NUMBER SYSTEMS

Base 10 (Decimal):

The usual way that we write our numbers is based on the number 10. For example, the number 1382 can be expressed as:

$$1 \times 10^3 + 3 \times 10^2 + 8 \times 10^1 + 2 \times 10^0$$

Because this system requires writing numbers that are based on powers of ten, we say that the number 1382 is written in base 10. Some believe that the base 10 number system evolved because we have ten fingers; thus, making base 10 natural for humans.

Base 2 (Binary):

A bit in a computer has two possible states: off (0) or on (1). Therefore, it is natural for a computer to use a base 2 number system. A base 2 number system uses powers of two instead of powers of ten. A number expressed in base 2 is called a binary number. For example, 1010 can be expressed as:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

Solving the expression gives us the decimal equivalent of the binary number. In the example above, the decimal equivalent is:

$$1010 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 10$$

or simply,

$$1010_2 = 10_{10}$$

Base 8 (Octal):

Programmers often use a number system based on 8 and on 16. These number systems are called octal (base 8) and hexadecimal (base 16), respectively. Because base 8 and base 16 are each powers of 2, these systems are more closely related to a computer's binary system than the decimal system is.

The octal number system is base 8, thus we use the digits 0 to 7 for octal numbers. For example, the octal number 2471 (written 02471 in C) can be expressed as:

$$2 \times 8^3 + 4 \times 8^2 + 7 \times 8^1 + 1 \times 8^0$$

Solving the expression gives us the decimal equivalent of the octal number. In the example above, the decimal equivalent is:

$$02471 = 2 \times 512 + 4 \times 64 + 7 \times 8 + 1 \times 1 = 1337$$

or simply,

$$2471_8 = 1337_{10}$$

Also, since 8 is 2^3 (i.e., 3 bits per octal number), we can write the binary equivalent of the octal number by inspection. Thus,

```
02471 = 010 100 111 001
```

or simply,

$2471_8 = 010100111001_2$

Base 16 (Hexadecimal):

The hexadecimal number system is base 16; thus we use the digits 0 to F for hexadecimal numbers. For example, the hexadecimal number 1F4A (written 0x1F4A in C) can be expressed as:

$1 \times 16^3 + F \times 16^2 + 4 \times 16^1 + A \times 16^0$

Solving the expression gives us the decimal equivalent of the hexadecimal number. In the example above, the decimal equivalent is:

```
0x1F4A = 1 x 4096 + 15 x 256 + 4 x 16 + 10 x 1 = 8010
```

or simply,

$1F4A_{16} = 8010_{10}$

Also, since 16 is 2^4 (i.e., 4 bits per hexadecimal number), we can write the binary equivalent of the hexadecimal number by inspection. Thus,

```
0x1F4A = 0001 1111 0100 1010
```

or simply,

$1F4A_{16} = 0001111101001010_2$

The table below shows the binary, octal, and hexadecimal equivalents for decimal numbers ranging from 0 to 15.

DECIMAL	BINARY	OCTAL	HEXADECIMAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Unsigned and Signed Integers:
For ease of presentation, we will consider byte-sized integers (i.e., integers that are 8 bits wide). The student can extrapolate this discussion for larger sized integers

An unsigned integer uses all bits to represent the magnitude of the number. The binary number 00000000 is zero and 11111111 is the maximum number that can be represented by the 8-bit unsigned integer, i.e., 255. So, the range is from 0-255.

A signed integer uses 00000000 to 01111111 for representing the range of positive numbers (0 to 127), and a two's complement method for representing negative numbers (-1 to -128).

