Bitwise Operators

An operator that acts on individual bits (0 or 1) of the operands is called **bitwise operator in java**.

It acts only integer data types such as byte, short, int, and long. Bitwise operators in java cannot be applied to float and double data types.

The internal representation of numbers in the case of bitwise operators is represented by the binary number system. Binary number is represented by two digits 0 or 1.

Therefore, these operators are mainly used to modify bit patterns (binary representation).

Operator	Meaning
&	bitwise AND (Binary)
|	bitwise OR (Binary)
^	bitwise exclusive OR (Binary)
~	bitwise NOT (Unary)
<<	shift left
>>	shift right
>>>	unsigned right shift

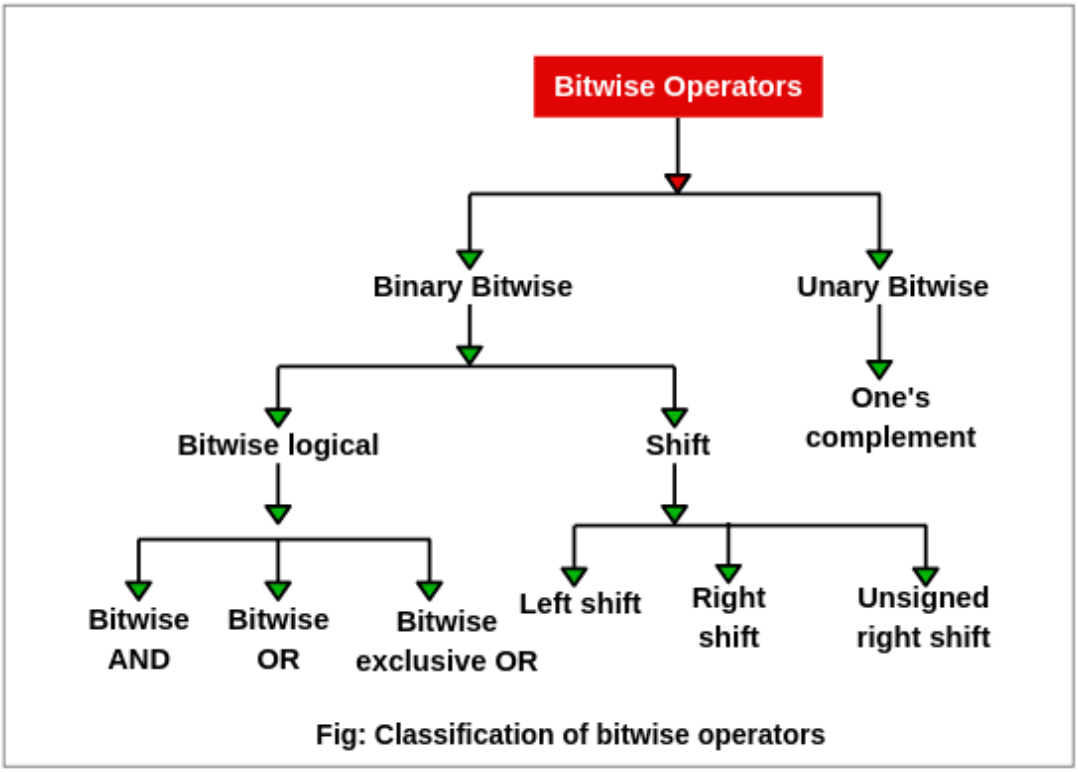

Fig: Classification of bitwise operators

Bitwise AND Operator (&)(arithmetic multiplication)

This operator is used to perform bitwise AND operation between two integral operands. The

AND operator is represented by a symbol & which is called ampersand. It compares each bit of the left operand with the corresponding bit of right operand.

