Chapter 15: Bitwise operators

Section 15.1: Bitwise operators

Bitwise operators perform operations on bit values of data. These operators convert operands to signed 32-bit integers in [two's complement](#).

Conversion to 32-bit integers

Numbers with more than 32 bits discard their most significant bits. For example, the following integer with more than 32 bits is converted to a 32-bit integer:

```
Before: 10100110111110100000000010000011110001000001
After:              10100000000010000011110001000001
```

Two's Complement

In normal binary we find the binary value by adding the `1`'s based on their position as powers of 2 - The rightmost bit being `2^0` to the leftmost bit being `2^n-1` where `n` is the number of bits. For example, using 4 bits:

```
// Normal Binary
// 8 4 2 1
   0 1 1 0  => 0 + 4 + 2 + 0 => 6
```

Two complement's format means that the number's negative counterpart (6 vs -6) is all the bits for a number inverted, plus one. The inverted bits of 6 would be:

```
// Normal binary
   0 1 1 0
// One's complement (all bits inverted)
   1 0 0 1  => -8 + 0 + 0 + 1 => -7
// Two's complement (add 1 to one's complement)
   1 0 1 0  => -8 + 0 + 2 + 0 => -6
```

Note: Adding more `1`'s to the left of a binary number does not change its value in two's compliment. The value `1010` and `1111111111010` are both `-6`.

Bitwise AND

The bitwise AND operation `a & b` returns the binary value with a `1` where both binary operands have `1`'s in a specific position, and `0` in all other positions. For example:

```
13 & 7 => 5
// 13:     0..01101
//  7:     0..00111
//-----------------
//  5:     0..00101 (0 + 0 + 4 + 0 + 1)
```

Real world example: Number's Parity Check

Instead of this "masterpiece" (unfortunately too often seen in many real code parts):

```
function isEven(n) {
    return n % 2 == 0;
}
```

```
function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

You can check the (integer) number's parity in much more effective and simple manner:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Bitwise OR

The bitwise OR operation `a | b` returns the binary value with a `1` where either operands or both operands have `1`'s in a specific position, and `0` when both values have `0` in a position. For example:

```
13 | 7 => 15
// 13:     0..01101
//  7:     0..00111
//-----------------
// 15:     0..01111 (0 + 8 + 4 + 2 + 1)
```

Bitwise NOT

The bitwise NOT operation `~a` *flips* the bits of the given value `a`. This means all the `1`'s will become `0`'s and all the `0`'s will become `1`'s.

```
~13 => -14
// 13:     0..01101
//-----------------
//-14:     1..10010 (-16 + 0 + 0 + 2 + 0)
```

Bitwise XOR

The bitwise XOR (*exclusive or*) operation `a ^ b` places a `1` only if the two bits are different. Exclusive or means *either one or the other, but not both*.

```
13 ^ 7 => 10
// 13:     0..01101
//  7:     0..00111
//-----------------
// 10:     0..01010  (0 + 8 + 0 + 2 + 0)
```

Real world example: swapping two integer values without additional memory allocation

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a is now 22 and b is now 11
```

Section 15.2: Shift Operators

Bitwise shifting can be thought as "moving" the bits either left or right, and hence changing the value of the data operated on.

Left Shift

The left shift operator `(value) << (shift amount)` will shift the bits to the left by `(shift amount)` bits; the new bits coming in from the right will be `0`'s:

```
5 << 2 => 20
//  5:      0..000101
// 20:      0..010100 <= adds two 0's to the right
```

Right Shift (*Sign-propagating*)

The right shift operator `(value) >> (shift amount)` is also known as the "Sign-propagating right shift" because it keeps the sign of the initial operand. The right shift operator shifts the `value` the specified `shift amount` of bits to the right. Excess bits shifted off the right are discarded. The new bits coming in from the left will be based on the sign of the initial operand. If the left-most bit was `1` then the new bits will all be `1` and vice-versa for `0`'s.

```
20 >> 2 => 5
// 20:      0..010100
//  5:      0..000101 <= added two 0's from the left and chopped off 00 from the right

-5 >> 3 => -1
// -5:      1..111011
// -2:      1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Right Shift (*Zero fill*)

The zero-fill right shift operator `(value) >>> (shift amount)` will move the bits to the right, and the new bits will be `0`'s. The `0`'s are shifted in from the left, and excess bits to the right are shifted off and discarded. This means it can make negative numbers into positive ones.

```
-30 >>> 2 => 1073741816
//       -30:      111..1100010
//1073741816:      001..1111000
```

Zero-fill right shift and sign-propagating right shift yield the same result for non negative numbers.