Number Representation and Arithmetic in Various Numeral Systems

Computer Organization and Assembly Language Programming 203.8002

A numeral system is a collection of symbols used to represent small numbers, together with a system of rules for representing larger numbers. Each numeral system uses a set of digits. The number of various unique digits, including zero, that a numeral system uses to represent numbers is called base or radix.

Base - b numeral system
b basic symbols (or digits) corresponding to natural numbers between 0 and b − 1 are used in the representation of numbers.
To generate the rest of the numerals, the position of the symbol in the figure is used. The symbol in the last position has its own value, and as it moves to the left its value is multiplied by b.
We write a number in the numeral system of base b by expressing it in the form

$$N_{(b)} = a_n a_{n-1} a_{n-2} \dots a_1 a_0 a_{-1} a_{-2} \dots a_{-m}$$

N(b), with n+1 digit for integer and m digits for fractional part, represents the sum:

$$N_{(b)} = a_n.b^n + a_{n-1}.b^{n-1} + a_{n-2}.b^{n-2} + \dots + a_1.b^1 + a_0.b^0 + a_{-1}.b^{-1} + a_{-2}.b^{-2} + \dots + a_{-m}.b^{-m}$$

or

$$N_{(b)} = \sum_{i=-m}^{n} a_i.b^i$$

in the decimal system.

Decimal, Binary, Octal and Hexadecimal are common used numeral system. The decimal system has ten as its base. It is the most widely used numeral system, because humans have four fingers and a thumb on each hand, giving total of ten digit over both hand.

Modern computers use transistors that represent two states with either high or low voltages. Binary digits are arranged in groups to aid in processing, and to make the binary numbers shorter and more manageable for humans. Thus base 16 (hexadecimal) is commonly used as shorthand. Base 8 (octal) has also been used for this purpose.

Decimal System

Decimal notation is the writing of numbers in the base-ten numeral system, which uses various symbols (called digits) for no more than ten distinct values (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to represent any number, no matter how large. These digits are often used with a decimal separator which indicates the start of a fractional part, and with one of the sign symbols + (positive) or − (negative) in front of the numerals to indicate sign.

Decimal system is a place-value system. This means that the place or location where you put a numeral determines its corresponding numerical value. A two in the one's place means two times one or two. A two in the one-thousand's place means two times one thousand or two thousand.

The place values increase from right to left. The first place just before the decimal point is the one's place, the second place or next place to the left is the ten's place, the third place is the hundred's place, and so on.

The place-value of the place immediately to the left of the "decimal" point is one in all place-value number systems. The place-value of any place to the left of the one's place is a whole number computed from a product (multiplication) in which the base of the number system is repeated as a factor one less number of times than the position of the place.

For example, 5246 can be expressed like in the following expressions

$$\begin{aligned} 5246 &= 5\times10^3 + 2\times10^2 + 4\times10^1 + 6\times10^0 \\ &= 5\times1000 + 2\times100 + 4\times10 + 6\times1 \end{aligned}$$

Binary System

The binary number system is base 2 and therefore requires only two digits, 0 and 1. The binary system is useful for computer programmers, because it can be used to represent the digital on/off method in which computer chips and memory work.
A binary number can be represented by any sequence of bits (binary digits), which in turn may be represented by any mechanism capable of being in two mutually exclusive states.

Counting in binary is similar to counting in any other number system. Beginning with a single digit, counting proceeds through each symbol, in increasing order. Decimal counting uses the symbols 0 through 9, while binary only uses the symbols 0 and 1. When the symbols for the first digit are exhausted, the next-higher digit (to the left) is incremented, and counting starts over at 0. A single bit can represent one of two values, 0 or 1.Binary numbers are convertible to decimal numbers.

Example:

$$\begin{aligned}(100101)_2 &= 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= 37_{10}\end{aligned}$$

Hexadecimal System

The hexadecimal system is base 16. Therefore, it requires 16 digits. The digits 0 through 9 are used, along with the letters A through F, which represent the decimal values 10 through 15. Here is an example of a hexadecimal number and its decimal equivalent:

$$34F5C_{(16)} = 3\times16^{4} + 4\times16^{3} + 15\times16^{2} + 5\times16^{1} + 12\times16^{0} = 216294_{(10)}$$

The hexadecimal system (often called the hex system) is useful in computer work because it is based on powers of 2. Each digit in the hex system is equivalent to a four-digit binary number.
Table below shows some hex/decimal/binary equivalents.