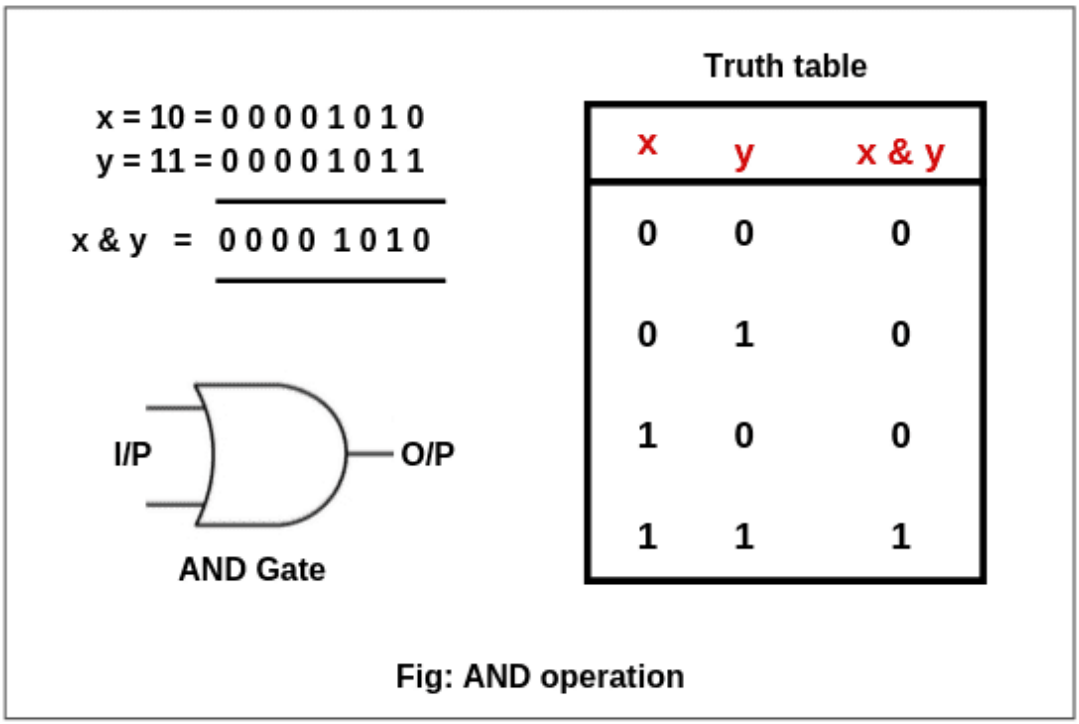

Fig: AND operation

Truth table is a table that gives the relationship between inputs and output. In AND operation, on multiplying two or more input bits, we get output bit.

From the truth table shown in figure, if both compared input bits are 1, we get output 1. Otherwise, we get output 0.

From the truth table, On multiplying the individual bits of x and y, we get x & y = 0 0 0 0 1 0 1 0. It is nothing but 10 in decimal form. Let’s take an example program based on it.

```
package javaProgram;
public class BitwiseANDExample {
public static void main(String[] args)
{
 int a = 10, b = 11;
 System.out.println("(10 & 11): " +(a & b));
 }
}
```

Bitwise OR Operator (|)(arithmetic addition)

This operator is used to perform OR operation on bits of numbers. It is represented by a symbol | called pipe symbol.

In OR operation, each bit of first operand (number) is compared with the corresponding bit of the second operand.

