, in n bits binary code $N = \sum_{i=0}^{n-1} 2^i = \underbrace{2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}}_{\text{sum of a geometric sequence}} = \frac{2^n - 1}{2 - 1} = 2^n - 1.$

Therefore, an $n-$bit binary representation can be used to represent decimal numbers in the range of 0 to $2^n - 1$; n représents the magnitude and $c = 2^{n-1}$ the capacity of register containing this number. For sufficiently large n, we can write $c \simeq 2^n$, then $n \simeq log_2 c$. This relationship allows for estimating the length of a register who can contain a given number.

Example 11. Let us find the minimum size of a register required to represent integers less than or equal 300. We must search the naturel number n such that $2^n \simeq 300$, so $n \simeq log_2 300 \simeq 8.22$. So, it is necessary to design a register with capacity at least nine bits.

Sign-magnitude representation

In the sign-bit representation of positive and negative decimal numbers, the MSB represents the 'sign', with a $'0'$ denothing a plus sign and a $'1'$ denoting a minus sign. The remaining bits represent the magnitude. In the following, we represent a signed number using 8, 16, 32,... bits.

Example 12. +7 = | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | and -7 = | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

An n-bit binary representation can be used to represent decimal numbers in the range of $-(2^{n-1}$ to $+(2^{n-1} - 1$; we note this representation by SM (Sign-magnitud).

Example 13. In 4-bit SM representation, we can represent decimal numbers between -7 and $+7$ as follow

Decimal	SM
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
+0	0000
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

Remark. The sign-magnitude representation presents two problems. Firstly in mathematics $+0 = -0 = 0$ but we remark that zero has two representations in SM representation. Secondly, this representation is not appropriate for addition operations. For example $(-4)+(+3)=+1$ but in SM representation (for reduction of magnitude we take 4 bits) we have $(1100)_{SM}+(0011)_{SM}=(1111)_{SM}=-7$, it's incorrect.

1's Complement

We first define the 1's Complement of binary number.

Definition 1. *To obtain the 1's Complement of binary number, we inverse 1 to 0 and 0 to 1.*

Example 14. Let us define the 1's Complement of 10010_2.
$10010_2=(01101)_{C1}$

Definition 2. *To obtain the 1's Complement of a sign-magnitude number, the positive numbers remain unchanged and for negatif number; we keep the sign bit and convert the remaining bits to 1's Complement.*

Example 15. The 1's Complement of the decimal integer +9 is $(00001001)_{C1}=(00001001)_{SAV}$, the 1's Complement of the decimal integr i $-9=-1001_2=(10001001)_{SM}=(11110110)_{SM}$.

Again, n bit notation can be used to represent numbers in the range from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$ using the 1's complement format.

Example 16. In 4−bit 1's Complement representation, we can represent decimal numbers between −7 and + 7 as follow

Decimal	SM	1's Complement
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
-0	1000	1111
-1	1001	1110
-2	1010	1101
-3	1011	1100
-4	1100	1011
-5	1101	1010
-6	1110	1001
-7	1111	1000

Remark. Again, in 1's Complement representation zero has two representations and this is a drawback.

1's Complement addition

One's complement addition is based on the following principle.

- If no carry is generated by the sign bit, the result is accurate and expressed in 1's Complement.
- If a carry is generated by the sign bit, it will be added to the result of the operation which is expressed in 1's Complement.

Example 17. Let us do the following 1's complement addition. $35 + (-25) = (+100011)_2 + (-11001)_2 = (00100011)_{SM} + (10011001)_{SM} = (00100011)_{C1} + (11100110)_{C1}$

$$\begin{array}{c|cccccccc}
 & {}^{1}0 & {}^{1}0 & 1 & 0 & {}^{1}0 & {}^{1}0 & 1 & 1 \\
+ & & & & & & & & \\
 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\hline
1 \hookrightarrow & 0 & 0 & 0 & 0 & 1 & 0 & {}^{1}0 & 1 \\
+ & & & & & & & & 1 \\
\hline
= & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}$$

$00001010_{C1} = +1010_2 = +10$, the result is correct.

$15 - 34 = +15 + (-34) = (+1111)_2 + (-100010)_2 = (00001111)_{SM} + (10100010)_{SM} = (00001111)_{C1} + (11011101)_{C1}$

$$\begin{array}{c|cccccccc}
 & 0 & 0 & {}^{1}0 & {}^{1}0 & {}^{1}1 & {}^{1}1 & {}^{1}1 & 1 \\
+ & & & & & & & & \\
 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
\hline
= & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}$$

$(11101100)_{C1} = (10010011)_{SAV} = (-10011)_2 = -19$, the result is correct.

