

# Bit Manipulation

---

## What is Bit Manipulation?

Bit manipulation is basically the act of algorithmically manipulating bits (smallest unit of data in a computer).

A bit represents a logical state with one of two possible values '0' or '1', which in other representations can be referred to as 'true' or 'false' / 'on' or 'off'. Since bits represent logical states efficiently, this makes the binary system suitable for electronic circuits that use logic gates and this is why binary is used in all modern computers.

## Decimal and Binary number systems

- **Decimal number system:**

We usually represent numbers in decimal notation (base 10) that provides ten unique digits - 0,1,2,3,4,5,6,7,8,9. To form any number, we can combine these digits to form a sequence such that each decimal digit represents a value multiplied by a power of 10 in this sequence.

### For example:

$$244 = 200 + 40 + 4 = 2 \cdot 10^2 + 4 \cdot 10^1 + 4 \cdot 10^0.$$

- **Binary number system:**

The binary system works in a similar manner to that of the decimal number system, the only difference lies in the fact that binary notation (base 2) provides two digits - 0 and 1. A binary number is defined as a sequence of 1s and 0s like '010100', '101', '0', '1111', '000111', etc. In a binary number, each digit is referred to as a bit, where each such bit represents a power of decimal 2.

Similar to decimal notation we can convert a given binary sequence to its decimal equivalent.

**For example:**

(base 2)1100: (Base 10)  $1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 8 + 4 + 0 + 0 = 12$ .

A bit is said to be **set** if it is '1' and **unset** if the bit is '0'.

## Bitwise Operators

Bitwise operators are used to performing bitwise operations. A bitwise operation operates on a bit string(string consisting of 0s and 1s only) or a bit array(array of 0s and 1s only) at the level of its individual bits. Bit operations are usually fast and are used for calculations and comparisons at the processor level.

- **Bitwise AND(&)**

A **binary operator** that takes bit representation of its operands and the resulting bit is 1 if both the bits in bits patterns are 1 otherwise, the resulting bit is 0.

**For example:**

$X = 5 (101)_2$  and  $Y = 3 (011)_2$

$X \& Y = (101)_2 \& (011)_2 = (001)_2 = 1$

- **Bitwise OR(|)**

A **binary operator** that takes bit representation of its operands and the resulting bit is 0 if both the bits in bits patterns are 0 otherwise, the resulting bit is 1.

**For example:**

$X = 5 (101)_2$  and  $Y = 3 (011)_2$

$X \mid Y = (101)_2 \mid (011)_2 = (111)_2 = 7$

- **Bitwise NOT(~)**

A **unary operator** that takes bit representation of its operand and just flips the bits of the bit pattern, i.e if the  $i$ th bit is 0 then it will be changed to 1 and vice versa.

Bitwise NOT is the **1's complement** of a number.

**For example:**

$X = 5 (101)_2$

$\sim X = (010)_2 = 2$

- **Bitwise XOR(^)**

A **binary operator** that takes bit representations of its operands and the resulting bit is 0 if the bits in bit patterns are the same i.e the bits in both the patterns at a given position is either 1 or 0 otherwise, the resulting bit is 1 if the bits in bit patterns are different.

**For example:**

$X = 5 (101)_2$  and  $Y = 3 (011)_2$

$X \wedge Y = (101)_2 \wedge (011)_2 = (110)_2 = 6$

- **Left shift operator(<<)**

A **binary operator** that left shifts the bits of the first operand, with the second operand deciding the number of places to shift left, and append that many 0s at the end. We discard the  $k$  left most bits.

**For example:**

$$X = 5 \text{ (101)}_2$$

$$X \ll 2 = (00101)_2 \ll 4 = (10100)_2 = 20$$

In other words, left shift by  $k$  places is equivalent to multiplying the bit pattern by  $2^k$ , as shown in the above example -  $5 \ll 2 = 5 * 2^2 = 20$ .

So,  $A \ll x = A * 2^x$  that is why  $(1 \ll n)$  is equal to  $\text{power}(2, n)$  or  $2^n$ .

- **Right shift operator(>>)**

A **binary operator** that right shifts the bits of the first operand, with the second operand deciding the number of places to shift right. We discard the  $k$  rightmost bits.

**For example:**

$$X = 5 \text{ (101)}_2$$

$$X \gg 2 = (101)_2 = (001)_2 = 1$$

In other words, the right shift by  $k$  places is equivalent to dividing the bit pattern by  $2^k$  (integer division), as shown in the above example -  $5 \gg 2 = 5 / (2^2) = 1$ .

