

# CPE 323 Data Types and Number Representations

Aleksandar Milenkovic

## Numerical Systems: Decimal, binary, hexadecimal, and octal

We ordinarily represent numbers using decimal numeral system that has 10 as its base (also base-10 or denary). The decimal numeral system uses 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. It is the most widely used numeral system, perhaps because humans have ten fingers over both hands.

For example, a decimal number 456 could also be represented as (4 100's, 5 10's, and 6 1's):

$$456_{10} = 4*10^2 + 5*10^1 + 6*10^0$$

The binary numeral system, or base-2 (radix-2) number system, is a numeral system that represents numeric values using only two symbols, 0 and 1. These symbols can be directly implemented using logic gates and that is why all modern computers internally use the binary system to store information.

For example,  $10111_2$  is a binary number with 5 binary digits. The left-most bit is known as the most significant bit (msb), and the rightmost one is known as the least-significant bit (lsb). This binary number  $10111_2$  can be converted to a corresponding decimal number as follows:

$$10101_2 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 23_{10}$$

Hexadecimal (base-16, hexa, or hex) is a numeral system with a radix, or base, of 16. It uses sixteen distinct symbols, most often the symbols 0–9 to represent values zero to nine, and A, B, C, D, E, F (or a through f) to represent values ten to fifteen.

For example, a hex number  $1A3_{16}$  corresponds to

$$1A3_{16} = 1*16^2 + 10*16^1 + 3*16^0 = 256 + 160 + 3 = 419_{10}$$
$$1A3_{16} = 1\_1010\_0011_2$$

Its primary use is as a human friendly representation of binary coded values, so it is often used in digital electronics and computer engineering. Since each hexadecimal digit represents four binary digits (bits), it is a compact and easily translated shorthand to express values in base two.

Octal (also base-8) is a number system with a radix, or base, of 8. It uses 8 distinct symbols 0, 1, 2, 3, 4, 5, 6, and 7. An octal number  $372_8$  corresponds to

$$372_8 = 3 \cdot 8^2 + 7 \cdot 8^1 + 2 \cdot 8^0 = 192 + 56 + 2 = 250_{10}.$$

## Storing and interpreting information in modern computers

An n-bit binary number can differentiate between  $2^n$  things. In digital computers information and space are organized in bytes. A byte can represent up to  $2^8 = 256$  things.

1 byte = 8 binary digits = 8 bits (e.g. 10010011)

$\frac{1}{2}$  byte = 1 nibble = 4 bits

In 16-bit computers 1 word = 2 bytes (16 bits).

In 32-bit computers 1 word = 4 bytes (32 bits), a half-word is 2 bytes (16-bits), and a double word is 8 bytes (64 bits).

Meaning of bits and bytes is assigned by the convention.

Some examples of common encoding formats are as follows:

1 byte ASCII = one of 256 alphanumeric or special purpose text character (definitions are in the ASCII table, see <http://en.wikipedia.org/wiki/ASCII>)

1 byte = 1 short unsigned integer (0 – 255)

1 byte = 1 short signed integer (-128 – 127)

1 byte = 2 Binary coded decimal (BCD) digits (i.e. one per nibble)

1 byte = 2 hexadecimal (hex) digits (one per nibble)

1 byte = 8 individual bits (e.g. 8 status flags)

2 bytes = 1 unsigned integer (0 – 65535)

2 bytes = 1 signed integer (-32768 – 32767)

...

## Conversion to binary from other numeral systems

To convert from a base-10 integer numeral to its base-2 (binary) equivalent, the number is divided by two, and the remainder is the least-significant bit. The (integer) result is again divided by two, its remainder is the next most significant bit. This process repeats until the result of further division becomes zero.

For example,  $23_{10}$ , in binary, is:

Operation Remainder

$$23 \div 2 = 11 \text{ } 1$$

$$11 \div 2 = 5 \text{ } 1$$

$$5 \div 2 = 2 \text{ } 1$$

$$2 \div 2 = 1 \text{ } 0$$

$$1 \div 2 = 0 \text{ } 1$$

Reading the sequence of remainders from the bottom up gives the binary numeral  $10111_2$ .

This method works for conversion from any base, but there are better methods for bases which are powers of two, such as octal and hexadecimal given below.

