Chapter 2

Data Representation

2.1 Introduction

Data representation implies the use (encoding) of a collection of symbols (codes) that under some rules of interpretation (decoding) provides an important element of communication between human beings. The *base* or *radix* used in a representation corresponds to the maximum number of different symbols used in a language or numeric representation. For example, 24 different symbols in the alphabet provide the code for written English language, and 10 different symbols are used to encode the decimal (base 10) representation of numeric values. A binary representation of data relies on two basic symbols: 0 and 1. Modern digital computers use the binary number system because it is easy to represent the state of physical components that observe a binary behavior in nature. For example the terms *(open, close), (on, off), (in, out), (above, below), (hot, cold), (high, low)*, etc., are tuples that can easily be represented with the symbols (0,1) which in turn are associated within a digital computing device with the presence or non-presence of an electric signal, normally a voltage level. As most numeric data representation is based on the number of digits and their position, this chapter reviews the most common positional number systems, conversions, signed number representations, and character representation as well.

Table 2.1: Common positional number representations

System	Base	Symbols
Binary	2	0, 1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

2.2 Positional Number Systems

A positional number system represents values in terms of their *radix* and the position of each digit in the representation. In the binary system the smallest unit of data is a *bit* which represents two different values (zero and one). Most applications require the use of large strings of binary data. Grouping binary data in different sets of bits derive into systems with a base that are powers of two, for example, in systems with base 4, 2 bits are required to represent a single digit; likewise octal systems ($r = 8$) require 3 bits for each digit. Hexadecimal systems ($r = 16$) require four binary digits. A *nibble* is a collection of four bits and define the basic digit in a hexadecimal representation as well as in the BCD (Binary Coded Decimal) representation, in which each decimal digit is represented with a nibble . Typical representations used in the computer literature are listed in table 2.1.

A radix-r number is encoded as a digital vector of $n + k$ digits. Let A denote a vector defined as follows:

$$A = a_{n-1}a_{n-2}...a_1a_0.a_{-1}a_{-2}\ldots a_{-k}$$

where each component $a_i, -k \leq i \leq n-1$ is referred to as the ith digit of the vector A. The weight assigned to the ith element of A is r^i. The first n digits form the integer portion of A and the remaining k digits form the fraction portion of the A. The overall value V represented is then obtained as follows:

$$\begin{aligned} V &= a_{n-1}r^{n-1} + \ldots + a_1r^1 + a_0r^0.a_{-1}r^{-1}\ldots a_{-k}r^{-k} \\ &= \sum_{i=-k}^{n-1} a_i r^i \end{aligned} \tag{2.1}$$

Note that the value V obtained with equation (2.1) is in turn the decimal representation of A.

The following examples illustrate several representations and the conversion to the value V represented.

1. For $r = 10$ and $A = 95243.25$ the value V is obtained as follows:

$$V = 9 \times 10^4 + 5 \times 10^3 + 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

2. For $r = 2$ and $A = 1011010.101$ obtain V:

$$\begin{aligned} V &= 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 \\ &\quad +1 \times 2^3 + 1 \times 2^1 + 0 \times 2^0 \\ &\quad +1 \times 2^{-1} + 1 \times 2^{-3} \\ &= 64 + 16 + 8 + 2 + .5 + .125 = 90.625 \end{aligned}$$

3. For $r = 16$ and $A = 5A7$ obtain V as follows:

$$\begin{aligned} V &= 5 \times 16^2 + 10 \times 16^1 + 7 \times 16^0 \\ &= 1280 + 160 + 7 = 1447 \end{aligned}$$

4. For $r = 8$ and $A = 2647$ then:

$$\begin{aligned} V &= 2 \times 8^3 + 6 \times 8^2 + 4 \times 8^1 + 7 \times 8^0 \\ &= 1024 + 384 + 32 + 7 = 1447 \end{aligned}$$

Notation

It is common to represent a number indicating the *radix* with a subscript. For example:

$$\begin{aligned} (1011010)_2 &= (90)_{10} \\ (5A7)_{16} &= (2647)_8 \\ &= (1447)_{10} \end{aligned}$$

Other notation used particularly in the context of assembly language include the following:

$$\begin{aligned}(5A7)_{16} &= 5A7h \\ (1011010)_2 &= 1011010b\end{aligned}$$

Also in the context of assembly language it is common that a hexadecimal number that starts with a letter from A to F is preceded by a 0. For example: $AFCDh = 0AFCDh$. Otherwise, the assembler will interpret a number as a string of characters.