So,  $A \gg x = A / (2^x)$  (integer division).

X	Y	X & Y	X   Y	X ^ Y	~X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

## Some useful bitwise algorithms/bit hacks

- **Checking if a number is a power of 2**

**Algorithm:**

The numbers which are powers of 2 have only one set bit in their binary representation. If the number is 0 or is not a power of 2 then it will have more than one bit set in binary representation.

Let us say, we need to check if a number  $x$  is a power of 2 or not.

Now, think about the binary representation of  $(x-1)$ , the binary representation of  $(x-1)$  will have all the bits the same as  $x$  except for the rightmost set bit in  $x$  and all the bits to the right of the rightmost set bit.

**For example:**

$x = 4$  (00100)<sub>2</sub>

$x-1 = 3$  (00011)<sub>2</sub>

$x = 10$  (1010)<sub>2</sub>

$x-1 = 9$  (1001)<sub>2</sub>

Now, how  $(x-1)$  can help determine whether  $x$  is a power of 2?

The binary representation of  $(x-1)$  can be simply obtained from the binary representation of  $x$  by flipping all the bits to the right of the rightmost set bit including the rightmost set bit itself, so if  $x$  has only one-bit set, then  $x \& (x-1)$  must be equal to 0.

**For example:**

$x = 4$  (00100)<sub>2</sub>

$$x-1 = 3 \text{ (00011)}_2$$

$$x \& (x-1) = \text{(00000)}_2$$

In the above example, since the rightmost set bit and all the bits to the right of it were flipped, so the bitwise and will make all those bits 0 and we have no set bits left, which tells us that there was only one set bit in x.

### Pseudocode:

```
function isPowerOf2(num)
    return (num && !(num & (num - 1)))
```

A similar approach can be used for checking if a given number is **even** or **odd**.

One of the obvious approaches would be to check the number's divisibility by 2, but the other approach is to check if the last(rightmost) bit i.e if the last bit is 1 or not if the last bit is 1 then the number is odd, otherwise even, as the rightmost bit i.e  $2^0$  only contributes odd value - 1 to the number, all other bits represent the value of  $2^i$  where  $i > 0$ , so all values remain even.

- **Checking if the kth bit is set/unset**

### Algorithm:

We need to check if the kth bit in the binary representation of the given number is set or unset. We can use bitwise AND operations to check this efficiently.

We can take another number  $n = (1 \ll k)$ , whose all bits are 0 except the kth bit.

Now doing AND operation with the given number say x, must return a positive number if the kth bit is set, otherwise the result of bitwise AND will be 0.

**For example:**

$x = 5$  ( $101$ )<sub>2</sub>,  $k = 2$

So, let  $no = (1 \ll k) = 2^k = 4 = (100)$ <sub>2</sub>

$x \& no = (101)$ <sub>2</sub>  $\&$   $(100)$ <sub>2</sub> =  $(100)$ <sub>2</sub> =  $4$ .

So the 2nd bit of  $x$  is set.

**Pseudocode:**

```
function checkBit(N, k)
    if (N & (1<<k))
        return true
    else
        return false
```

- **Counting number of 1s in the binary representation of a number**

The naive approach is to simply convert the given number into its binary representation and simply count the number of 1s in the binary representation, which takes  $O(\log_2 N)$  time, where  $N$  is the number.

However, we can efficiently count the number of 1s in the binary representation of a number whose time complexity depends on the number of 1s in its binary representation.

**Algorithm:**

Let us say we want to find the number of 1s in the binary representation of  $x$ , we can perform a similar operation used to check if a given number is a power of 2.

We can apply bitwise and between  $x$  and  $(x-1)$  and then update the value of  $x$  to  $x \& (x-1)$ , every time this operation flips the rightmost set bit and the bits after the rightmost set bit, so every time the rightmost set bit is unset, which helps us count the number of 1s in the binary representation of the number.

**For example:**

$x = 5$   $(101)_2$

Step 1:  $x = 5$   $(101)_2$

$x-1 = 4$   $(100)_2$

$x \& (x-1) = (100)_2 = 4$ , update  $x$  to 4.

Step 2:  $x = 4$   $(100)_2$

$x-1 = 3$   $(011)_2$ ,

$x \& (x-1) = (000)_2 = 0$

So, the number of set bits are 2 in  $x$ .

