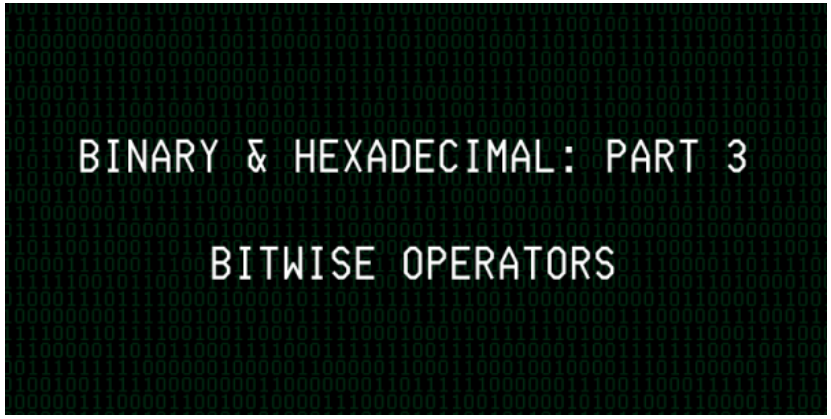

Binary and Hexadecimal: Part

3 - Bitwise Operators

2672 words. Time to Read: About 26 minutes.

In the last binary/hex post, we worked through converting numbers between base 10 (decimal) numbers, base 2 (binary) numbers, and base 16 (hexadecimal) numbers. In this post, we're introducing some new operations you can do to numbers now that you know that they are all secretly made up of ones and zeros.

Math -> Computing

Before we dive in to these operations, lets look at some terminology that's used in Computer Science that you might see in other resources about this topic.

bit:

Numbers (as well as any other data represented inside a computer) are made up of ones and zeros. After reading the previous two posts, you know

that this format is called called binary. What you might *not* know is that each one of these ones or zeros is called a **bit**, and it is the smallest unit of information stored by a computer. Since there are only two states (1 or 0), computers can store this information as electrical or magnetic measurements: a bit can be either "on" (1) or "off" (0). Think of it like if you wanted to store a number by turning on and off a row of flashlights. The pattern of on and off would represent the overall data/number you're trying to represent.

byte:

For purely historical reasons, computers store bits in groups of 8. Each group of 8 bits is called a **byte**. This word is probably familiar even if you've never encountered binary before, because you're probably familiar with terms like "megabyte", "gigabyte", "terabyte", and beyond. Our phones are never shy about telling us when we're running out of "Gee-Bees" (Gbs). I won't go into the math behind the various prefixes, but it's good to know where it comes from. Suffice it to say that we've standardized on processing things in **bytes**. Because **bytes** values can be between `0b00000000` and `0b11111111` (that is, 0 and 255), the number 256 might also look familiar in computery contexts. It's a handy range to store data in.

nibble:

My personal favorite in the binary space is a **nibble**. A **nibble** is half of a

byte, or 4 bits. Because of this, you'll have nibbles with values between `0b0000` and `0b1111` (0-15). This is 16 possibilities. The particularly astute among you might recognize 16 possibilities as the *exact* number that can be stored in one hexadecimal digit. For this reason, converting one hex digit will yield you a **nibble** of binary. Chunking binary into 4 helps me keep track as I convert between binary and hex and back.

Bitwise Operations

OK! Now that we know what a bit is, I can introduce **bitwise operations**. If you're currently picturing a 1 and a 0 hanging out, both with giant grey beards, saying wise things... me too. You're not alone.

But bitwise operations are more than just some wise old digits. They're operations that can only computed by comparing the bits that make up two numbers. Let's go through a few. If you're familiar with logic/booleans at all, you'll probably see some similarities to those.

NOT

NOT is also referred to as the complement. It's a *unary* operation, which means you only do it to one number. It's similar-ish to a negative sign in

normal math.

In order to NOT a number, you take each one of its bits and flip the value. If it's a 1, you make it a 0. If it's a 0, you make it a 1!

```
NOT 0011 (decimal 3)
  = 1100 (decimal 12)
```

If you're working with decimal numbers, you can take a little shortcut. If you know the number of bits you're using, the result of NOT-ing your number is `max value of bits - your number`.

For example: If we're using a **byte** to represent our number, and we've got 200 as our number, the max value for a byte is 255. So:

```
NOT 200

= 255 - 200
= 55!
```

Sometimes the shortcut helps. Sometimes it's easier to convert everything to binary and flip the bits.

AND

You can think of AND in the following sentence: The result is only *on* if both of the inputs is *on*. If either one of the inputs is *off*, then the result is *off*. It's kind of like if you had a water hose with a valve at the spigot and a sprayer nozzle at the end of the hose. Water only flows when the spigot is open AND the sprayer is open. If the sprayer is open but the spigot is off, then nothing comes out and you look silly. If the spigot is on but the sprayer is closed, then nothing comes out (and if you've got a leaky sprayer, you get wet).

```
.   01001110  (decimal 78)
AND 11000010  (decimal 194)
=   01000010  (decimal 66)
```

It's essentially like you're multiplying each set of bits one by one. `0 * 1 = 0`, `1 * 1 = 1`, `0 * 0 = 0`, etc.