To convert a binary number to a decimal number use the following approach:

$$10101_2 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = (((1*2 + 0)*2 + 1)*2 + 1)*2 + 1.$$

Octal and hexadecimal numbers are easily converted to binary numbers and vice versa. For example:

$$372_8 = 001\_111\_010_2 = 7A_{16}$$

$$1011\_0101_2 = 265_8 = B5_{16}.$$

## Integers

The term integer is used in computer engineering/science to refer to a data type which represents some finite subset of the mathematical integers. Integers may be unsigned (capable of representing only non-negative integers) or signed (capable of representing negative integers as well).

An  $n$ -bit unsigned integer can encode  $2^n$  numbers that represent the non-negative values 0 through  $2^n - 1$ .

There are three different ways to represent negative numbers in a binary numeral system. The most common is **two's complement**, which allows a signed integral type with  $n$  bits to represent numbers from  $-2^{(n-1)}$  through  $2^{(n-1)} - 1$ . Two's complement arithmetic is convenient because there is a perfect one-to-one correspondence between representations and values, and because addition, subtraction and multiplication do not need to distinguish between signed and unsigned types. The other possibilities are sign-magnitude and ones' complement.

*Sign and Magnitude* representation uses  $n-1$  bits to convey magnitude with “most significant bit” or MSB used for sign (0 = +, 1 = -). Note: the problem is that there exists two representations for value 0.

With one's complement representation a negative binary number is created by applying the bitwise NOT to its positive counterpart. Like sign-and-magnitude representation, ones' complement has two representations of 0: 00000000 (+0) and 11111111 (-0).

As an example, the ones' complement form of 00101011 (43) becomes 11010100 (-43). The range of signed numbers using ones' complement in a conventional eight-bit byte is  $-127_{10}$  to  $+127_{10}$ .

We focus on two's complement because it is the dominant way to represent signed integers today. Table 1 shows common integer data types and their ranges.

Table 1. Common integer data types and ranges.

Bits	Name	Range
8	byte, octet	<i>Signed:</i> -128 to +127
		<i>Unsigned:</i> 0 to +255
16	halfword, word	<i>Signed:</i> -32,768 to +32,767
		<i>Unsigned:</i> 0 to +65,535
32	word, doubleword, longword	<i>Signed:</i> -2,147,483,648 to +2,147,483,647
		<i>Unsigned:</i> 0 to +4,294,967,295
64	doubleword, longword, long long, quad, quadword	<i>Signed:</i> -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
		<i>Unsigned:</i> 0 to +18,446,744,073,709,551,615
128	octaword	<i>Signed:</i> -170,141,183,460,469,231,731,687,303,715,884,105,728 to +170,141,183,460,469,231,731,687,303,715,884,105,727
		<i>Unsigned:</i> 0 to +340,282,366,920,938,463,463,374,607,431,768,211,455
$n$	$n$ -bit integer (general case)	<i>Signed:</i> $(-2^{n-1})$ to $(2^{n-1} - 1)$
		<i>Unsigned:</i> 0 to $(2^n - 1)$

## Binary Coded Decimal Numbers

Binary-coded decimal (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of circuits needed to implement mathematical operations and a relatively inefficient encoding—it occupies more space than a pure binary representation.

Though BCD is not as widely used as it once was, decimal fixed-point and floating-point are still important and still used in financial, commercial, and industrial computing.

To BCD-encode a decimal number using the common encoding, each decimal digit is stored in a four-bit nibble.

Decimal:	0	1	2	3	4	5	6	7	8	9
BCD:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Thus, the BCD encoding for the number 127 would be:  
0001 0010 0111

Since most computers store data in eight-bit bytes, there are two common ways of storing four-bit BCD digits in those bytes:

- each digit is stored in one nibble of a byte, with the other nibble being set to all zeros, all ones (as in the EBCDIC code), or to 0011 (as in the ASCII code)
- two digits are stored in each byte.

Unlike binary encoded numbers, BCD encoded numbers can easily be displayed by mapping each of the nibbles to a different character. Converting a binary encoded number to decimal for display is much harder involving integer multiplication or divide operations.

## Two's Complement

### Calculating two's complement

In two's complement notation, a positive number is represented by its ordinary binary representation, using enough bits that the high bit, the sign bit, is 0. The two's complement operation is the negation operation, so negative numbers are represented by the two's complement of the representation of the absolute value.

In finding the two's complement of a binary number, the bits are inverted, or "flipped", by using the bitwise NOT operation; the value of 1 is then added to the resulting value.

Let's assume 8-bit signed binary numbers.  $A = 23_{10} = 17_{16} = 00010111_2$ . We would like to find the two's complement for A (-A).

```
one's complement of A:  11101000
                        +           1
                        -----
                          11101001
```

So the two's complement representation of  $-23_{10} = 11101001_2 = E9_{16}$ .