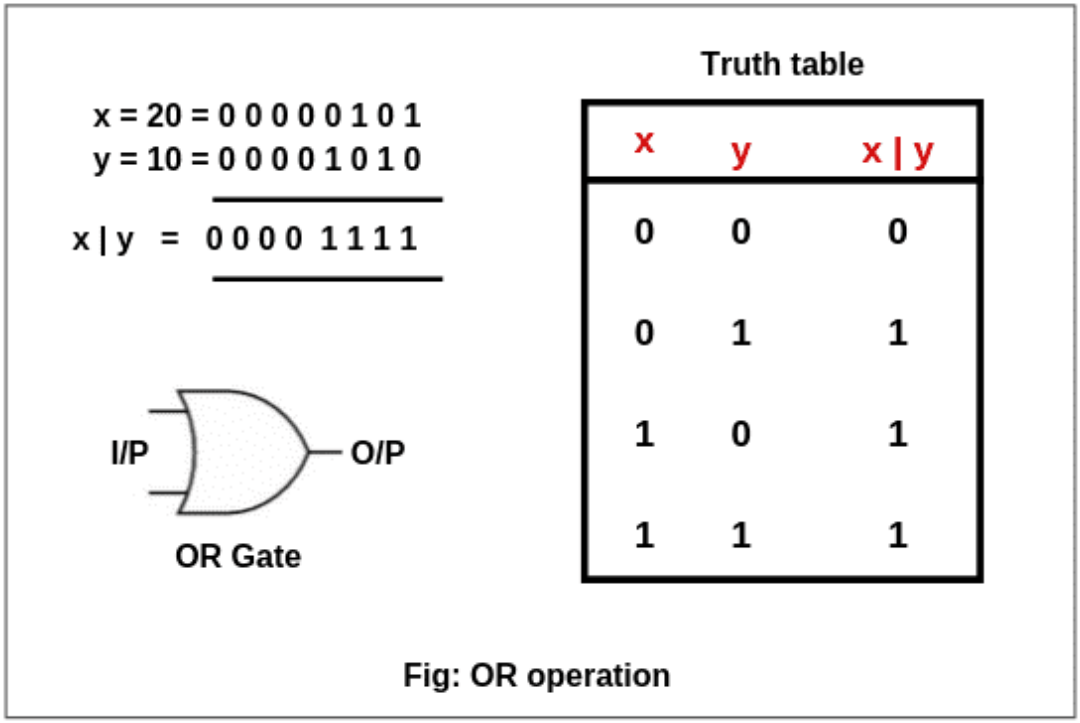

Fig: OR operation

To understand OR operation, consider truth table given in the above figure. If both compared bits are 0, the output is 0. If any one bit is 1 in both bits, the output is 1.

On adding input bits of (x = 20) and (y = 10), we get output bits 0 0 0 0 1 1 1 1. This is nothing but 30 in decimal form.

```
package javaProgram;
public class BitwiseORExample {
public static void main(String[] args)
{
 int a = 20, b = 10;
 System.out.println("(20 | 10): " +(a | b));
 }
}
```

Bitwise Exclusive OR (XOR) Operator (^)

The operator ^ performs exclusive OR (XOR) operation on the bits of numbers. This operator compares each bit first operand (number) with the corresponding bit of the second operand.

Exclusive OR operator is represented by a symbol ^ called cap. Let us consider the truth table of the XOR operator to understand exclusive OR operation.

