3.2.1.4 Two's Complement Representation

Almost all computers use two's complement representation to store a signed integer in an n-bit memory location. T*wo's complement* numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00\ldots000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01\ldots111_2 = 2^{N-1} - 1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10\ldots000_2 = -2^{N-1}$. And -1 is written as all ones: $11\ldots111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the overall number is interpreted differently for two's complement numbers and sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called *taking the two's complement*. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. Another method is to copy bits from the right until a 1 is copied, then invert the rest of the bits. The two's complement representation is useful to find the representation of a negative number or to determine the magnitude of a negative number.

To store an integer in two's complement representation, the computer follows the step below:

i. The integer is changed to an n-bit binary
ii. If the integer is positive or zero, it is stored as it is; if it is negative, the computer takes the two's complement of the integer and then stores it.

To retrieve an integer in two's complement representation, the computer follows the steps below:

i. If the leftmost bit is 1, the computer applies the two's complement operation to the integer. If the leftmost bit is 0, no operation is applied
ii. The computer changes the integer to decimal

Example: 4

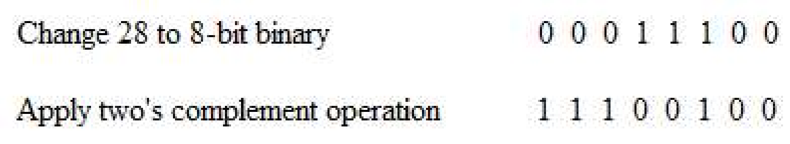
a. Store +28 in an 8-bit memory location using two's complement representation.
b. Store -28 in an 8-bit memory location using two's complement representation.

Solution:

a. The integer is positive, so after decimal to binary conversion, no more action is needed. Note that three extra 0s are added to the left of the integer to make it eight bits.

Change 28 to 8-bit binary	00011100

b. The integer is negative, so after decimal to binary conversion, the computer applies the two's complement operation on the integer.



Change 28 to 8-bit binary	0 0 0 1 1 1 0 0
Apply two's complement operation	1 1 1 0 0 1 0 0

Example 5:

Two's Complement Representation of a Negative Number

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution

Start with $+ 2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

Example 6:

Value of Negative Two's Complement Numbers

Find the decimal value of the two's complement number 1001_2.

Solution

1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding N-bit numbers, the carry out of the Nth bit (i.e., the $N + 1^{\text{th}}$ result bit) is discarded.

Example 7:

Adding Two's Complement Numbers

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.
Solution
(a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.
Subtraction is performed by taking the two's complement of the second number, then adding.

Example 8:

Subtracting Two's Complement Numbers

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.
Solution
(a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.
The two's complement of 0 is found by inverting all the bits (producing $11\ldots111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate −0. Zero is considered positive because its sign bit is 0.
Like unsigned numbers, N-bit two's complement numbers represent one of 2^N possible values. However the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: −8 to 7. In general, the range of an N-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no −0. The most negative number $10\ldots000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01\ldots111_2$) and adding 1, which produces $10\ldots000_2$, the weird number, again. Hence, this negative number has no positive counterpart.
Adding two N-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate

overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 9:

Adding Two's Complement Numbers with Overflow

Compute $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution

$4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and −3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively.

3.2.1.5 Comparison of the three Systems

Table 1 shows a comparison between unsigned, two's complement, and sign-and-magnitude integers. A 4-bit memory location can store an unsigned integer between 0 and 15, and the same location can store two's complement signed integers between -8 and +7.

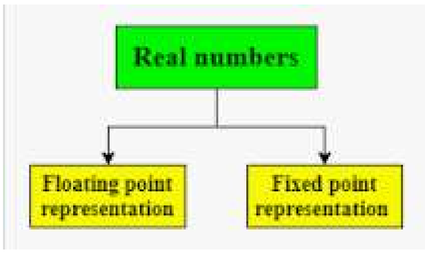
Table 1: Summary of Integer Representation

Contents of Memory	**Unsigned**	**Sign-and-magnitude**	**Two's Complement**
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	8	-0	-8
1001	9	-1	- 7
1010	10	-2	- 6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

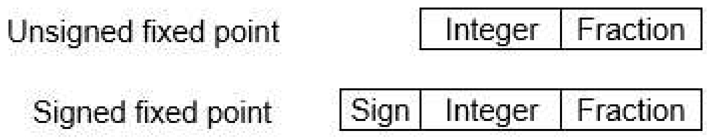
3.2.2 Storing Reals

A real is a number that include fractions/values after the decimal point. For example, 231.54 is a real number. Real numbers are represented either as fixed point number representation or floating-point number representation. In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.



