BITWISE OPERATORS

There are a number of ways to manipulate binary values. Just as you can with decimal numbers, you can perform standard mathematical operations - addition, subtraction, multiplication, division - on binary values (which we'll cover on the next page). You can also manipulate individual bits of a binary value using **bitwise operators**.

Bitwise operators perform functions bit-by-bit on either one or two full binary numbers. They make use of logic operating on a group of binary symbols. These bitwise operators are widely used throughout both electronics and programming.

Complement (NOT)

The complement of a binary value is like finding the exact opposite of everything about it. The complement function looks at a number and turns every *1* into a *0* and every *0* becomes a *1*. The complement operator is also called **NOT**.

For example to find the complement of 10110101:

```
COPY CODE

NOT 10110101 (decimal 181)
   -------- =
   01001010 (decimal 74)
```

NOT is the only bitwise operator which only operates on a single binary value.

OR

OR takes two numbers and produces the **union** of them. Here's the process to OR two binary numbers together: line up each number so the bits match up, then compare each of their bits that share a position. For each bit comparison, if **either or both** bits are 1, the value of the result at that bit-position is 1. If both values have a 0 at that position, the result also gets a 0 at that position.

The four possible OR combinations and their outcome are:

- 0 OR 0 = 0
- 0 OR 1 = 1
- 1 OR 0 = 1
- 1 OR 1 = 1

For example to find the 10011010 OR 01000110, line up each of the numbers bit-by-bit. If either or both numbers has a *1* in a column, the result value has a *1* there too:

```
COPY CODE

  10011010
OR 01000110
  -------- =
  11011110
```

Think of the OR operation as binary addition, without a carry-over. 0 plus 0 is 0, but 1 plus anything will be 1.

AND

AND takes two numbers and produces the **conjunction** of them. AND will only produce a *1* if both of the values it's operating on are also *1*.

The process of AND'ing two binary values together is similar to that of OR. Line up each number so the bits match up, and then compare each of their bits that share a position.

For each bit comparison, if **either or both** bits are *0*, the value of the result at that bit-position is *0*. If both values have a *1* at that position, the result also gets a *1* at that position.

The four possible AND combinations and their outcome are:

- 0 AND 0 = 0
- 0 AND 1 = 0
- 1 AND 0 = 0
- 1 AND 1 = 1

For example, to find the value of 10011010 AND 01000110, start by lining up each value. The result of each bit-position will only be *1* if both bits in that column are also *1*.

```
COPY CODE

   10011010
AND 01000110
   -------- =
   00000010
```

Think of AND as kind of like multiplication. Whenever you multiply by 0 the result will also be 0.

XOR

XOR is the **exclusive OR**. XOR behaves like regular OR, except it'll **only** produce a *1* if **either one or the other** numbers has a *1* in that bit-position.

The four possible XOR combinations and their outcome are:

- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0

For example, to find the result of 10011010 XOR 01000110:

```
COPY CODE

   10011010
XOR 01000110
   -------- =
   11011100
```

Notice the 2nd bit, a *0* resulting from two *1*'s XOR'ed together.

Bit shifts

Bit shifts aren’t necessarily a bitwise operator like those listed above, but they are a handy tool in manipulating a single binary value.

There are two components to a bit shift - the **direction** and the **amount** of bits to shift. You can shift a number either to the **left or right**, and you can shift by one bit or many bits.

When shifting to the right, one or more of the least-significant bits (on the right-side of the number) just get cut off, shifted into the infinite nothing. Leading zeros can be added to keep the bit-length the same.

For example, shifting 10011010 to the right two bits:

```
COPY CODE

RIGHT-SHIFT-2 10011010 (decimal 154)
        -------- =
        00100110 (decimal 38)
```

Shifting to the left adds pushes all of the bits toward the most-significant side (the left-side) of the number. For each shift, a zero is added in the least-significant-bit position.

For example, shifting 10011010 to the left one bit:

```
COPY CODE

LEFT-SHIFT-1 10011010 (decimal 154)
        -------- =
        100110100 (decimal 308)
```

That simple bit shift actually performs a relatively complicated mathematical function. Shifts to the left *n* bits **multiplies a number by 2^n** (see how the last example multiplied the input by two?), while a shift *n* bits to the right will do an **integer divide by 2^n**. Shifting to the right to divide can get weird - any fractions produced by the shift division will be chopped off, which is why 154 shifted right twice equals 38 instead of 154/4=38.5. Bit shifts can be a powerfully fast way to divide or multiply by 2, 4, 8, etc.

These bitwise operators provide us most of the tools necessary to do standard mathematical operations on binary numbers.

Source: https://learn.sparkfun.com/tutorials/binary