Data representation

A. Chouraqui-Benchaib

Tlemcen University,
Tlemcen, Algeria.

Outline

Definition-Straight Binary

Definition

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

Example

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

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 the table (see your manuscript); 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.

Gray code

See the table which 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.

Straight Binary-Gray code & Gray code-Straight Binary

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 corrresponding 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

Example

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.

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

BCD

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 1 lists the BCD code.

BCD

Table: **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-Binary

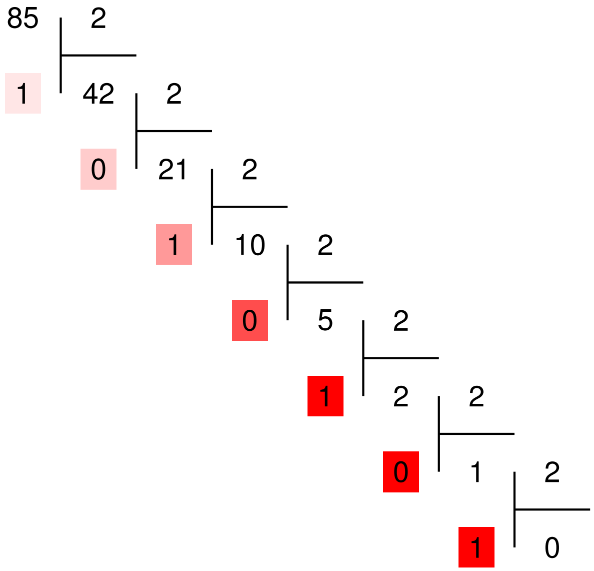
A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent.

Example

Example

Find the binary equivalent of the BCD number $(\underbrace{1000}_{8}\ \underbrace{0101}_{5})_{BCD}$,

the corresponding decimal number is:85, therefore



$$85 = 1010101_2$$

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

Find the BCD equivalent of the binary number 101000011. The decimal equivalent of this binary number is 323, then the BCD equivalent is 001100100011_{BCD}.

XS-3

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 2 lists the Excess-3 code for the decimal numbers $0 - 9$.

XS-3

Table: Excess-3 Code

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

Examples

Example

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}.

Example

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 0010 1001 0000, so the decimal equivalent is: 290.

Introduction

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.

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.

Example

Example

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

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

'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.

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

'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.

UTF-8

Binary UTF-8 representation	Meaning
0*xxxxxxx* (Ascii)	For 1 to 7 significant bits
110*xxxxx* 10*xxxxxx*	For 8 to 11 significant bits
1110*xxxx* 10*xxxxxx* 10*xxxxxx*	For 12 to 16 significant bi
11110*xxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*	For 17 to 21 significant bi

Example

Example

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*.

Unsigned representation

We can easily prove 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
$n = [log_2 c] + 1;$ where [.] denotes the floor number. This relationship allows for estimating the length of a register who can contain a given number.

Example

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 $n = [log_2 300] + 1 = 9$.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′ for a plus sign and a ′1′ for a minus sign. The remaining bits represent the magnitude. In the following, we represent a signed number using 8, 16, 32,... bits.

Example

+7 =

0	0	0	0	0	1	1	1

and -7 =

1	0	0	0	0	1	1	1

Interval of SM

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

Example

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

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

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.

Definitions

Definition

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

Example

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

Definition

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

Example

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

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

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

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

Principle of addition in 1C

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

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

$$\begin{array}{ccccccccc}
 & {}^{1}0 & {}^{1}0 & 1 & 0 & {}^{1}0 & {}^{1}0 & 1 & 1 \\
+ & & & & & & & & \\
 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\hline
1 \hookrightarrow & 0 & 0 & 0 & 0 & 1 & 0 & {}^{1}0 & 1 \\
+ & & & & & & & & 1 \\
\hline
= & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}$$

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

1's Complement

Example

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

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

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

Definitions

Definition

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

Example

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

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

Example

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

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

Definition

To obtain the 2's Complement of a sign-magnitude number, the positive numbers remain unchanged and for negative numbers; 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

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 find the SM corresponding number, then we convert to the 2's Complement:$(10001111)_{SM} = (11110001)_{C2}$.

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. $1\underbrace{00\cdots0}_{n-1\text{ times}}$ represents the smallest value on n bits in 2's Complement representation.

Example

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

1's Complement

Decimal	SM	1's Complement	2's Complement
-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 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

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

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

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

Fixed-point number

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	Enteger part with n bits	Fractional part with p bits

Example

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

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

We represent $0.00001453,\ 1453_8,\ (643.ACE)_{16}$ in floating-point notation. $0.00001453 = 1.453 \times 10^{-5};\ 1453 = 1.453 \times 8^3;\ 643.ACE = 6.43ACE \times 16^2$.

The IEEE-754 floating point is the most commonly used representation for real numbers on computer. Table 3 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.

IEEE-754 formats

Table: characteristic parameters of IEE-754 format

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

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

Figure: Single-precision format

Example

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;

$$0100\ 0101\ 0010\ 0101\ 1110\ 0000\ 0000\ 0000 = 4525E000_{16}.$$

Example

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
$-1.01011000101 \times 2^7 = -10101100.0101_2 =$
$-(2^7 + 2^5 + 2^3 + 2^2 + 2^{-2} + 2^{-3} = -172.375.$