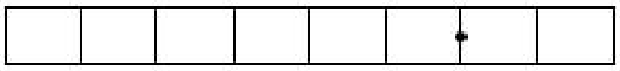
3.2.2.1 Fixed-Point Representation

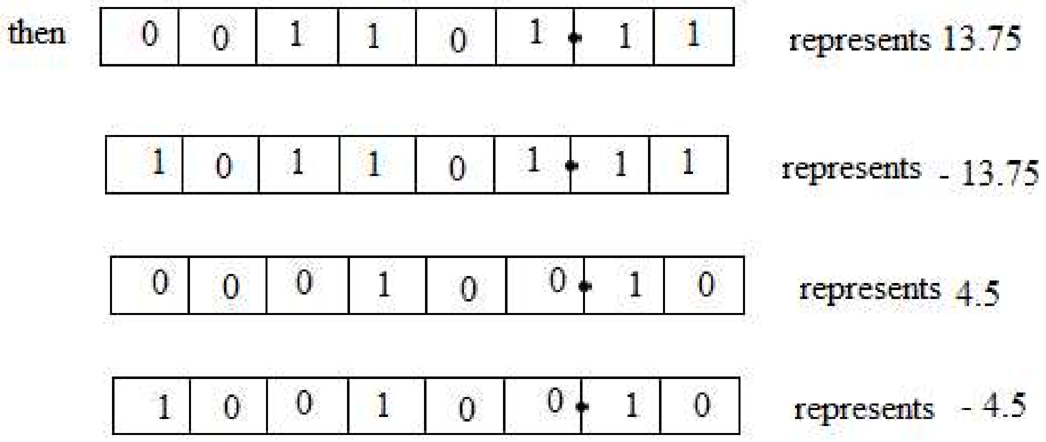
This representation has fixed number of bits for integer part and for fractional part. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



For a fixed-point number representation the programmer requires a computer-storage location of sufficient size to store all the digits of the number.

For an 8-bit representation, if the programmer assumes the point to be:





In using fixed-point number representation, the result may not be accurate or it may not have the required precision. For example in a decimal system, assume a fixed-point representation with two digits at the right of the decimal point and fourteen digits at the left of the decimal point, for a total of sixteen digits is used. The precision of a real number in this system is lost if a decimal number such as 1.00792 is represented; the system stores the number as 1.00. Also, assume a fixed-point number representation with six digits to the right of the decimal point and ten digits to the left of the decimal point for a total of sixteen digits. The accuracy of a real number in this system is lost if a decimal number such as 631254378943.43 is represented; the system stores the number as 1254378943.43. Therefore, real numbers with very large parts or very small fractional parts should not be stored in fixed-point representation.

3.2.2.2 Floating-Point Number Representation

The solution for maintaining accuracy or precision is to use floating-point representation. This representation allows the decimal point to float that is, there can be different numbers of digits to the left or right of the decimal point. The range of real numbers that can be stored using this method increases tremendously. Numbers with large integer parts or small fractional parts can be stored in memory. In floating-point number representation, a number is made of three section:

sign	shifter	Fixed-point number

The first section is the sign, either positive or negative, the second section shows how many places the decimal point should be shifted to the right or left to form the actual number while the third section is a fixed-point representation in which the position of the decimal is fixed.

To make the fixed part of the representation uniform, the floating-point method uses only one none-zero digit to the left of the decimal point. This is called normalisation. In the decimal system, this digit can be 1 to 9, while in the binary system it can only be 1. After a binary number is normalized, only three pieces of information about the number are stored: sign, exponent, and mantissa. The sign can be stored using 1 bit (0 or 1). The exponent (power of 2) defines the shifting of the decimal point. The power can be negative or positive. A new representation call the Excess system is used to store the exponent. In the Excess system, both the positive and negative integers are stored as unsigned integers. To represent a positive or negative, a positive integer called a bias is added to each number to shift them uniformly to the non-negative side. The value of this bias is $2^{m-1} - 1$, where m is the size of the memory location to store the exponent. The mantissa is the binary integer to the right of the decimal point. It defines the precision of the number. The mantissa is stored in fixed-point notation. The mantissa is a fractional part that together with the sign, is treated like an integer stored in sign-and-magnitude representation.