The above algorithm described is known as **Brian Kernighan's Algorithm**. The time complexity of the algorithm is dependent on the number of 1s in the binary representation of the number, however in the worst case still the time complexity will be  $O(\log_2 N)$  as all bits of the number may be set, where  $N$  is the number.

**Pseudocode:**



```
function count1s(num)
    count = 0
    while(num > 0)
        num = num & (num-1)
        count = count+1
    return count
```

- **Generating all possible subsets of a set**

Bit manipulation can be useful for the generation of all the possible subsets of a set. We can represent the elements of the set in the form of a binary sequence having a length equal to the number of elements in the set. Since the number of subsets of a set having  $N$  elements is  $2^N$ , so a binary sequence of length  $N$  will represent values from range 0 to  $2^N-1$  which is equal to the number of subsets of the set.

All set bits in the binary sequence denote the elements present from the set, thus representing a subset.

**For example:**

Set  $S = \{10, 20, 30\}$

We know that the number of subsets of a set having  $N$  elements is  $2^N$ . In this case  $N = 3$ , so  $2^N = 8$  subsets.

So taking a binary sequence of length 3, where each bit value represents whether the element of the set is present (bit value is 1) or not (bit value is 0) in the corresponding subset.

$0 = 000 \Rightarrow \{\}$  - empty subset

$1 = 001 \Rightarrow \{30\}$

$2 = 010 \Rightarrow \{20\}$

3 = 011 => {20,30}

4 = 100 => {10}

5 = 101 => {10,30}

6 = 110 => {10,20}

7 = 111 => {10,20,30}

Hence, the binary sequence is useful to represent all the subsets of a set.

### Pseudocode:

```
function generateSubsets(arr, N)
    /*
        arr represents the set of size N
        Iterating from 0 to 2^N-1 to get all subsets
    */
    for i = 0 to 2^N-1
        /*
            For each i representing a subset checking which
            bits are set in binary representation of i
        */
        for j = 0 to N-1
            if (i & (1 << j))
                print arr[j]
        print newline
    return
```

### Some other bitwise tricks

- **res = x | (1 << k):** Set the bit at kth position in the binary representation of x.

**For example:**

$x = 5 (101)_2, k = 1$

Let  $no = (1 \ll k) = 2^1 = 2 = (010)_2$

$res = x \mid no = (101)_2 \mid (010)_2 = (111)_2 = 7$

- **$res = x \& \sim(1 \ll k)$** : Unset the bit at kth position in binary representation of x.

**For example:**

$x = 5 (101)_2, k = 2$

Let  $no = (1 \ll k) = 2^2 = 4 = (100)_2$

$no = \sim(no) = (011)_2$

$res = x \& no = (101)_2 \& (011)_2 = (001)_2 = 1$

- **$res = x \wedge (1 \ll k)$** : Toggle the bit at kth position in the binary representation of x  
i.e if the bit at kth position is 0 then change it to 1 and vice versa.

**For example:**

$x = 5 (101)_2, k = 0$

Let  $no = (1 \ll k) = 2^0 = 1 = (001)_2$

$res = x \wedge no = (101)_2 \wedge (001)_2 = (100)_2 = 4$

- **$res = x \& (x - 1)$** : Unsetting the rightmost set bit of x.

**For example:**

$x = 5 (101)_2$

$x-1 = 4 (100)_2$

$res = x \& (x-1) = (101)_2 \& (100)_2 = (100)_2 = 4$

- **$res = \sim x + 1$  (2's complement)**: 2's complement of a number is its 1's complement +1. 2's complement of a number is the same as negative of the number i.e -x.
- **$res = x \& (-x)$** : Getting the rightmost set bit of x.

**For example:**

$x = 5 \text{ (101)}_2$

1's complement:  $\sim x = (010)_2$

2's complement:  $\sim x + 1 = (011)_2$

$\text{res} = x \& (-x) = (101)_2 \& (011)_2 = (001)_2 = 1$

## Applications of bit manipulation

- Bitwise operations are prominent in embedded systems, control systems, etc where memory(data transmission/data points) is still an issue.
- They are also useful in networking where it is important to reduce the amount of data, so booleans are packed together. Packing them together and taking them apart use bitwise operations and shift instructions.
- Bitwise operations are also heavily used in the compression and encryption of data.
- Useful in graphics programming, older GUIs are heavily dependent on bitwise operations like XOR(^) for selection highlighting and other overlays.