- *Hexadecimal*: In the BCD encoding, some bit sequences of length 4 do not correspond to any decimal digit. Therefore not every binary sequence can be interpreted as a valid BCD encoding. For example, 1100 0011 does not represent any valid BCD encoded number. By assigning unique symbols to those sequences not representing a decimal digit, it is possible to encode an alphabet of 16 symbols. Such an alphabet is called an hexadecimal alphabet. The symbols assigned to the binary sequences in the BCD encoding are as follows:

  1010 is represented by A
  1011 is represented by B
  1100 is represented by C
  1101 is represented by D
  1110 is represented by E
  1111 is represented by F

  The hexadecimal alphabet is most often used as a "short-hand" way of expressing long binary sequences. Unlike converting between decimal and binary, it is very easy to convert between binary and hexadecimal:

  Binary to Hexadecimal : For any arbitrary binary sequence, group the bits in sets of 4 beginning from the right. Pad the **left** of the binary sequence with 0's if there are less than 4 bits in the leftmost group. Now, replace each set of 4 bits by its corresponding hexadecimal symbol.

  Hexadecimal to Binary : Replace each hexadecimal symbol by its corresponding 4-bit binary codeword.

  For example, the sequence 101 0010 1010 can be expressed in hexadecimal as 52A.

- *ANSI*: Also known as ASCII, this encoding scheme assigns binary codewords to each symbol that can be entered on a keyboard. Originally the alphabet consisted of fewer than 127 characters and so a 7-bit binary sequence was sufficient to define a unique codeword for each character.

  With the inclusion of non-English characters, the ANSI encoding scheme was expanded to 8-bit codewords to accommodate up to 256 different characters.

- *UNICODE*: To provide an encoding scheme that can represent uniquely the alphabets of any language as well as the many fonts, successive extensions of the UNICODE encoding system have been introduced. This system uses much longer codewords because of the much larger alphabet that must be accommodated. The latest standard, Unicode 8.0 can accommodate over 120,000 distinct symbols.

2.2 Minimum Length Fixed Encodings

The BCD and Hexadecimal encodings are examples of fixed length encodings. Given an alphabet of m symbols, the minimum length, n, of each codeword in a fixed length encoding satisfies:

$$2^{n-1} < m \leq 2^n$$

This is equivalent to :

$$n - 1 < log_2 \, m \leq n$$

$$\text{or } n = \lceil log_2\ m \rceil$$

Thus, for an alphabet of 10 symbols such as the digits, $n = 4$. Hence the BCD and HEXADECIMAL codes are different minimal fixed length encodings requiring only 4-bit codewords. While this is the minimum, longer codewords may be defined for a given alphabet.

3 SIGNED INTEGERS

Given a fixed length encoding system with k-bit codewords, 2^k distinct codewords are available. Therefore it is necessary to identify the range of integers that will be defined by any encoding of signed integers, since not all integers will be represented. A suitable finite set of integers to be encoded will satisfy the following:

- the number 0 will be represented.
- For every positive number, its negative counterpart will be represented.
- The integers should be consecutive.

With these constraints, the integers from $-(2^{k-1}-1)$ to $+2^{k-1}-1$ can be represented with $2^k - 1$ codewords, leaving one codeword initially "unassigned".

To design an encoding of this set of $2^k - 1$ symbols, it is necessary to assign a unique k-bit codeword to each symbol (i.e., integer). There are many ways to do this and all will result in one codeword not being used. Two such scenes are describe here.

3.1 Sign-Magnitude Encoding

With *sign-magnitude* encoding system, the most significant bit of the sequence specifies the sign: 0 if positive, 1 if negative. The remaining bits determine the magnitude of the integer. For k = 4 the following table defines the encoding:

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

The sign-magnitude binary representation of $-Y$ can be obtained by evaluating:

$$-Y = (Y + 2^{k-1})$$

expressed in natural binary, where Y denotes the magnitude of the integer.

More simply, to convert from decimal to sign-magnitude binary, express the magnitude as an $k-1$ bit binary number. Then assign the appropriate sign bit as the most significant bit.

To convert a sign-magnitude encoding to decimal, remove the most significant bit, and then convert the remaining $k-1$ bits to decimal. Set the sign according to the value of the most significant bit: 0 = "+"; 1 = "−".

3.2 2's Complement Encoding

The "complement" of a binary sequence is obtained by changing all 1's to 0's and all 0's to 1's. Thus, for example, the complement of 01101 is 10010.

For non-negative integers, the 2's complement encoding system is similar to sign-magnitude encoding. All non-negative numbers are represented by their k-bit natural binary encodings, so all such codewords for the values from 0 to $2^{k-1}-1$ will have 0 in the most significant bit position.

