NOTES - Bit Manipulation

Working on bytes, or data types comprising of bytes like ints, floats, doubles or even data structures that stores a large number of bytes is normal for a programmer. In some cases, a programmer needs to go beyond this - that is to say that on a deeper level where the importance of bits is realized.

Operations with bits are used in Data compression (data is compressed by converting it from one representation to another, to reduce the space), Exclusive-Or Encryption (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at the bit level. Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

We all know that 1 byte comprises 8 bits and an integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

Example:

1) $14 = \{1110\}_2$

$= 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

$= 14.$

2) $20 = \{10100\}_2$

$= 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$

$= 20.$

For characters, we use ASCII representation, which is in the form of integers which again can be represented using bits as explained above.

Bitwise Operators:

There are different bitwise operations used in the bit manipulation. These bit operations operate on the individual bits of the bit patterns. Bit operations are fast and can be used in optimizing time complexity. Some common bit operators are:

NOT (~): Bitwise NOT is a unary operator that flips the bits of the number i.e., if the ith bit is 0, it will change it to 1 and vice versa. Bitwise NOT is nothing but simply the one’s complement of a number. Let's take an example.

$N = 5 = (101)_2$

$\sim N = \sim 5 = \sim(101)_2 = (010)_2 = 2$

AND (&): Bitwise AND is a binary operator that operates on two equal-length bit patterns. If both bits in the compared position of the bit patterns are 1, the bit in the resulting bit pattern is 1, otherwise 0.

$A = 5 = (101)_2$, $B = 3 = (011)_2$ $A \& B = (101)_2 \& (011)_2 = (001)_2 = 1$

OR (|): Bitwise OR is also a binary operator that operates on two equal-length bit patterns, similar to bitwise AND. If both bits in the compared position of the bit patterns are 0, the bit in the resulting bit pattern is 0, otherwise 1.

$A = 5 = (101)_2$, $B = 3 = (011)_2$

$A \mid B = (101)_2 \mid (011)_2 = (111)_2 = 7$

XOR (^): Bitwise XOR also takes two equal-length bit patterns. If both bits in the compared position of the bit patterns are 0 or 1, the bit in the resulting bit pattern is 0, otherwise 1.

$A = 5 = (101)_2$, $B = 3 = (011)_2$

$A \wedge B = (101)_2 \wedge (011)_2 = (110)_2 = 6$

Left Shift (<<): Left shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the left and append 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2k (if we are shifting k bits).

$1 << 1 = 2 = 2^1$

$1 << 2 = 4 = 2^2$ $1 << 3 = 8 = 2^3$

$1 << 4 = 16 = 2^4$

…

$1 << n = 2^n$

Right Shift (>>): Right shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the right and append 1 at the end. Right shift is equivalent to dividing the bit pattern with 2k (if we are shifting k bits).

4 >> 1 = 2

6 >> 1 = 3

5 >> 1 = 2

Bitwise operators are good for saving space and sometimes to cleverly remove dependencies.

Note: All left and the right side are taken with reference to the reader.