Two's Complement Method:
To represent a negative number in binary using the two's complement method, you should do the following:

1. Write the magnitude of the number to be negated in binary
2. Perform a one's complement on the number in step 1 (i.e., flip each bit)
3. Add 1 to the number in step 2

To represent a negative number (written in binary) by determining its decimal equivalent, you should do the following:

1. Subtract 1 from the binary number
2. Perform a one's complement on the number in step 1 (i.e., flip each bit)
3. Determine the magnitude of the number in step 2

BINARY ARITHMETIC

Binary Addition:

Fundamentals:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (carry the 1 to the next order column)
```

Example:

```
  0 0 1 0 0 1 0 1              37
+ 0 0 1 1 0 0 0 1            + 49
-----------------            ----
  0 1 0 1 0 1 1 0              86
```

Binary Subtraction:

Fundamentals:

```
0 - 0 = 0
0 - 1 = 1 (borrow 1 from the next order column)
1 - 0 = 1
1 - 1 = 0
```

Example:

```
  0 1 0 1 0 1 0 1              85
- 0 0 0 1 1 1 0 1            - 29
-----------------            ----
  0 0 1 1 1 0 0 0              56
```

Binary Multiplication:

Fundamentals:

```
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
```

Example:

```
        0 0 0 1 1 0 0 1              25
      - 0 0 0 0 0 1 0 1            x  5
      -----------------            ----
        0 0 0 1 1 0 0 1             125
      0 0 0 0 0 0 0 0
+ 0 0 0 1 1 0 0 1
---------------------
  0 0 0 1 1 1 1 1 0 1   = 125(10)
```

Binary Division:

Binary division is the repeated process of subtraction, just as we do when performing base 10 division.

Example:

```
            _________1_1_0_0_          ___12_
0 1 1 1 ) 0 1 0 1 0 1 0 0          7 ) 84
            - 1 1 1
            -------
                1 1 1
              - 1 1 1
              -------
                    0 0
                  - 0 0
                  -----
                      0 0
                    - 0 0
                    -----
                        0
```

FLOATING-POINT NUMBERS (IEEE Standard 754)

Floating-point numbers are used as representations of real numbers on a computer. IEEE Standard 754 defines the requirements for floating point numbers and should be referred to for information beyond the simple presentation below. That is, some special operations such as infinity, NaN, etc... are not covered in this presentation

Floating-point numbers represent real numbers in scientific notation. This representation is, of course, in binary, which means that the exponent is in base 2. There are three components of a floating-point number: the sign, the exponent, and the mantissa.

The Sign Bit:
The sign bit is the located in the MSB position of the floating-point number. A 0 denotes a positive number and a 1 denotes a negative number. Flipping the value of this bit will flip the sign of the number.

The Exponent Field:
The exponent must represent both positive and negative exponents. In order to accomplish this, a bias is added to the actual exponent. Single-precision floating-point numbers (32-bit floats) use a byte-sized exponent field. Since there are 8 bits in the exponent field, the unsigned range of the exponent field is from 0 to 255. To best split this bit range for representing positive and negative exponents, a bias of 127 is added to the field.

So, an exponent of zero means that 127 (0+127) is stored in the exponent field. An exponent of –127 means that 0 (-127+127)is stored in the exponent field, and an exponent of 128 means that 255 (128+127) is stored in the exponent field. It should be noted that double-precision floating point numbers (64-bits) have an exponent field 11 bits wide and use a bias of 1023.

The Mantissa Field:
The mantissa, or significand, represents the precision of the number. The mantissa has two components: an implicit leading bit (always 1) and the fraction bits. For single-precision floating-point numbers, this field is 23 bits wide. For double-precision floating-point numbers, this field is 52 bits wide.

The Implicit Leading Bit:
We say that the implicit leading bit is always one due to an optimization that is available to us due to the binary numbering. Since we are using binary, the only possible digit values are 0 and 1. When we store a floating-point number in normalized form (normalized scientific notation), the radix point is put after the first non-zero digit. In binary, this digit can only be a 1, thus, we can assume this leading bit and gain an additional bit of resolution (from 23 bits to 24 bits).

The Fraction Bits:
Since the mantissa has an implicit leading bit that is always 1, the form of the mantissa is 1.*f*, where *f* is the field of fraction bits.

Example,

What is the single-precision, floating-point representation of the float number 14.125?

The base 10 fraction equivalent of 14.125 is: 14 1/8
The binary value prior to normalization is: 1110.001
The normalized binary value is: 1.110001×2^3

Sign Bit:
Since this is a positive number, the sign bit is $\boxed{0}$.

Exponent Field:
Since the exponent is +3, the exponent field is:

$= 3 + 127 = 130$

Converting to binary, we have:

$130_{10} = \boxed{10000010_2}$

Mantissa Field:
The normalized binary value was determined to be 1.110001×2^3. The implicit bit is not used in the actual mantissa field. Thus, the mantissa (which is 23 bits wide) is:
$\boxed{11000100000000000000000_2}$.

Putting it all together, we have:

0 10000010 11000100000000000000000

or simply,

$14.125_{10} = 01000001011000100000000000000000_2 = 0x41620000$

ASCII CHARACTER TYPES

Character data is not merely letters of the alphabet. Character data is also numeric characters, punctuation, spaces, tabs, newlines, etc... The ASCII (American Standard Code for Information Interchange) character set (non-extended) contains 128 ASCII characters. So, only 7 bits are needed to represent the ASCII character set. This nicely fits into a byte where the MSB of the ASCII character is set to 0. The ASCII character set is shown below along with its base 10 numerical equivalent.

ASCII Characters (Decimal)