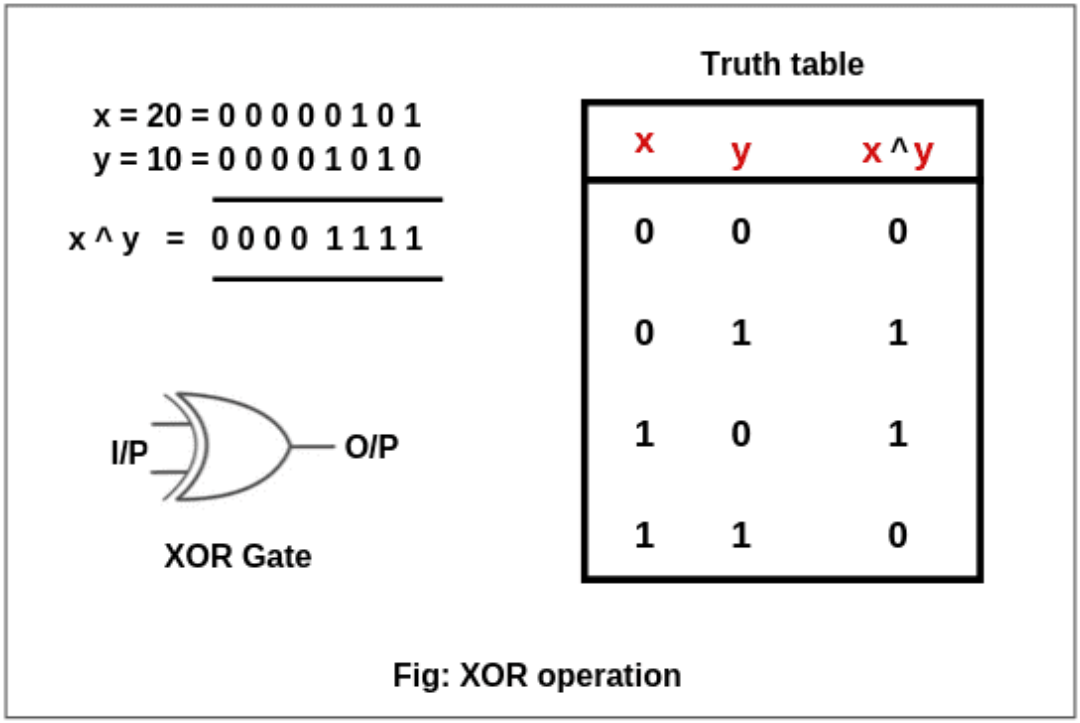

Fig: XOR operation

To understand exclusive OR (XOR) operation, consider the truth table as shown in the above figure. If we have odd number of 1's in input bits, we get output bit as 1.

In other words, if two bits have the same value, the output is 0, otherwise the output is 1.

From the truth table, when we get odd number of 1's then notice that the output is 1 otherwise the output is 0. Hence, x ^ y = 0 0 0 0 1 1 1 1 is nothing but 30 in decimal form.

```
package javaProgram;
public class BitwiseXORExample {
public static void main(String[] args)
{
 int a = 20, b = 10;
 System.out.println("(20 ^ 10): " +(a ^ b));
 }
}
```

Bitwise NOT operator (~)

The bitwise NOT operator in Java
 is basically an inverter. It returns the reverse of its operand or value. It converts all 1s to 0s, and all the 0s to 1s. Therefore, it is also called unary operator or bit flip or one's complement operator.

A	~A
0	1
1	0

```
package javaProgram;
public class BitwiseOperatorsEx {
public static void main(String[] args)
{
 int a = 2, b = 10;
 System.out.println("(2 & 10): " +(a & b));
 System.out.println("(2 | 10): " +(a | b));
 System.out.println("(2 ^ 10): " +(a ^ b));
 System.out.println("~10: " +~b);
 }
}
```

Shift Operator in Java | Left, Signed, Unsigned Right

The operator which is used to shift the bit patterns right or left is called **shift operator in java**.

`left_operand  op  n`

where,

left_operand ➜ left operand that can be of integral type.
op ➜ left shift or right shift operator.
n ➜ number of bit positions to be shifted. It must be of type int only.

`10 << 2`

Here, the value 10 is operand, << is left shift operator, 2 is the number of bit positions to be shifted.

Types of Shift Operator in Java

There are two types of shift operators in Java. They are:

1. Left shift operator
2. Right shift operator

Left Shift Operator in Java

The operator that shifts the bits of number towards left by n number of bit positions is called **left shift operator in Java**.

This operator is represented by a symbol <<, read as double less than.
If we write x << n, it means that the bits of x will be shifted towards left by n positions.

Let us take an example to understand the concept of the Java left shift operator.

1. If int x = 20. Calculate x value if x << 3.

The value of x is 20 = 0 0 0 1 0 1 0 0 (binary format). Now x << 3 will shift the bits of x towards left by 3 positions. Due to which leftmost 3 bits will be lost.

Hence, after shifting, bits of x is 1 0 1 0 0 0 0 0 that is 160 in decimal form. The bit pattern after the left shift.