The negative integer encodings, however, are obtained by complementing the base-2 encoding of the magnitude and adding 1. The following table shows the 4-bit codewords for the integers from −7 to 7:

codeword	integer
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	–
1001	−7
1010	−6
1011	−5
1100	−4
1101	−3
1110	−2
1111	−1

The 2's complement binary representation of $-Y$ can be obtained by evaluating:

$$-Y = \overline{Y_{base-2}} + 1$$

where the "overline" denotes the complement of the base-2 representation of Y.

To convert from integer to 2's complement encoding, if the number is positive, use the k-bit base-2 representation. If it is negative, then first obtain the base-2 representation, then complement that binary sequence, and finally add 1.

To determine the integer represented by a 2's complement binary sequence, first examine the most significant bit. If it is 0 then convert the number from base 2 to decimal. If the bit is 1, then first "complement" the binary sequence, then add 1, and finally convert the resulting sequence from base 2 to decimal, and add the "$-$" sign to the resulting decimal integer.

One advantage of the 2's complement encoding is that it results in a unique codeword for representing the number 0, whereas the sing-magnitude and 1's complement encodings have codewords for both "0" and "-0". However, 2's complement encoding results in a codeword for -2^{n-1} without a corresponding codeword for 2^{n-1}.

A second advantage in representing integers using 2's complement comes from observing that:

$$X - Y = X \text{ plus } (-Y) = X \text{ plus } (\overline{Y} \text{ plus } 1)$$

Thus if signed integers are represented using 2's complement encoding, then subtraction can be performed using addition. Otherwise separate adder and subtracter circuits will be required to provide both arithmetic functions.

3.3 Bias Encoding

Given that integers are to be encoded with binary sequences of length n, the "n-bit bias" is the integer $2^{n-1} - 1$. To encode an integer in the interval from $-(2^{n-1} - 1)$ to $2^{n-1} - 1$, add the n-bit bias to the value to obtain an unsigned integer and express the result in natural binary encoding.

For $n = 4$, the 4-bit bias is 7. The range of values is -7 to 7. The table of encodings is:

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

Note that, except for 0, the sign field of a bias encoding is set to 1 if the number is positive. This is in contrast to the other integer encoding systems described previously.

4 REPRESENTATION OF FRACTIONS

Externded Placeholder Notation :

Values can be represented symbolically as a sequence of characters, called "placeholder notation." The significance of each character is determined by its position in the sequence and by the number, m, of characters in the alphabet. The value represented by the sequence:

$$b_{n-1}b_{n-2}\ldots b_1b_0.b_{-1}b_{-2}\ldots b_{-k}$$

is obtained by evaluating the expression:

$$m^{n-1}\cdot b_{n-1}+m^{n-2}\cdot b_{n-2}+\cdots m^1\cdot b_1+m^0\cdot b_0+m^{-1}\cdot b_{-1}+m^{-2}\cdot b_{-2}+\cdots m^{-k}\cdot b_{-k}$$

The "period" between b_0 and b_{-1} is called a "radix point" and the value m is called the "base." Symbols to the left of the radix point define a value greater than or equal to zero and those to the right define a value less than 1. The sum of the two values defines the number.

For $m = 10$, the decimal system is defined and the radix point is more commonly referred to as a "decimal point." For $m = 2$, the binary system is defined and the radix point is called a "binary point." For $m = 16$, the hexadecimal system is defined.

These three systems are the most commonly used ones in digital systems design and analysis. Conversion of values from one base to another is a common requirement.

Scientific Notation :

A second way to represent numeric values is by expressing the value in the form:

$$0.N\times m^E$$

The value of N represents a fractional value in placeholder notation, called the "significand." The value m is the base and the value E is a signed integer called the "exponent." Scientific notation is useful for expressing very large and very small numbers where placeholder notation would require very long sequences.

4.1 Floating Point

Floating Point is an encoding of numbers that is based on representing values using "scientific notation." Given a decimal number in scientific notation, say $.N\times 10^{\ E}$, the significant digits, N, can be expressed by a pair of integers: the *significand*, N, and the *exponent*, E. (also an integer) determining the position of the decimal point. Thus a number expressed in scientific notation is defined by a pair of integers, E and M.

This representation results in multiple ways of expressing the same number, since scientific notation does so. For example:

$$\begin{aligned}255 &= .255\times 10^3 && (1)\\ &= .0255\times 10^4 && (2)\\ &= .00255\times 10^5 && (3)\end{aligned}$$

etc.

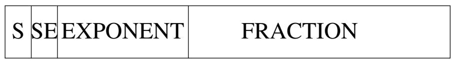
Therefore, by convention, one representation is chosen, called *normalized form.* This form assumes the most significant digit is non-zero. Thus only (1) among the examples above satisfies the definition, and (1) is said to be a "normalized decimal representation of 255."