2's Complement

We first define the 2's Complement of binary number.

Definition 3. *To obtain the 2's Complement of binary number, we add 1 to the 1's Complement.*

Example 18. Let us give the 2's Complement of 10011. We first define the 1's Complement of this binary number : $10011_2 = (01100)_{C1}$, then we add 1 to obtain the 2's Complement $01100 + 1 = 01101$, therefore $10011_2 = (01100)_{C1} = (1101)_{C2}$

Definition 4. *To obtain the 2's Complement of a sign-magnitude number, the positive numbers remain unchaged and for negatif number ; we keep the sign bit and convert the remaining bits to 2's Complement.*

Example 19. The 2's Complement of the decimal integer +10 is $(00001010)_{C2} = (00001010)_{C1} = (00001010)_{SAV}$, the 2's Complement of the integer $-10 = -0001010_2 = (10001010)_{SM} = (11110101)_{C1} = (11110101 + 1)_{C2} = (11110110)_{C2}$.

An other method to obtain the 2's Complement of integer numbers is illustrated by the following definition.

Definition 5. *To obtain the 2's Complement of a sign-magnitude number, the positive numbers remain unchaged and for negatif number ; we keep the sign bit and starting from the right, we copy all the zeros and the first encountered 1, then we invert the remaining bits.*

Example 20.

1. The 2's Complement of the decimal integer -10 is $(11110110)_{C2}$.
2. Let us give the 2's Complement of the decimal integer -15. We first the SAV corresponding number :$(10001111)_{SM} = (11110001)_{C2}$.

Remark.

1. The $n-$bit notation of the 2's Complement format can be used to represent all decimal numbers from -2^{n-1} to $+(2^{n-1} - 1)$
2. $1\underbrace{00\cdots0}_{n \text{ fois}}$ represents the smallest value on n bits in 2's Complement representation.

Example 21. In $4-$bit 2's Complement representation, we can represent decimal numbers from -8 to $+7$ as follow.

Decimal	SM	1's Complement	2's Complement
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	1000	1111	/
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	/	/	1000

Remark.
1. We see that zero has a unique representation.
2. 1000 which represented 0 in SM representation, represents -8 which is the smallest value in 4-bit 2's Complement representation.

2's Complement addition

2's Complement addition is performed in the same manner as for 1's Complement, except that we do not carry over the overflow but ignore it and the result in 2's Complement.

Example 22. Let us do the following 2's Complement addition. $35 + (-25) = (00100011)_{SM} + (10011001)_{SM} = (00100011)_{C2} + (11100111)_{C2}$, the result is correct.

$$\begin{array}{cccccccccc}
 & & {}^{1}0 & {}^{1}0 & 1 & 0 & {}^{1}0 & {}^{1}0 & {}^{1}1 & 1 \\
+ & & & & & & & & & \\
 & & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
\hline
= & \cancel{1} & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}$$

$(00001010)_{C2} = (00001010)_{SM} = (1010)_2 = 10.$

0.4 Fractional numbers

0.4.1 Fixed-point

A fixed-point number is represented as a binary integer. The position of the decimal point is managed by the programmer, and it's a drawback added to the limitation of values. It is represented as follow.

Sign	Entire part with n bits	Fractional part with p bits

Example 23. Let us represent a number in 6 bits; one bit for the sign, three bits for entire part and two bits for fractional part. The minimum value is represented by $(1\ 111\ 11)_2 = -7.75$ and the maximum value is $(0\ 111\ 11)_2 = +7.75$.

0.4.2 Floating-Point Numbers

At the begining the Floating-point representation was not standardized and each computer used its own format. Several standards were defined ; among them the $IEEE$ 754 standard (Institute of electrical and electronics Engineers).

Floating-point numbers are in general expressed in the form

$$N = \sigma M b^E, \tag{1}$$

where σ is the sign $\pm$, M is the fractional part called the significand or mantissa, E is the integer part, called the exponent, and b is the base of the number system or numeration. Fractional part M is a $p-$digit number of the form $(d.ddd \cdots d)$, each digit d is an integer between 0 and $b-1$.