2.3 Conversions

The examples shown in the previous section illustrate conversions from a number represented in any base to a decimal representation. Given a decimal number, its representation in any other base r is obtained by successively dividing the number by r and keeping track of the remainders until the last quotient obtained is no longer divisible by r. The target representation is obtained by ordering digits from left-to-right in the order in which they are generated. The following examples illustrate this simple rule for the most common representations.

Decimal-to-binary. In this case $r = 2$, hence, the decimal value given is successively divided by two. Keeping track of the remainders results in a collection of 1's and 0's. For example the binary representation of $(90)_{10}$ is obtained as follows:

90:	45	0
	22	1
	11	0
	5	1
	2	1
	1	0
	0	1

therefore, $(90)_{10} = (1011010)_2$

Decimal-to-hex. In this case, the successive divisions are by $r = 16$. The remainders are a collection of numbers between 0 and 15, with the substitution of hexadecimal symbols when appropriate. Consider the hexadecimal representation of $(6841)_{10}$:

6841:	427	9
	26	11 (B)
	1	10 (A)
	0	1

therefore, $(6841)_{10} = (1AB9)_{16}$

Decimal-to-octal. . Succesive divisions by $r = 8$ result in a set of remainders between 0 and 7. Examine for example the octal representation of $(1447)_{10}$:

$$\begin{array}{rrr} 1447: & 180 & 7 \\ & 22 & 4 \\ & 2 & 6 \\ & 0 & 2 \end{array}$$

and $(1447)_{10} = (2647)_8$

2.3.1 Other conversions

Other conversions between bases not involving $r = 10$ are simple and are illustrated in the examples that follow.

Hex-to-binary. Since $16 = 2^4$, then each hex digit takes up to four binary digits and the conversion process simply consists on converting each hex digit into its 4-bit binary representation. Example:

$$(0EAC)_{16} = (0000\ 1110\ 1010\ 1100)_2$$

Binary-to-hex. From right to left divide the binary digits into groups of four. Each group of four bits is the binary representation of a single hexadecimal digit. Example:

$$(101\ 1010)_2 = (5A)_{16}$$

Octal-to-binary. Again note that $8 = 2^3$ and indicates that each octal digit requires thee binary digits. Each digit in the octal representation is converted to its 3-bit binary representation. Example:

$$(2647)_8 = (010\ 110\ 100\ 111)_2$$

Binary-to-octal. From right to left divide digits in groups of three binary digits and obtain their corresponding octal representation. Example:

$$(010\ 110\ 100\ 111)_2 = (2647)_{8=2^3}$$

Octal-to-hex. Convert first octal-to-binary followed by a binary-to-hex conversion. Example:

$$(2647)_8 = (010\ 110\ 100\ 111)_2 = (5A7)_{16}$$

2.3.2 Conversion of Decimal Fractional Parts

Consider the conversion of the fractional part of a decimal representation into a representation A where:

$$A = (.a_{-1}a_{-2}\ldots a_{-k})_r$$

The conversion process requires a repeated multiplication by r. At each multiplication step an integer ≥ 0 is generated and the resulting fraction is again multiplied by r. The collection of these integers form the fractional part of the new representation in the order in which they are generated from left-to-right.

Example. Obtain the binary representation of $(.625)_{10}$:

$$\begin{aligned} .625 \times 2 &= 1.250 \\ .250 \times 2 &= 0.500 \\ .500 \times 2 &= 1.000 \end{aligned}$$

Thus, the resulting fraction is $A = (.101)_2$. It is easy to verify that $(.625)_{10} = (.22)_4 = (.5)_8$.