In some programming languages, you'll see AND shown as a single `&` operator.

In math notation, they use a ^ symbol for "intersection", which makes sense if you think of this as a Venn Diagram between the bits of the two

numbers. The result of an AND is all of the bits that are in the center, overlapping section of the graph, shared between the two numbers. 78 has `{2, 4, 8, 64}` as its binary digits, and 194 has `{2, 64, 128}`. Inside the overlapping "intersection" between the two is `{2, 64}`, so the result is 66!

AND is really useful for doing a thing called "bit-masking" which is where you want to take a number and "mask off" certain digits so that only certain digits come "through the mask."

For example, let's say I'm going to ask you for a number and then I want to tell you if it has a 8's place bit set in it. What I can do is "mask off" all of the other bits by blocking them, only letting the 8's place bit through. Then I can check if it came through or not!

```
.   01001110 (decimal 78)
AND 00001000 (decimal 8)
=   00001000 # Yes!  There *was* a 1 in the 8's place!

.   11000010 (decimal 194)
AND 00001000 (decimal 8)
=   00000000 # Nope!  No 1 in the 8's place
```

So, for this mystery number you gave me, once I've AND-ed it with 8, I can check to see if the result is 8, which means that you *did* have a 1 in the 8's

place, or 0, which means your number *didn't* have a 1 in the 8's place.

Clear as mud? This whole "getting your brain to switch back and forth from decimal to binary" thing is a skill that comes with practice. Try using AND to only show me the right-most two binary digits of 254. What is the result as a decimal number?[1]

One other neat thing you can do with AND is check if a number is even or odd! Think about the binary. Each digit is a power of 2, so a number can *only be odd* if it has a 1 in the far right digit: the ones place! Soooooo, in order to check if a number is odd, we can AND our number with 1!

```
.   01010101  (decimal 85)
AND 00000001  (decimal 1)
=   00000001  (decimal 1 means ODD!)

.   01010100  (decimal 84)
AND 00000001  (decimal 1)
=   00000000  (decimal 0 means EVEN!)
```

`683 AND 1 = 1`, so 683 is odd! This may not seem very useful when doing it by hand—why not simply look at the number and say if it's even!—but when you start to do things in programming, it's a little harder to "just look at it."

OR

The OR is a similar to philosophy to the AND, but it's much more generous. The result of an OR will be *on* if *either of the two inputs is on*. It's more like a bucket with two holes in it that are being opened and closed. If either one or both of the holes is open, water will flow. It's only when *neither* of the holes is open that the water stops.

```
.   01110010 (decimal 114)
 OR 10110001 (decimal 177)
=   11110011 (decimal 243)
```

In some programming languages, this is represented by the `|` single vertical pipe symbol.

In mathematical symbols, you'll see it as `∨`, the downwards caret, also known as the "union". In Venn Diagram terms, this represents everything inside the left circle, the right circle, and the middle overlap area. If there is a bit present in either or both of the sets, then it gets carried through. 114 has `{2, 16, 32, 64}` and 177 has `{1, 16, 32, 128}`. Combining all of the numbers from both sets gives `{1, 2, 16, 32, 64, 128}` which equals 243!

You can use this to “set” a particular bit in an unknown number. Let’s say that I’m going to ask you for a number, and regardless of what it is, I want to make sure I get the 4’s place bit set in my result. What I need to do is to OR your number with a 4!

```
.   01010001 (decimal 81)
 OR 00000100 (decimal 4)
 =  01010101 (decimal 53, and the 4's bit is set!)

.   00001111 (decimal 15)
 OR 00000100 (decimal 4)
 =  00001111 (decimal 15.  4's bit was aleady set, so no change!)
```

XOR

This one is a little more complicated than the rest, but not scarily so. The XOR, also known as the “exclusive OR”, will output a 1 if either of the inputs is a 1, but *not both*. If both are 1 or both are 0, it will output 0. More simply, it outputs 0 when both inputs are the same and 1 if they’re different.

```
.   01001110 (decimal 78)
```

```
XOR 11001100 (decimal 204)
=   10000010 (decimal 130)
```

In some programming languages, the ^ symbol is used to represent XOR (a little confusingly since that's the mathematical symbol for AND). However, XOR makes up for it in mathematical notation with the coolest math symbol: the ⊕ circled plus symbol. I think a huge opportunity was missed to call it a "plussircle."