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	10000
F0	240	11110000
FF	255	11111111

Octal System

Binary is also easily converted to the octal numeral system, since octal uses a radix of 8, which is a power of two (namely, 2^3, so it takes exactly three binary digits to represent an octal digit). The correspondence between octal and binary numerals is the same as for the first eight digits of hexadecimal in the table above. Binary 000 is equivalent to the octal digit 0, binary 111 is equivalent to octal 7, and so forth. Converting from octal to binary proceeds in the same fashion as it does for hexadecimal:

$$65_{(8)} = 110\,101_2$$

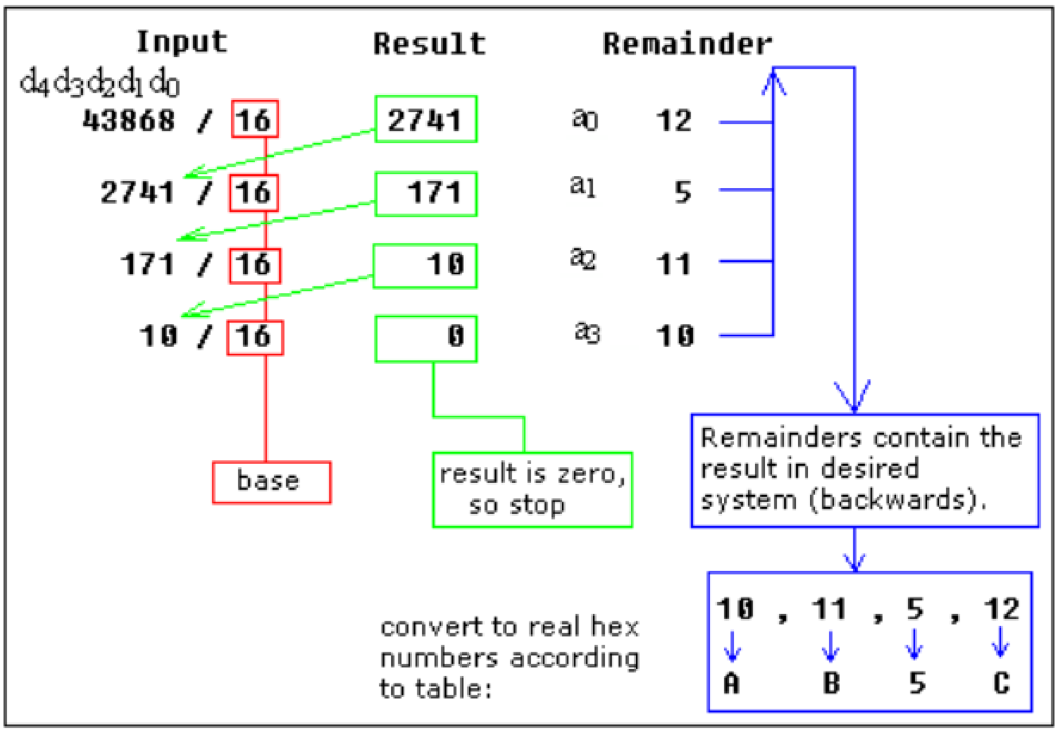
$$17_{(8)} = 001\;111_2$$

And from octal to decimal:

$$235.64_{(8)} = 2\times8^{2} + 3\times8^{1} + 5\times8^{0} + 6\times8^{-1} + 4\times8^{-2} = 157.8125_{(10)}$$

Converting from decimal to base–b

To convert a decimal fraction to another base, say base b, you split it into an integer and a fractional part. Then divide the integer by b repeatedly to get each digit as a remainder. Namely, with value of integer part = $d_{n-1}d_{n-2}...d_2d_1d_{0(10)}$, first divide value by b the remainder is the least significant digit a_0 . Divide the result by b, the remainder is a_1 .Continue this process until the result is zero, giving the most significant digit, a_{n-1} . Let's convert $43868_{(10)}$ to hexadecimal:



Unsigned Integers
Unsigned integers are represented by a fixed number of bits (typically 8, 16, 32, and/or 64)

- With 8 bits, 0…255 can be represented;
- With 16 bits, 0…65535 can be represented;
- In general, an unsigned integer containing n bits can have a value between 0 and 2^n-1

If an operation on bytes has a result outside this range, it will cause an 'overflow'

Signed Integers
The binary representation discussed above is a standard code for storing unsigned integer numbers. However, most computer applications use signed integers as well; i.e. the integers that may be either positive or negative.

There are three common methods to represent signed numbers. In all three methods, positive numbers are represented in the same way, whereas negative numbers are represented differently. Moreover, in all three methods, the most significant bit indicates whether the number is positive (msb=0) or negative (msb=1).

Sign-Magnitude
The most significant bit is the sign (0 – positive, 1 – negative). The magnitude (which is the absolute value of the number) is stored in the remaining bits, as an unsigned value.

For example, an 8-bit signed magnitude representation of decimal 13 is 00001101 while −13 is 10001101.

Note that using 8-bit signed magnitude, one can represent integers in the range −127 (11111111) to +127 (01111111).

Sign-magnitude has one peculiarity, however. The integer 0 can be represented in two ways: 00000000 = +0 and 10000000 = −0.

One's Complement
In this method, positive numbers are represented the same way as in sign-magnitude. Given a positive number (msb=0), we get the corresponding negative number by complementing each bit (1 becomes 0, and 0 becomes 1). This works also in the other direction (from negative to positive).

For example, an 8-bit one's complement representation of decimal 13 is 00001101 while −13 is 11110010.

Using 8-bit one's complement, we can represent integers in the range −127 (10000000) to +127 (01111111).

Like sign-magnitude, the number 0 has two representations: 00000000 = +0 and 11111111 = −0.

Two's Complement
This is the most common representation scheme. Positive numbers are represented the same way as in the other methods. Given a positive number (msb=0), the corresponding negative number is obtained by the following steps:

- Complement each bit (getting in fact a 1’s complement number)
- Add one to the complemented number.

This works also in the other direction (from negative to positive).

Example: +4210 = 00101010_2 , -4210 = 11010110_2

Example : Performing two's complement on the decimal 42 to get -42 using 8-bit representation:

```
42= 00101010   Convert to binary

    11010101   Complement the bits

    11010101   Add 1 to the complement
  + 00000001
    --------
    11010110   Result is -42 in two's complement
```

Notice that in two’s complement, there is a single representation for 0.

The range of numbers that can be represented in two’s complement is asymmetric. For example, using 8-bit two’s complement, we can represent integers in the range −128 (10000000) to +127 (01111111). If you try to convert -128 to the corresponding positive number, you end up getting -128 again. In other words, the lowest negative number has no complement.

Arithmetic Operations on Integers

Addition and Subtraction of integers

Binary Addition is much like normal everyday (decimal) addition, except that it carries on a value 2 instead of value 10.
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0, and carry 1 to the next more significant bit

Adding two's-complement numbers requires no special processing if the operands have opposite signs: the sign of the result is determined automatically. For example, adding 15 and -5:

```
 11111 111   (carry)
  0000 1111  (15)
+ 1111 1011  (-5)
=================
  0000 1010  (10)
```

This process depends upon restricting to 8 bits of precision; a carry to the (nonexistent) 9th most significant bit is ignored, resulting in the arithmetically correct result of 10_{10}

The last two bits of the carry row (reading right-to-left) contain vital information: whether the calculation resulted in an overflow, a number too large for the binary system to represent (in this case greater than 8 bits). An overflow condition exists when a carry (an extra 1) is generated out of the far left bit (the MSB), but not into the MSB. As mentioned above, the sign of the number is encoded in the MSB of the result.

In other terms, if the left two carry bits (the ones on the far left of the top row in these examples) are both 1's or both 0's, the result is valid; if the left two carry bits are "1 0" or "0 1", a sign overflow has occurred.