2.4 Signed Radix Numbers

In general an n-bit signed radix number can be represented as follows:

$$A = (a_{n-1}a_{n-2}\ldots a_1a_0)_r$$

where a_{n-1} bit is dedicated to represent the sign as follows:

$$a_{n-1} = \begin{cases} 0 & \text{if } V \geq 0; \\ r-1 & \text{if } V < 0, \end{cases}$$

Thus, for $r = 2$, a_{n-1} takes the value 0 to represent positive values and 1 for negative values. The remaining digits in A represent the true magnitude of A or the magnitude in a complemented form. In a positional number system, there are three conventional ways to represent positive and negative numbers: *sign magnitude, diminished-radix complement* and *radix complement.*

2.4.1 Signed magnitude

Positive integers are represented as follows:

$$A = (0a_{n-2}a_{n-3}\ldots a_1a_0)_r$$

The range of integer values V is bounded as follows: $0 \leq V \leq r^{n-1} - 1$

Negative numbers are represented as follows:

$$A = [(r-1)a_{n-2}\ldots a_1a_0]_r$$

with values V in the range: $-(r^{n-1} - 1) \leq V \leq 0$

Note that, since the most significant digit is used for the sign, then the magnitude of V falls in the range: $0 \leq |V| \leq r^{n-1} - 1$.

Therefore, positive and negative representations differ only in the sign. Typical problems encountered with the signed-magnitude representation include:

1. the value of 0 is represented as $+0$ and -0, and
2. Signs must be compared during '+' and '-' operations.

2.4.2 Diminished-radix complement

The representation of positive integer values is the same as the signed-magnitude representation:

$$A = (0a_{n-2}\ldots a_1a_0)_r$$

with values V within the same range, $0 \leq V \leq r^{n-1} - 1$ that includes the representation of zero.

However, a negative representation of A can be obtained in terms of the absolute value by complementing each digit such that:

$$\bar{A} = [(r-1)\bar{a}_{n-2}\ldots \bar{a}_1\bar{a}_0]_r$$

where $\bar{a}_i = (r-1) - a_i$

The values represented are also fall in the range $-(r^{n-1} - 1) \leq V \leq 0$, which provides a second representation of zero.

2.4.3 Radix Complement

Representation of positive integers:

$$A = (0a_{n-2} \ldots a_1 a_0)_r$$

with values V in the range $0 \leq V \leq r^{n-1} - 1$

Given $A = (0a_{n-2} \ldots a_1 a_0)_r$ a representation of negative integers is obtained as:

$$(\bar{A})_{+1} = \{[(r-1)\bar{a}_{n-2} \ldots \bar{a}_1 \bar{a}_0] + 1\}_r \tag{2.2}$$

with values in the range $-(r^{n-1}) \leq V < 0$ The procedure to obtain a radix-complement representation of negative integers implicit in equation (2.2) can be divided into three steps:

1. Diminished-radix-complement each digit, i.e., $a_i = (r-1) - a_i$,
2. Add one to the least significant digit.

Note that that a decimal representation is normally preceded with the + and - signs to denote positive and negative values respectively. In this case the two steps apply to the absolute value instead.

Example: Use six digits to obtain the 10-complement representation of -018135:

1. Derive the 9-complement of the absolute value 018135: 981864
2. Add one to the least significant digit: 981865

You may verify that the 10's complement representation of 981865 is indeed 018135.

A "simplified rule" normally applied to a fast derivation of a two's complement representation can be generalized to any base r with the following two steps:

1. From right to left obtain the r-complement of the first non-zero digit,
2. (r-1)-complement the remaining digits

Example 1: Obtain the 10's complement representation of 0189300.

1. From right to left up to the first non-zero digit: 300 becomes 700
2. The remaining digits are 9's complemented: and 0189 becomes 9810

with a result: 9810700.

Example 2: Verify that the two's complement representation of 00110010 is 11001110.

2.4.4 Two's complement

The two's complement representation is the representation of choice in the hardware implementation of modern digital devices. Two's complement not only makes possible one representation of zero but simplifies the hardware implementation of arithmetic functions because:

1. obtaining the two's complement representation of any number is straightforward,
2. the addition of negative numbers is equivalent to the addition of their two's complement representation, and
3. the four fundamental operations can always be implemented in terms of addition and subtraction operations.