In digital systems, base 2 rather than base 10 is used. A normalized base 2 floating point number has the form $1.bbbbb \times 2^{ddd}$ where $1.bbbbb$ is the base 2 significand, and ddd is the base 2 exponent. Thus 63, expressed as a normalized binary number, is given by:

$$63_{10} = 111111_2 = 1.11111 \times 2^5$$

Because a normalized binary number has only one possible value for the most significant bit, namely "1", that bit is never included in the encoding of the number into floating point. Instead, when a binary sequence is to be decoded into its corresponding decimal representation a '1' bit is placed in the most significant bit position of any bit string that represents the "fraction" field of the encoding. The only exception is the representation of the number "0" which, by convention, is represented by a bit string of all zeros.

The encoding of a decimal number expressed in scientific format into a floating point codeword requires that the word be partitioned into three fields to hold the fraction part of the significand, the exponent, and their respective signs. The typical format of a floating point word is:



S denotes the sign of the significand, SE the sign of the exponent.

NOTE: Because of the implied "1" to the right of the binary point, the sign of the significand is placed at the most significant bit position of the entire word rather than at the most significant position of the fraction

In digital systems, the maximum number of significant bits (the size of the significand) is constrained by the size of the word and the size of the exponent. For example, if the word-size is 12 bits, then 4 bits might be allocated for the exponent field and its sign, and 8 bits for the significand and its sign. Finally, the fraction part of the significant is "left-justified" in its field. One possible floating point representation of 63 from the example above is:

0	0 1 0 1	1 1 1 1 1 0 0

NOTE: Because the most significant bit of the significand is always "1", this bit is not included in the representation. That is why there are only five "1's" in the fraction field rather than six.

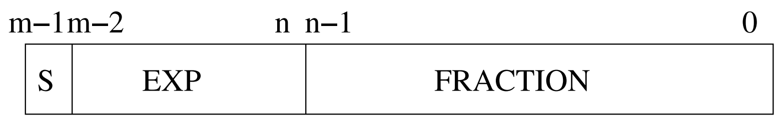
In this example, sign-magnitude encoding was used to represent both the significand and the exponent. However, when performing computations with floating point numbers in CPU's, exponents are represented using "bias" encoding.

Therefore, the representation of 63 as a floating point number using bias encoding of the exponent is:

0	1 1 0 0	1 1 1 1 1 0 0

Encoding Normalized FP Representations:

For any given fixed length encoding system with codewords of length, m, a floating point encoding that provides for n bits for the fraction field can be denoted by FP(m,n). Such an encoding will have the following format:



The following examples illustrate encoding and decoding floating point codewords:

1. If m = 8, n = 3, encode the decimal value: -13

$$\begin{aligned} -13_{10} &= -1101_2 \\ &= -1.101 \times 2^3 \text{ (normalized)} \\ \text{S} &= 1 \text{ (negative)} \\ \text{EXP} &= 0011 + 0111 \text{ (bias is} 2^{4-1} - 1 = 7 = 0111) \\ &= 1010 \\ \text{FRAC} &= 101 \end{aligned}$$

   Therefore the FP(8,3) floating point codeword for -13 is 11010101.

2. If m = 16, n = 8, encode the decimal value: 13.75

$$\begin{aligned} 13.75_{10} &= 1101.11_2 \\ &= 1.10111 \times 2^3 \text{ (normalized)} \\ \text{S} &= 0 \text{ (positive)} \\ \text{EXP} &= 0000011 + 0111111 \text{ (bias is} 2^{7-1} - 1 = 63 = 0111111) \\ &= 1000010 \\ \text{FRAC} &= 10111 \end{aligned}$$

   Therefore the FP(16,8) floating point codeword for 13.75 is 0100001010111000

3. If m = 8, n = 3, 0100 0111 is the codeword for what FP number?

$$\begin{aligned} \text{S} &= 0 \text{ (positive)} \\ \text{EXP} &= 1000 - 0111 \text{ (bias is} 2^{4-1} - 1 = 7 = 0111) \\ &= 0001 \\ &= 1 \\ \text{FRAC} &= 111 \\ \text{SIGNIFICAND} &= 1.111 \end{aligned}$$

   Therefore the number is $1.111 \times 2^1 = 11.11_2 = 3.75_{10}$.

Special Cases:

- **zero**: The value 0 is represented by the m-bit codeword 00...0
- **Denormalized Numbers**: Codewords with an exponent field of all 0's are called "denormalized." There are used to represent very small values. The exponent is given by $1-$ bias and the significant consists of a 0 to the left of the binary point .
- $+$ **infinity**: A codeword with S = 0 and an exponent field of all 1's and a fraction field of all 0's.
- $-$ **infinity**: A codeword with S = 1 and an exponent field of all 1's and a fraction field of all 0's.
- **NaN (Not a Number)**: Any binary sequence with an exponent field of all 1's and a fraction field that is NOT all 0's.