As an example, consider the 4-bit addition of 7 and 3:

```
 0111   (carry)
  0111  (7)
+ 0011  (3)
=============
  1010  (−6)  invalid!
```

In this case, the far left two (MSB) carry bits are "01", which means there was a two's-complement addition overflow. That is, $1010_2 = 10_{10}$ is outside the permitted range of −8 to 7.

In general, any two *n*-bit numbers may be added *without* overflow, by first sign-extending both of them to *n*+1 bits, and then adding as above. The *n*+1 bit result is large enough to represent any possible sum (*e.g.*, 5 bits can represent values in the range −16 to 15) so overflow will never occur. It is then possible, if desired, to 'truncate' the result back to *n* bits while preserving the value if and only if the discarded bit is a proper sign extension of the retained result bits. This provides another method of detecting overflow.

Like addition, the advantage of using two's complement is the elimination of examining the signs of the operands to determine if addition or subtraction is needed. For example, subtracting −5 from 15 is really adding 5 to 15, but this is hidden by the two's-complement representation:

```
 11110 000   (borrow)
  0000 1111  (15)
- 1111 1011  (-5)
===========
  0001 0100  (20)
```

Another example is a subtraction operation where the result is negative: 15 − 35 = −20:

```
 11100 0000  (borrow)
  0000 1111  (15)
- 0010 0011  (35)
===========
  1110 1100  (-20)
```

Overflow is detected the same way as for addition, by examining the two leftmost (most significant) bits of the borrows; overflow has occurred if they are different.

Multiplication and Division of Integers

Binary Multiplication

Multiplication in the binary system works the same way as in the decimal system:
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1, and no carry or borrow bits
Example:

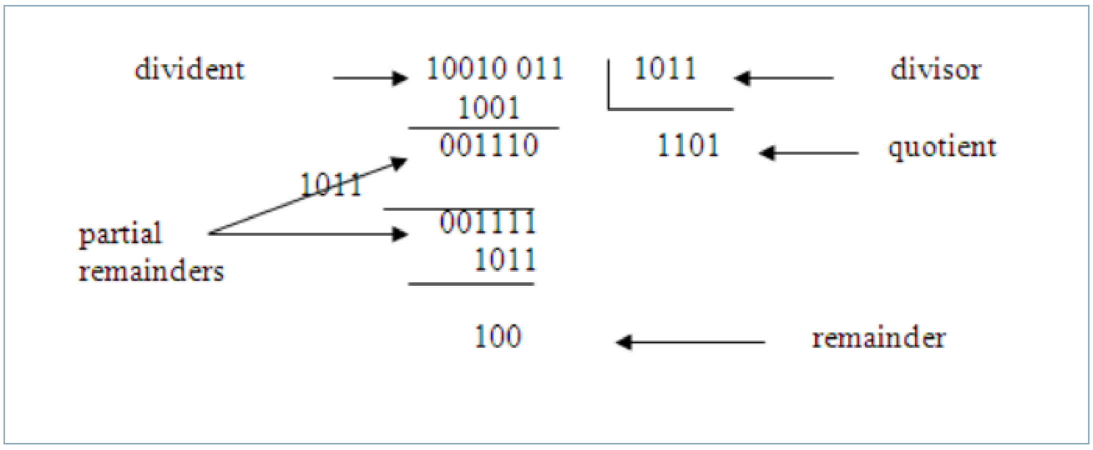
```
00010111 × 00000011 = 01000101

       0   0   0   1   0   1   1   1       =    23(base 10)
×      0   0   0   0   0   0   1   1       =    3(base 10)
       ----------------------
               1   1   1   1   1                        carries
           0   0   1   0   1   1   1
       0   0   1   0   1   1   1
   0   0   1   0   0   0   1   0   1       =    69(base 10)
```

Another Example:

```
    1000
  ×
    1011
    ----
    1000
   1000
  0000
 1000
 1011000
```

Binary division

Follow the same rules as in decimal division.


divident
10010 011
1011
divisor
1001
001110
1101
quotient
1011
partial
remainders
001111
1011
100
remainder