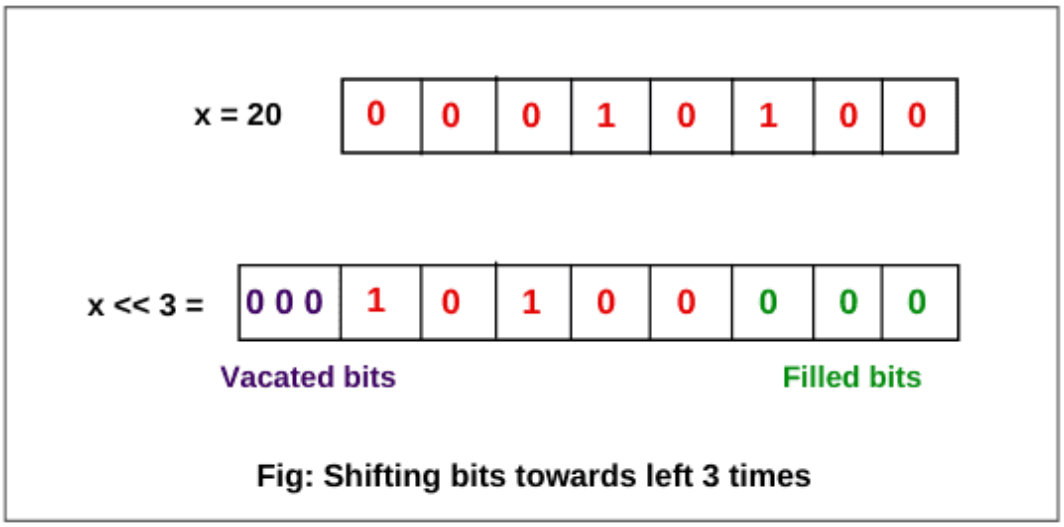

Fig: Shifting bits towards left 3 times

Key point: Shifting a value to the left, n bits, is equivalent to multiplying that value by 2^n.

For example: In the above expression, n = 3. So, 20 * 2^3 = 20 * 8 = 160.

Let's take another example based on the left shift operator.

2. If a = 45, calculate a value if a << 1.

The value 45 is represented in binary format as 0 0 1 0 1 1 0 1. If this value is shifted towards left by one position, we will get 0 1 0 1 1 0 1

0. This number in decimal form is nothing but 90 that is twice of 45.
i.e. 45 * 2^1 = 90.

Right Shift Operator in Java

The operator that shifts the bits of number towards the right by n number of bit positions is called
right shift operator in Java. The right shift operator in java is represented by a symbol >>, read as double greater than.

If we write x >> n, it means that the bits of x will be shifted towards right by n positions. There are two types of right shift
operators in java:

1. Signed right shift operator (>>)

2. Unsigned right shift operator (>>>)

Both of these operators shifts bits of number towards right by n number of bit positions.

Signed Right Shift Operator

The signed right shift operator >> shifts bits of the number towards the right and also reserves the sign bit, which is leftmost bit. A sign bit represents the
sign of a number.

If the sign bit is 0 then it represents a positive number. If the sign bit is 1, it represents a negative number.

If the number is positive, the leftmost position is filled with 0. If the number is negative, the leftmost position is filled with 1. The signed shift operator uses the same sign as used in the number before shifting of bits.

Let’s understand the concept of right shift operator with the help of an example.

1. If int x = 10 then calculate x >> 3 value.

The value of x is 10 = 0 0 0 0 1 0 1 0. Since the number is positive, the leftmost bit position will be filled with 0. Now x >> 3 will shift
the bits of x towards the right by 3 positions. The rightmost 3 bits will be lost due to shifting.