Example 1: Obtain the binary representation of 50, and add it to its two's complement representation to verify a zero result.

$$\begin{array}{rl} 0\,0\,1\,1\,0\,0\,1\,0 & = \ 50 \\ 1\,1\,0\,0\,1\,1\,1\,0 & = -50 \\ \hline 0\,0\,0\,0\,0\,0\,0\,0 & = \ 0 \end{array}$$

which shows that the two's complement operation has the effect of negating the original value. Note that for r=2, negative values can be obtained directly from the binary representation considering the negative contribution of the sign bit. For example using equation (2.1), the value of -51 is obtained from its binary representation (11001101) as follows:

$$V = \sum_{i=0} -2^7 + 2^6 + 2^3 + 2^2 + 2^0 = -128 + 64 + 8 + 4 + 1 = -51$$

The two's complement representation is not fully symmetrical, i.e., the most negative value $-(r^{n-1})$ has no positive counterpart. For example, for $r = 2$ and $n = 8$, the range of values:

Table 2.2: Integer representation for $r = 2$ and $n = 4$

Bit pattern	SM	DTC	TC
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	- 3	-4
1101	-5	-2	-3
1110	-6	- 1	-2
1111	-7	- 0	-1

$$-128\ (1000\ 0000) \leq V \leq 127\ (0111\ 1111)$$

for $n = 16$

$$-32768 <= V <= 32767$$

2.5 Overflow

So far the representation of signed and unsigned integers has been reviewed. Table 2.2 shows the integer values represented with signed magnitud (SM), diminished 2's complement (DTC), and two's complement (TC), for $r = 2$ and $n = 4$. The values shown are within the range of values that can be represented.

Table 2.3 compares the range of values for several number of bits for both signed and unsigned integers. The number of bits designated to represent a value signed or unsigned determines the *precision* of the representation. Values beyond the possible range cause *overflow*. Likewise, values below the minimum boundary cause

Table 2.3: Numeric range for several precision bits

Bits (Type)	Bytes	Unsigned	Signed (TC)
4 (nibble)	1/2	0 to 16	-8 to +7
8 (byte)	1	0 to 255	-128 to +127
16 (word)	2	0 to 65,535	-32,768 to +32,767
32 (double word)	4	0 to 4,294,967,295	-2,147,483,648 to 2,147,483,647
64 (quad word)	8	0 to 18,446,744,073,709,551,615	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

an *underflow*. Overflow may arise during the execution of signed integer operations. For example, the addition of values of the same sign may lead to a result either too big or too small to be represented with the available number of bits.

2.6 Floating Point Numbers

Table 2.3 also shows the limitation of fixed-point arithmetic operations because even for a large number of precision bits the values involved in the results eventually run into overflow, particularly, in scientific applications where the use of large integers and fractions is common. A floating point representation is an alternative to deal with very large numbers, integers and fractions. The reason for using floating point representation is that the range of possible values is much greater. A floating point number is one in which the position of the point is determined within the number and it is determined at processing time. Floating point representation is similar to scientific notation in which floating point decimal numbers are displayed as decimal numbers multiplied by some power of 10 that may be positive or negative. Likewise, a binary representation of floating point numbers requires a *mantissa* that is multiplied by some power of 2. A typical notation used to express a floating point number f is as follows:

$$f = (m, e)$$

where m and e denote the mantissa and the exponent, respectively. This notation is used to express a real number:

$$f = m \times r^e$$

For example, $f = (0.00000578, 3)$ represents the following number:

$$f = 0.00000578 \times 10^{+3}$$

A shift of the mantissa k places to the left results in the same value if the exponent is decremented by k. Likewise a shift of k places to the right and incrementing the exponent by k does not change the value represented. Therefore, the following is an alternate representation of the example above:

$$f_1 = (0.57800, -2) = 0.578 \times 10^{-2}$$

where $k = 5$ shifts to the left, and $e = 3 - 5 = -2$. A *flp* number is said to be *normalized* if the leading digit in the mantissa is a non-zero digit. Thus the *normalization* process involves shifting the mantissa a number of places such that m results in a normalized mantissa and its absolute value is now within the range:

$$\frac{1}{r} \leq |m| < 1$$

For binary systems ($r = 2$) $0.5 \leq |m| < 1$. After the mantissa is normalized, the exponent is adjusted accordingly.