```
 0 nul   16 dle   32 sp    48 0     64 @     80 P     96 `    112 p
 1 soh   17 dc1   33 !     49 1     65 A     81 Q     97 a    113 q
 2 stx   18 dc2   34 "     50 2     66 B     82 R     98 b    114 r
 3 etx   19 dc3   35 #     51 3     67 C     83 S     99 c    115 s
 4 eot   20 dc4   36 $     52 4     68 D     84 T    100 d    116 t
 5 enq   21 nak   37 %     53 5     69 E     85 U    101 e    117 u
 6 ack   22 syn   38 &     54 6     70 F     86 V    102 f    118 v
 7 bel   23 etb   39 '     55 7     71 G     87 W    103 g    119 w
 8 bs    24 can   40 (     56 8     72 H     88 X    104 h    120 x
 9 ht    25 em    41 )     57 9     73 I     89 Y    105 i    121 y
10 nl    26 sub   42 *     58 :     74 J     90 Z    106 j    122 z
11 vt    27 esc   43 +     59 ;     75 K     91 [    107 k    123 {
12 np    28 fs    44 ,     60 <     76 L     92 \    108 l    124 |
13 cr    29 gs    45 -     61 =     77 M     93 ]    109 m    125 }
14 so    30 rs    46 .     62 >     78 N     94 ^    110 n    126 ~
15 si    31 us    47 /     63 ?     79 O     95 _    111 o    127 del
```

The characters between 0 and 31 are generally not printable (control characters, etc).

BIT-WISE LOGICAL AND SHIFT OPERATORS

The bit-wise operators, in their order of precedence, are shown in the table below.

Operator	Meaning	Type	Associativity
~	One's Complement	Logical	R-L
&	Bit-wise AND	Logical	L-R
^	Bit-wise Exclusive OR	Logical	L-R
|	Bit-wise OR	Logical	L-R
<< >>	Left-shift Right-Shift	Shift	L-R
<<= >>=	Left-shift assignment Right-shift assignment	Shift	L-R
&= |= ^=	Bit-wise AND assignment Bit-wise OR assignment Bit-wise XOR assignment	Logical	R-L

UNARY OPERATOR:

The unary (one's complement) operator is used to perform a one's complement of the element. That is, each 1 changes to a 0 and each 0 changes to a 1 within the element.

For example,