Hence, after shifting, bits of x is 0 0 0 0 0 0 0 1 that is 1 in decimal form. The bit pattern after the right shift

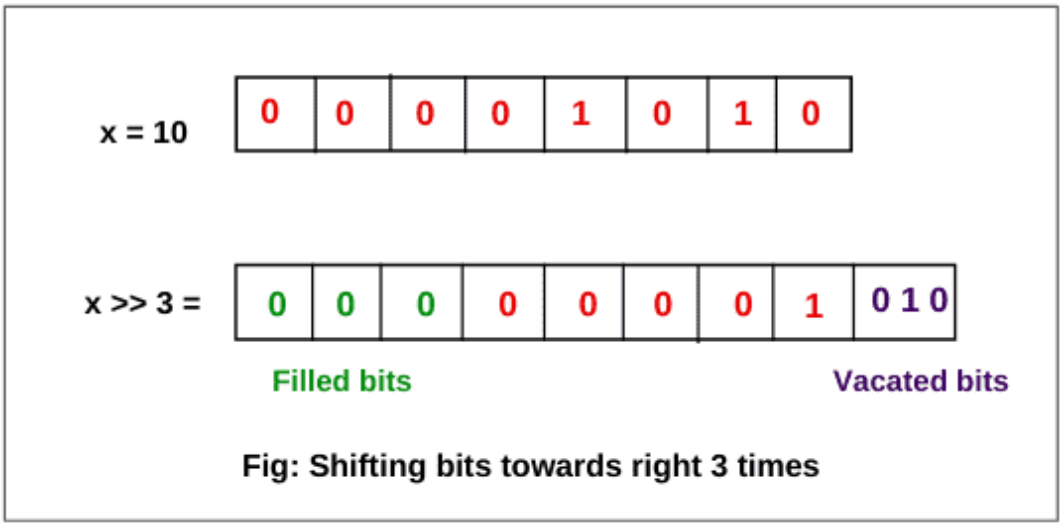

Fig: Shifting bits towards right 3 times

You will notice in this example that after performing right shift operation on a positive number 10, we get a positive value 1 as a result.

Similarly, if we perform right shift operation on a negative number, again we will get a negative value only.

Key point:

Shifting a value to the right is equivalent to dividing the number by 2^n.

```
1. int i = 10;
    int result = i >> 3; // result = 1
In this expression, 10 / 2^3 = 10 / 8 = 1.

2. int i =20;
     int result = i >> 2; // result = 5.
In this expression,  20 / 2^2 = 5.

3. int i = -20
    int result = i >> 2; // result = -5.
In this expression, -20 / 2^2 = -5.
```

Let’s take an example that is based on right shifting on a negative number.

1. If int x = -10 then calculate x >> 2 value.

The value of x is -10 that is a negative number. So, we will use 2’s complement system to represent the negative numbers