Equation 1 in the case of decimal, hexadecimal and binary number systems will be written as follows :

— Decimal system

$$N = \sigma M 10^E. \tag{2}$$

— Hexadecimal system

$$N = \sigma M 16^E. \tag{3}$$

— Binary system

$$N = \sigma M 2^E. \tag{4}$$

Example 24. We represent 0.00001453, 1453_8, $(643.ACE)_{16}$ in floating-point notation.

— $0.00001453 = 1.453 \times 10^{-5}$.

— $1453 = 1.453 \times 8^3$.

— $643.ACE = 6.43ACE \times 16^2$.

In the case of normalized binary numbers, the leading digit which is the most significant bit is always $'1'$ and thus does not need to be stored explicitly. While expressing a given mixed binary number as a floating-point number, the radix point is so shifted as to have the most significant bit immediately to the right of the radix point as a $'1'$. The mantissa and the exponent can have a positive or a negative value.

Example 25. Let us represent the mixed binary numbers $(11.0111)_2$, $(0.000101)_2$, $(-0.00000011)_2$ in floating-point notation.

1. $(11.0111)_2$ will be represented as follow :

$$0.110111 \times 2^2 = .110111E + 0010.$$

   Here 0.110111 is the mantissa and $E + 0010$ implies that the exponent is $+2$.

2. $(0.000101)_2$ will be written as

$$0.101 \times 2^{-3} = .101E - 0011,$$

   0.101 is the mantissa and $E - 0011$ implies that the exponent is -3.

3. $(-0.00000011)_2$ will be written as

$$-0.11 \times 2^{-6} = -.11E - 0110.$$

   Here -0.11 is the mantissa and $E - 0110$ indicates an exponent of -6.

In each of these cases, and if we want to write the mantissa with eight bits, we will represent it as follows :

$$.11011100, \ .10100000, \ .11000000.$$

0.4.3 IEEE-754 formats

The IEEE-754 floating point is the most commonly used representation for real numbers on computer. Table 6 lists characteristic parameters of single-precision and double-precision. Floating-point numbers represented in IEEE-754 format have three components including the sign, the exponent and the mantissa. The $n-$bit exponent field needs to represent both positive and negative exponent values. To achieve this, a bias equal to $2^{n-1}-1$ is added to the actual exponent in order to obtain the stored exponent. For the case of single-precision format, we add $2^{8-1}-1=127$ to the actual exponent then we obtain the biased exponent which is noted by E_b. Figure 1 shows the basics constituent parts of the single-precision format.

Sign (1 bit)	Biased Exponent (8 bits)	Mantissa (23 bits)

FIGURE 1 – Single-precision format

Example 26. 1. Let us represent the number 2654 in IEEE-754 single-precision format.
$2654 = 101001011110_2 = 1.01001011110 \times 2^{11}$. The three components are :
— Sign = 0.
— Mantissa = 01001011110.
— Actual exponent=11 and biased exponent ; $E_b = 11 + 127 = 138 = 10001010_2$.
Therefore, we represent the number as follow.

$\underbrace{0}_{Sign}\ \underbrace{10001010}_{Biased\ exponent}\ \underbrace{01001011110000000000000}_{Mantissa}$. We change the number to hexadecimal form in order to make writing easier ;

$$0100\ 0101\ 0010\ 0101\ 1110\ 0000\ 0000\ 0000 = 452E000_{16}.$$

2. Let us represent the hexadecimal IEEE-754 single-precision format $D2AC5000$ in decimal.
$C32C5000 = 1100\ 0011\ 0010\ 1100\ 0101\ 0000\ 0000\ 0000 = 11000011001011000101000000000000.$
The three components are :
— Sign =1, hence the number is negative.
— Biased exponent $=10000110_2 = 134$, actual exponent is given by $E = 134 - 127 = 7$.
— Mantissa=01011000101.
So the number is $-1.01011000101 \times 2^7 = -10101100.0101_2 = -(2^7 + 2^5 + 2^3 + 2^2 + 2^{-2} + 2^{-3} = -172.375$.

Table 5 – ASCII Code table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	32	20	SP	64	40	@	96	60	’
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	‘	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	HT	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	NP	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	∧	126	7E	∼
31	1F	US	63	3F	?	95	5F	-	127	7F	DEL

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	Fire separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	13 Unit separator
SP	Space	DEL	Delete

Table 6 – characteristic parameters of IEE-754 format

Precision	Sign (bit)	Exponent (bits)	Mantissa (bits)	Total length (bits)
Single	1	8	23	32
Double	1	11	52	64