The Institute of Electrical and Electronics Engineers (IEEE) has defined several standards for storing floating-point numbers. The two most common are the single precision and double precision formats. The single precision format uses a total of 32 bits to store a real number in floating-point representation. The 'sign' occupies one bit (0 for positive and 1 for negative), the 'exponent' occupies eight bits using a bias o 127, and the mantissa uses twenty-three bits (unsigned number). This standard is referred to as Excess_127 because the bias is 127. Double precision format uses a total of 64 bits to store a real number in floating-point representation. The sign occupies one bit, the exponent occupies eleven bits using a bias of 1023, and the mantissa uses fifty-two bits. The standard is referred to as Excess_1023 because the bias is 1023.

A real number can be stored in one of the IEEE standard floating-point format using the following procedure:

a. Store the sign as either 0 or 1 (depending on if the number is positive or negative)
b. Change the number to binary if not in binary
c. Normalise the number
d. Find the values of exponent and mantissa
e. Concatenate the sign, exponent and mantissa

Example 10:

Show the Excess_127 representation of the decimal number 5.75

Solution

a. The sign is positive, so sign = 0
b. Decimal to binary conversion: 5.75 = 101.11
c. Normalisation: $101.11 = 1.0111 \times 2^2$

d. Exponent = 2 + 127 = 129 = $(10000001)_2$, mantissa = 0111. Nineteen zeroes are added at the right of the mantissa to make it 23 bits
e. The presentation is shown below

Number	Normalised Value	Stored Value		
		sign	Exponent	Mantissa
5.75 = $(101.11)_2$	1.0111 x 2^2	0	10000001	01110000000000000000000

The number is stored as 01000000101110000000000000000000

Example 11:

Show the Excess_127 representation of the decimal number -161.875

Solution

a. The sign is negative, so sign = 1
b. Decimal to binary conversion: 161.875 = 10100001.111
c. Normalisation: 10100001.111= 1.0100001111 x 2^7
d. Exponent = 7 + 127 = 134 = $(10000110)_2$, mantissa = 0100001111. Thirteen zeroes are added at the right of the mantissa to make it 23 bits
e. The presentation is shown below

Number	Normalised Value	Stored Value		
		sign	Exponent	Mantissa
161.875 = 10100001.111	1.0100001111 x 2^7	1	10000110	01000011110000000000000

The number is stored as 11000011001000011110000000000000

Example 12:

The bit pattern $(11001010000000000111000100001111)_2$ is stored in memory in Excess_127 format. Show what the value of the number is in decimal notation.

Solution

a. The first bit represent the sign, the next eight bits represents the exponent, and the remaining 23 bits represents the mantissa;
b. The first bit represent the sign, the next eight bits represents the exponent, and the remaining 23 bits represents the mantissa;
c. The sign is negative;
d. The exponent $(10010100)_2 \equiv 148_{10}$
e. Therefore, the shifter = 148 – 127 = 21
f. Denormalisation gives $1.00000000111000100001111 \times 2^{21}$
g. The binary number is 1000000001110001000011.11
h. Conversion from binary to decimal: 1000000001110001000011.11 = 2,104,378.75 (absolute value)
i. The number is -2,104,378.75

3.3 STORING TEXT

Character data, sometimes referred to as “string” data, may consist of any digits, letters of the alphabet or symbols which, the internal coding system of the computer is capable of representing. Any sequence of symbols used to represent an idea is therefore referred to as text data type. Different sets of bit patterns have been designed to represent text symbols. Each set is called a code, and the process of representing symbols is called coding. Some popular codes are:

1. ASCII

The American National Standards Institute (ANSI) developed a code called American Standard Code for Information Interchange (ASCII). This code uses seven bit for each symbol. This means that 2^7 = 128 different symbols can be defined in this code. ASCII codes are widely used throughout the computer industry.

2. EBCDIC

Extended Binary Coded Decimal Interchange Code (EBCDIC) is sometimes called 8-bit ASCII. There are 256 characters in the EBCDIC character set.

3. Unicode