Bitwise manipulation

Bitwise manipulation refers to the manipulation of individual bits (0s and 1s) within binary numbers. It involves performing operations at the bit level, rather than on the entire number.

Think of a number as a series of switches that can be either on or off (1 or 0). Bitwise manipulation involves using operations to turn these switches on or off, check their status, or flip them.

Bitwise manipulation is often used in low-level programming, such as working with hardware interfaces, optimizing algorithms, or manipulating data at the bit level to conserve memory or improve performance.

By using bitwise operators, such as AND, OR, XOR, and NOT, you can modify or extract specific bits in binary numbers.

Here's a breakdown of some commonly used bitwise operators:

* AND (&): This operation compares each bit in two numbers. If both bits are 1, the result is 1. Otherwise, it's 0. It's like saying "if both switches are on, keep the switch on; otherwise, turn it off."  
  Example: 5 & 3 = 1 (binary: 101 & 011 = 001)
* OR (|): This operation also compares each bit in two numbers. If at least one bit is 1, the result is 1. It's like saying "if either switch is on, turn the switch on."

(Compares two bits and returns 1 if either or both bits are 1.) Example: 5 | 3 = 7 (binary: 101 | 011 = 111)

* XOR (^): This operation compares each bit in two numbers. If the bits are different, the result is 1. If they're the same, the result is 0. It's like saying "turn the switch on only if the switches are different."

Example: 5 ^ 3 = 6 (binary: 101 ^ 011 = 110)

* NOT (~): This operation flips each bit. If the bit is 0, it becomes 1, and if it's 1, it becomes 0. It's like saying "flip the status of each switch."

Example: ~5 = -6 (binary: ~101 = -110)

Binary shift operation

Binary shift operations are used to move the bits in a binary number to the left or the right.

Logical Shifts:

Logical Left Shift: This is the simplest form of shift operation. They are called "logical" shifts because they don't take into account the sign bit (the leftmost bit) when shifting.

There are two types of logical shifts:

* Logical left shift.

All the bits in a binary number are shifted to the left by a specified number of positions. The leftmost bits are discarded, and zeros are filled in on the right. For example, if you logical left shift 0011 (3 in decimal) by two positions, it becomes 1100 (12 in decimal).

It is equivalent to multiplying the number by 2 raised to the power of the shift amount.

* Logical Right Shift:

In this operation, bits are shifted to the right. The rightmost bits are discarded, and zeros are filled in on the left. For example, logical right shifting 1100 by two positions gives 0011.

It is equivalent to dividing the number by 2 raised to the power of the shift amount.

**Arithmetic Shifts:**

Arithmetic shifts are similar to logical shifts, but they preserve the sign bit when shifting. They are typically used with signed integers.

There are two types of arithmetic shifts:

* **Arithmetic Left Shift:** This is identical to the logical left shift. The bits are shifted to the left, the leftmost bits are discarded, and zeros are filled in on the right.   
  This operation shifts the bits of a number to the left by a specified number of positions, filling the empty positions with zeros. The sign bit remains unchanged.
* **Arithmetic Right Shift:** This shift is different from the logical right shift. The bits are shifted to the right, but the leftmost bit (the sign bit in signed numbers) is replicated to fill in the new positions. This preserves the sign of the number in two's complement representation, where the leftmost bit indicates the sign (0 for positive, 1 for negative).

Cyclic Shift (also known as Circular Shift or Rotate):  
Cyclic shifts rotate the bits of a number to the left or right, and the shifted out bits are brought back to the other end. This creates a circular effect.

There are two types of cyclic shifts

* Cyclic Left Shift (Rotate Left): In this operation, instead of discarding the leftmost bits, they are wrapped around to the right side. So, if you cyclically shift 1011 to the left by one position, it becomes 0111.
* Cyclic Right Shift (Rotate Right): Similarly, the bits that fall off the right end are wrapped around to the left side. If you cyclically shift 1011 to the right by one position, it becomes 1101

Bit Masking

Bit mask manipulation involves using a bit mask to selectively modify or extract specific bits in a binary number.

Bit masking is a technique used in programming to handle individual bits within a binary number. To understand this, imagine a binary number as a row of light switches, where each switch can be either on (1) or off (0).

Bit masking involves using a special number called a "mask" to select certain bits in your original number. This mask is also a binary number, where each bit corresponds to the bit in the same position in the number you want to work with.

It operates by having 1s in the positions where you want to keep or modify the bits, and 0s in the positions where you want to clear or ignore the bits.

Using the bitwise operations, such as AND, OR, XOR, with a bit mask, you can control which bits are affected and achieve the desired result.

Performing the AND operation

Using the Binary Number: 110101 and Bit Mask: 101110 perform the AND operation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | **1** | 0 | 1 | **0** | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |

Result is 100100

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6th | **5th** | 4th | 3rd | **2nd** | 1st |
| 1 | **1** | 0 | 1 | **0** | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

The binary number 110101 and you want to set the second and fifth bits to 1, while keeping the other bits unchanged. You can use a bit mask with 1s in the positions you want to set and 0s elsewhere, such as 010010. By performing a bitwise **OR** operation between the original number and the bit mask, the desired bits are set to 1:  
Original number: 110101  
Bit mask: 010010  
Result: 111011

Clearing specific bits:  
Conversely, if you want to clear (set to 0) certain bits in a binary number, you can use a bit mask with 0s in the positions you want to clear and 1s elsewhere. The bitwise AND operation between the original number and the bit mask will clear the desired bits:  
Original number: 110101  
Bit mask: 110010  
Result: 110000

1. Extracting specific bits:  
   Bit masks can also be used to extract specific bits from a binary number. By performing a bitwise AND operation between the original number and a bit mask that has 1s only in the positions of the desired bits, you can obtain those bits while discarding the rest:  
   Original number: 110101  
   Bit mask: 001000  
   Result: 000000