To facilitate the exchange of data across different architectures, two standard formats are supported by the IEEE. One is the *single-precision* format with 32 bits that allows the representation of values from 1.175×10^{-38} to $3.403 \times 10^{+38}$. The *double-precision* format with 64 bits allows the representation of floating-point numbers between 2.225×10^{-308} to $1.798 \times 10+308$.

2.6.1 Single Precision

One of the standard floating point representations is the 32-bit IEEE single precision organized in three fields as shown in Fig. 2.1. Bit 31 represents the sign bit S, bits 23 to 30, a total of 8 bits represent the exponent E, and bits 0 to 22, for a total of 23 bits, represent the mantissa m. The value of the real number represented is:

$$(1 - 2S) \times 1.m \times 2^E$$

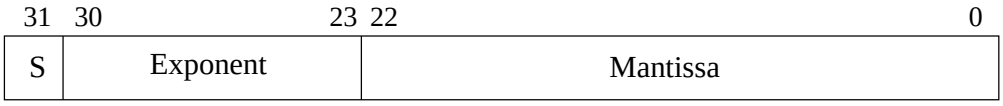

Figure 2.1: IEEE single-precision floating point representation

The exponent is an excess-127 number. This means that $E = e + 127$ where e is the actual *unnormalized* exponent. An excess-127 representation gives a range

of 0 to 255 for the machine representation while the range of the actual exponent values e with 8 bits is -128 to +127 in two's complement notation. The net effect is that the use of excess-127 notation shows all exponents to be positive and there is no need to represent negative exponents. The mantissa is shifted into a normalized form. In the IEEE standard one more shift to the left provides the format $1.m$ where the leading one is referred to as the *hidden bit* because there is no need to encode it. Note that the value $1.m$ is now between 1 and 2.

For example, consider the single-precision IEEE representation for the decimal number 8.1875. The first step is to convert from base 10 to base 2:

$$8.1875 = 1000.0011$$

A second step normalizes the mantissa by left shifting (floating) the point three positions (or shifting zeroes 3 positions to the right). The number of shifts is the shifting factor that is added to the excess number. Shifting the mantissa creates the form $1.m$:

$$1000.0011 = 1.0000011 \times 2^3$$

The last step calculates $E = 3+127 = 130 = 10000010$, and the IEEE floating point representation is given as follows:

S	E	m
0	1000 0010	000001100 . . . 000

Now consider a second example in which the decimal number is 0.1875. The first step converts to base 2:

$$0.1875 = 0.0011$$

The normalization step shifts all leading zeroes to the left by two positions. One more shift creates the form $1.m$ with a shifting factor of 3 that is subtracted from 127:

$$0.0011 = 1.100 \ldots \times 2^{-3}$$

The exponent $E = -3 + 127 = 124 = 01111100$. The final floating point representation is given as follows:

S	E	m
0	01111100	10000 . . . 000

2.6.2 Double Precision

The double-precision standard requires 64 bits organized also in three fields as shown in Fig. 2.2. Bit 63 is the sign bit. The exponent now occupies 11 bits, from bit 52 to 62. The mantissa occupies a total of 52 bits from bit 0 to 51. In two's complement notation the value of the exponent with 11 bits range from 1023 to -1024. Therefore, the excess-1023 notation used by the standard allows the representation of exponents from 0 to 2047. In excess-1023 notation, the exponent $E = e + 1023$. The mantissa is normalized to the standard format $1.m$.