```
Suppose:                num == 0x55, which is binary 01010101
Then it follows that:  ~num == 0xAA, which is binary 10101010
```

BIT-WISE AND OPERATOR:

The bit-wise AND operator will produce a result that is the logical AND of two operands on a bit-by-bit basis. The bit-wise AND follows the truth table shown below for each bit of the two operands, A and B.

Truth Table,

A_n	B_n	A_n & B_n
0	0	0
0	1	0
1	0	0
1	1	1

BIT-WISE EXCLUSIVE OR (XOR) OPERATOR:

The bit-wise XOR operator will produce a result that is the logical XOR of two operands on a bit-by-bit basis. The bit-wise XOR follows the truth table shown below for each bit of the two operands, A and B.

Truth Table,

A_n	B_n	A_n ^ B_n
0	0	0
0	1	1
1	0	1
1	1	0

BIT-WISE OR OPERATOR:

The bit-wise OR operator will produce a result that is the logical OR of two operands on a bit-by-bit basis. The bit-wise OR follows the truth table shown below for each bit of the two operands, A and B.

Truth Table,

A_n	B_n	A_n | B_n
0	0	0
0	1	1
1	0	1
1	1	1

LEFT-SHIFT OPERATOR:

The left-shift operator is used to shift all bits of a left operand to the left by the number of positions given by the right operand. The bits that are shifted out are lost bits and the bits that are shifted in are filled with 0's.

For example,

```
Suppose:                num == 0x55      which is binary 01010101
Then it follows that:   num << 3 == 0xA8 which is binary 10101000
```

RIGHT-SHIFT OPERATOR:

The right-shift operator is used to shift all bits of a left operand to the right by the number of positions given by the right operand. The bits that are shifted out are lost bits and the bits that are shifted in are filled with 0's for unsigned bits. For signed bits, they may be filled with 0's or copies of the leftmost bit (this is machine dependent).

For example,

```
Suppose num is unsigned: num == 0x55     which is binary 01010101
Then it follows that:    num >> 3 ==0x0A which is binary 00001010

Suppose num is unsigned: num == 0xAA     which is binary 10101010
Then it follows that:    num >> 3 ==0x15 which is binary 00010101

Suppose num is signed:  num == 0x55      which is binary 01010101
Then it follows that:   num >> 3 == 0x0A which is binary 00001010

Suppose num is signed:  num == 0xAA      which is binary 10101010
Then it follows that:   num >> 3 == 0x15 which is binary 00010101
OR:                     num >> 3 == 0xF5 which is binary 11110101
```

OPERATOR-ASSIGNMENT OPERATORS (<<=, >>=, &=, ^=, |=):

An operator-assignment operator will perform the operation of the operator and assign the value to the left operand. It is a short-hand notation for combining the bit-wise operator with the assignment operator. The following table lists the long-form assignment statement and the correlating short-hand operator-assignment statement.

LONG-HAND ASSIGNMENT NOTATION	EQUIVALENT OPERATOR-ASSIGNMENT
num = num << 4;	num <<= 4;
num = num >> 2;	num >>= 2;
num = num & num2;	num &= num2;
num = num ^ num2;	num ^= num2;
num = num | num2;	num |= num2;

MASKS – HOW TO TURN ON BITS:

A mask is a unique bit pattern in which one or more of the bits of the mask are turned ON (bit values = 1) and other bits are turned OFF (bit values = 0). They are useful for the purpose of turning ON or OFF certain bits of a flag where the bits are determined by the mask setting. To turn a bit ON, the mask value at the bit position should be set at 1 and the mask should be bit-wise OR'd with the flag.

```
For Example,
int flag = 0x98; /* 10011000 */
int mask = 0x82; /* 10000010 */

flag |= mask;    /* 10011010 Note: mask turned on b1, b7*/
```

MASKS – HOW TO TURN OFF BITS:

To turn a bit OFF, the mask value at the bit position should be set at 1 and the one's complement of the mask should be bit-wise AND'ed with the element.

```
For Example,
int flag = 0x98; /* 10011000 */
int mask = 0x82; /* 10000010 Note: ~mask is 01111101*/

flag &= ~mask;   /* 00011000 Note: ~mask turned off b1, b7*/
```