You may shorten this process by using hexadecimal representation

```
A          :    17
one's complement of A:  E8
                        +  1
                        -----
                          E9
```

An alternative approach: start from the least significant bit of the binary representation of A,  $00010111_2$ . As long as bits are equal to 0, copy them to the output. When the first 1 in sequence is encountered, copy it to the output, and afterwards complement each incoming bit. In our case, the lsb bit is 1, so it is copied to the output unchanged. All other bits are complemented, which results in  $11101001_2$ .

A shortcut to manually convert a binary number into its two's complement is to start at the least significant bit (LSB), and copy all the zeros (working from LSB toward the most significant bit) until the first 1 is reached; then copy that 1, and flip all the remaining bits. This shortcut allows a person to convert a number to its two's complement without first forming its ones' complement. For example: the two's complement of "0001\_0111" is "11101001", where the underlined digits are unchanged by the copying operation.

### Arithmetic operations: addition, subtraction, multiplication and the use of flags

#### Addition

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:

```

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

```

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.

The last two bits of the carry row (reading right-to-left) contain vital information: whether the calculation resulted in an arithmetic 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 into but not out of the far left sign bit, or out of but not into the sign bit. As mentioned above, the sign bit is the leftmost bit of the result.

If the last two carry bits are both 1's or both 0's, the result is valid; if the last two carry bits are "1 0" or "0 1", a sign overflow has occurred. Conveniently, an XOR operation on these two bits can quickly determine if an overflow condition exists. As an example, consider the 4-bit addition of 7 and 3:

```

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

```

In this case, the far left two (MSB) carry bits are "01", which means there was a two's-complement addition overflow. That is, ten is outside the permitted range of -8 to 7.

### Subtraction

Computers usually use the method of complements to implement subtraction. But although using complements for subtraction is related to using complements for representing signed numbers, they are independent; direct subtraction works with two's-complement numbers as well. 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:

```

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

```

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.

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

```

    0000 1111  (15)
-   0010 0011  (35)
=====
    1110 1100  (-20)
    11100 000  (borrow)

```

### Multiplication

The product of two n-bit numbers can potentially have 2n bits. If the precision of the two two's-complement operands is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result. For example, take  $6 \times -5 = -30$ . First, the precision is extended from 4 bits to 8. Then the numbers are multiplied, discarding the bits beyond 8 (shown by 'x'):

```

    00000110  (6)
  × 11111011  (-5)
  =====
           110
          110
           0
         110
        110
       110
      x10
     xx0
    =====
   xx11100010  (-30)

```

### Fraction Numbers

Two conventions: Fixed point and Floating Point

#### Fixed point

Binary radix point assigned a fixed location in byte (or word). Fixed-point numbers are useful for representing fractional values when the executing processor has no floating point unit (FPU) or if fixed-point provides improved performance or accuracy for the application at hand. Most low-cost embedded microprocessors and microcontrollers do not have an FPU. In binary fixed-point numbers, each magnitude bit represents a power of two, while each fractional bit represents an inverse power of two. Thus the first fractional bit is  $\frac{1}{2}$ , the second is  $\frac{1}{4}$ , the third is  $\frac{1}{8}$  and so on. For signed fixed point numbers in two's complement format, the upper bound is given by  $2^{(m-1)} - 2^{(-f)}$ , and the lower bound is given by  $-2^{(m-1)}$ , where m and f are the number of bits in M and F respectively. For unsigned values, the range is 0 to  $2^m - 2^{(-f)}$ .

00000.101 = 0.625

Granularity of precision in function of number of fractional bits assigned

4 fractional bits =  $2^4 = 0.0625$  = the smallest fraction.

### **Floating-point**

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point numbers (including negative zero and denormal numbers) and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions (including when the exceptions occur, and what happens when they do occur).

Single Precision, Double Precision, Extended Precision.

Single Precision (32 bits): S EEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF

Value =  $(-1)^S 2^{(E-127)} * (1.F)$

### **References**

- [http://en.wikipedia.org/wiki/Binary\\_number](http://en.wikipedia.org/wiki/Binary_number)
- [http://en.wikipedia.org/wiki/Two's\\_complement](http://en.wikipedia.org/wiki/Two's_complement)
- [http://en.wikipedia.org/wiki/IEEE\\_754](http://en.wikipedia.org/wiki/IEEE_754)