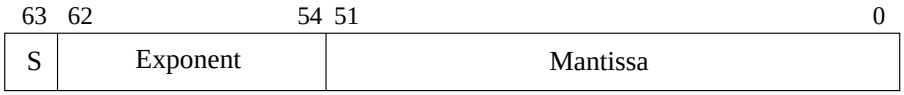

Figure 2.2: IEEE double-precision floating point representation

2.7 Character Representation

The most common binary representation of alphanumeric characters makes use of the ASCII (American Standard Code for Information Interchange) code which assigns a unique numeric value (code) to each alphanumeric character. The original proposed standard consists of 7-bit codes that allows the representation of up to $2^7 = 128$ characters that can be grouped as follows:

Bit 6	Bit 5	Code group	characters
0	0	01h — 1Fh	control characters
0	1	20h — 40h	digits and punctuation characters
1	0	41h — 60h	Upper case and special characters
1	1	61h — 7Fh	Lower case and special characters

Groups are identified by bits 5 and 6 in the code. The first group (00) codes non-printing control characters. For example the ascii code for the *carriage return* is 0Dh, the ascii code for the linefeed is 0Ah, and the code for the backspace key is 08h. The second group (01) includes the code for various punctuation symbols, special characters and numeric digits. The codes for numbers are from 48 (30h) to 57 (39h) and differ only in the lowest order nibble which corresponds to the numeric value represented. Therefore to obtain the ascii code for a number n perform "n" = 30h + n.

The third group (10) includes a subset of codes for 26 upper case alphabetic symbols. The remaining codes are used for special symbols. Likewise the fourth group (11) assigns 26 codes for lower case alphabetic symbols and the remaining codes are assigned to special characters including the last code (127) reserved for the delete control character. Note that bit 5 is the only bit that is different between the codes for upper and lower alphabetic characters.

The *extended* IBM ascii character set consists of 8 bits such that up to $2^8 = 256$ symbols can be represented. This extension is useful for accommodating mathematical, graphical, and foreign symbols.

Ascii tables are accessible at *http://www.LookupTables.com.* For convenience the extended IBM ascii set is reproduced in Appendix B.

2.7.1 The Unicode Standard

In response to the rapid growth of the Internet, web servers, and global service networking, the Unicode standard (http://www.unicode.org/standard/principles.html) is the universal character encoding standard for data representation currently in use. The design of Unicode is derived from the simplicity and consistency of ASCII, but goes far beyond ASCII's limited ability to encode only the Latin alphabet. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique numeric value and name. Codes are assigned to characters used in all the major languages written today; it includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc. It provides codes for diacritics, which are modifying character marks such as the tilde ($\tilde{()}$), that are used in conjunction with base characters to represent accented letters. The Unicode Standard, Version 3.2 provides codes for 95,221 characters from the world's alphabets, ideograph sets, and symbol collections. The original intent was to use a single 16-bit encoding that provides code points for more than 65,000 characters. While 65,000 characters are sufficient for encoding most of the many thousands of characters used in major languages of the world, the Unicode standard now supports three encoding forms that use a common repertoire of characters but allow for encoding as many as a million more characters. This is sufficient for all known character encoding requirements, including full coverage of all historic scripts of the world, as well as common notational systems.

2.8 Exercises

1. Obtain the decimal representation of the following unsigned binary integers:

(a) 1000 1111 0101 1110

(b) 0010 1110

(c) 1100 1011 0101 1011

2. What is the minimum number of binary bits required to represent each of the following unsigned integers?

   (a) 3098

   (b) 512

   (c) 65534

3. Obtain the binary representation of the decimal values in the previous exercise with a number of digits $n = 16$

4. Obtain the hexadecimal representation of the following unsigned binary numbers:

   (a) 0110 1110 1100 0010

   (b) 1111 1010 0011 1101

   (c) 1100 1110 1011 1110

5. Repeat the previous exercise assuming signed binary numbers in two's complement notation. Verify that the hexadecimal notation represents the same value.

6. Obtain the binary representation of the following numbers:

   (a) $(B687A3C1)_{16}$

   (b) $(3112301223231331)_4$

   (c) $(5645231671)_8$

7. Obtain a hexadecimal representation of each of the following signed decimal integers:

   (a) -455

   (b) -62

   (c) +138

8. The following hexadecimal values represent signed integers. Convert to decimal:

(a) F789h

(b) 7123h

(c) 83BAh

9. Obtain the 16-bit binary two's complement representation of the following signed numbers:

(a) -127

(b) -48

(c) +12345

(d) +254.125

(e) -5.25

10. What is the largest value you can represent using a 256-bit unsigned integer?

11. What is the smallest and largest number you can represent with 256 bits in two's complement?