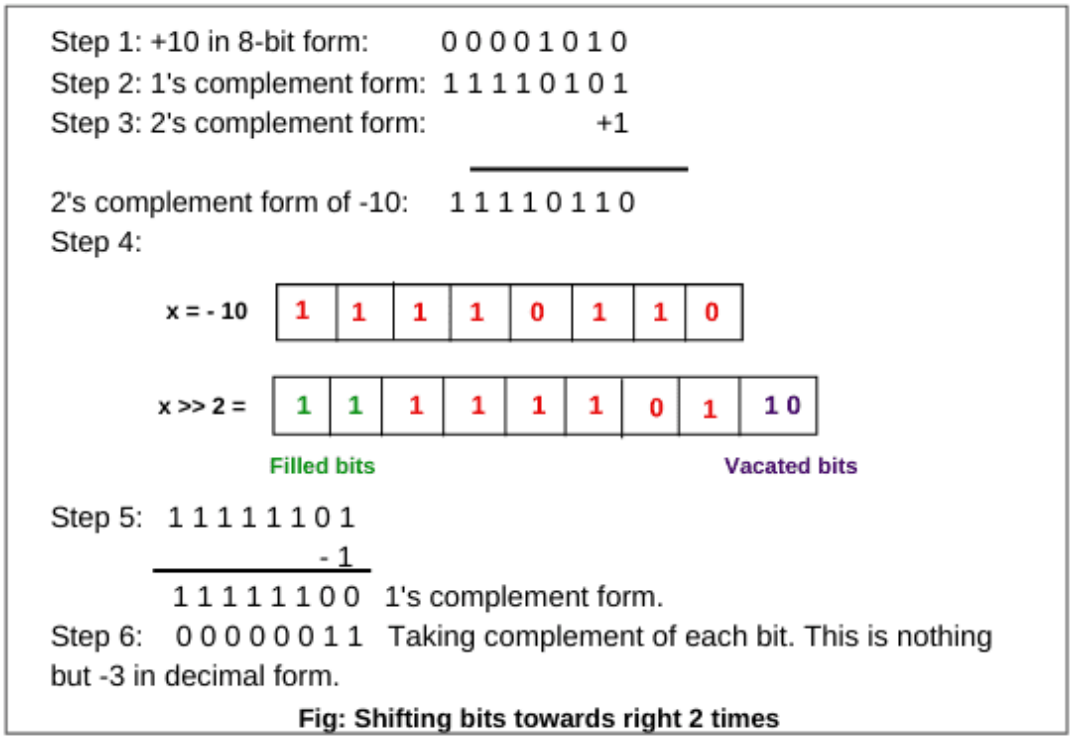

Fig: Shifting bits towards right 2 times

1. +10 in 8-bit form is 0 0 0 0 1 0 1 0.
2. Obtain the 1's complement of 0 0 0 0 1 0 1 0. The 1's complement can be obtained by simply complementing each bit of the number, that is, by changing all 0s to 1s and all 1s to 0s.

After taking complement of 0 0 0 0 1 0 1 0, the result in 1's complement form is 1 1 1 1 0 1 0 1.

1. Now add 1 to get 2's complement form. After adding 1 in LSB (Least Significant Bit), the result is 1 1 1 1 0 1 1 0. This is in 2's complement form.

Here, 1 in MSB (Most Significant Bit) represents negative number. Thus, the 2's complement form of -10 is 1 1 1 1 0 1 1 0.

1. Since the number is negative, the leftmost position is filled with two 1s. Now x >> 2 will shift the bits of x towards the right by 2 positions.

The rightmost 2 bits will be lost due to shifting. Hence, after shifting, bits of x in 2's complement form is 1 1 1 1 1 1 0 1.

1. Now convert 2's complement bits into 1's complement bits. Subtract 1 in LSB to convert 2's complement form into 1's complement. The result in 1's complement form is 1 1 1 1 1 1 0 0.

2. Take the complement of each bit to get original form of number after shifting. So, the result in original form is 0 0 0 0 0 0 1 1. This is nothing but -3 in decimal form.

Let’s verify the above result by creating a program in java.

```
package shiftOperator;
public class SignedRightShiftExample
{
public static void main(String[] args)
{
 int x = -10,
 c = 0;
 c = x >> 2;
System.out.println("x >> 2 = " +c);
 }
}
```

Unsigned Right Shift Operator

The unsigned right shift operator in java performs nearly the same operation as the signed right
shift operator in java. The unsigned right shift operator is represented
by a symbol >>>, read as triple greater than.

The unsigned right shift operator always fills the leftmost position with 0s
because the value is not signed. Since it always stores 0 in the sign
bit, it is also called zero fill right shift operator in java.

If you will apply the unsigned right shift operator >>> on a
positive number, it will give the same result as that of >>. But
in the case of negative numbers, the result will be positive because the
signed bit is replaced by 0.

Key points:

Both signed right shift operator >> and unsigned right shift operator >>> are used to shift bits towards the right.

The only difference between them is that >> preserve sign bits whereas >>> does not preserve sign bit. It always fills 0 in the sign bit whether number is positive or negative.

Let us create a program to observe the effect of various shift operators.

```
package shiftOperator;
public class UnsignedRightShiftExample {
public static void main(String[] args)
{
 int x = 10;
 int y = -10;
System.out.println("x >> 1 = " + (x >> 1));
System.out.println("x >>> 1 = " + (x >>> 1));
System.out.println("y >> 2 = " + (y >> 2));
System.out.println("y >>> 2 = " + (y >>> 2));
 }
}
```