XOR is really useful for "toggling" a bit inside a number. In order to toggle a bit, changing it's value no matter what it is, you XOR your number with a number with a 1 in the location you want to toggle:

```
.   11010101 (decimal 213)
XOR 00010000 (decimal 16, let's flip the 16's place bit!)
=   11000101 (decimal 197, the same number as 213, except for the
              bit that's been flipped!)
```

One other extremely neat usage of XOR is encrypting things! The cool thing about XOR is that it *undoes itself*.

```
.   11010101 (decimal 213)
XOR 01101100 (decimal 108)
```

```
=   10111001 (decimal 185, a totally different number)
XOR 01101100 (decimal 108, XOR-ing with the same number again)
=   11010101 (decimal 213, the same number we started with!)
```

So, if you want to encrypt a number (or a sentence encoded into binary, but that's a whole 'nother article), you pick a number to be your "key," and use it to encode your number. My safe code is 10, 20, 30, 40, 50 (super secure!). I'm going to use 42 as my key. Now, if my buddy already knows the key number, I can tell him that my safe code is:

```
.   00001010 (decimal 10)
XOR 00101010 (decimal 42)
=   00100000 (decimal 32)

.   00010100 (decimal 20)
XOR 00101010 (decimal 42)
=   00111110 (decimal 62)

.   00011110 (decimal 30)
XOR 00101010 (decimal 42)
=   00110100 (decimal 52)

.   00101000 (decimal 40)
```

```
XOR 00101010 (decimal 42)
=   00000010 (decimal 2)

.   00110010 (decimal 50)
XOR 00101010 (decimal 42)
=   00011000 (decimal 24)
```

I can tell my buddy my safe code is: 32, 62, 52, 2, 24... (*wink*). And my buddy can take those numbers, XOR them with the key, and get 10, 20, 30, 40, 50 back out!

Left and Right Shifts

Shifts take the pattern `67 >> 3` or `12 << 4`. Essentially, you're taking the binary number and shifting the pattern of bits left or right by that amount. There are several strategies for what to do at either end of the binary number as digits get shifted on and off.

Logical Shift

The most simple case (and what some programming languages do) is called a "logical shift," which is where we just use 0 any time we need a new digit shifted in. Let's see an example:

```
.   00011100 (decimal 28)
<<          4
=   11000000 (decimal 192)

.   11100000 (decimal 224)
>>          3
=   00011100 (decimal 28)
```

See how we keep the same relative pattern of the bits, but they get "shifted" over a certain amount? If we're dealing with bytes, bits that go off the end get dropped, and every time we shift over, we have to bring in an additional 0 at the "input" end. However, you could also treat the binary number as having as many bits as it needs. In that case, numbers wouldn't get dropped off the left side when we're shifting left. We'd just get bigger and bigger. This would have the effect of multiplying or dividing by two. A left shift (<<) is equivalent to doubling your value, and a right shift (>>) is equivalent to halving (and rounding down!) your value.

There are other types of shifts that are less commonly used: "Arithmetic shift" and "circular shift". I'll mention them briefly for completeness, but don't stress too hard about them.

Arithmetic Shift

In arithmetic shifts, the shift left is still the same. Zeros come in as needed. However, the right shift keeps the left-most bit the same even after a shift. This is because, sometimes, it's useful to use that value as a "sign" bit which specifies if your number is negative or positive. In that case, the left-most bit is the sign-bit, and the other 7 bits in the byte represent the numerical value.

```
.   11101100 (decimal 236)
<<          2
=   10110000 (decimal 176)

.   11101100 (decimal 236)
>>          2
=   11111011 (decimal 251, see how the 1 on the left side
                causes us to shift in more 1's?)
```

Circular Shift

A circular shift is when you recycle the bit that drops off and use it to refill the bit coming in:

```
.   11101100 (decimal 236)
<<          2
```

```
=   10110011 (decimal 179.  See the two ones from the left brougl
              in on the right?)

.   11101100 (decimal 236)
>>         2
=   00111011 (decimal 59)
```

Feeling a "Bit" More Powerful?

Bitwise operations have lots of neat little applications. You can use bits as flags specifying different unrelated options and turn them on or off as necessary. You can use shifting for CPU-quick doubling and halving, and you can use different masking and shifting techniques to focus on specific bits inside a number!

I'll have more posts soon about how this new knowledge can be used in programming contexts.

There are also a couple side-topics like "Endianness," which is just a fancy and intimidating way of saying that sometimes computers are programmed to expect the bits with the biggest value (Most Significant Bit: MSB) going left to right rather than right to left! For reference, all of the examples

you've seen here are "big-endian," with the MSB coming first. (Get it? big-endian = the "big end" first? I don't know why they had to make it sound so scary.) We'll talk about Endianness in a future article a little bit more.

1. `254 = 0b11111110`. Select the right-most two digits by AND-ing with `0b00000011`. `0b11111110 AND 0b00000011 = 0b00000010 (2)`. ↩

Author: Ryan Palo | Tags: computer-science basics beginner |

Featured Posts

Jan 18, 2017
Assert Not Magic
Nov 23, 2020
This is How You Lose the Time War: A Review
Nov 16, 2020
Building an Automatic Measuring Table: Part 0
Nov 11, 2020
Awestruck by C Structs

Tags

python | django | rails | algorithms | editors | emacs | ai | r | tools | javascript | not-magic | soft-skills | tricks | vscode | vim | sublime | puzzle | bash | math | teaching | ruby | jekyll | git | physics | fun | scientific | pythonic | functional | vue | es6 | tutorial | exercise | fitness | sysadmin | linux | html | seo | social | metaprogramming | css | front-end | animations | p5 | beginner |

encouragement | screencast | showdev | interview | performance | workflow | best-practice | oss | static-site | big-o | linked-lists | design-intent | struct | browser | focus | productivity | codenewbie | reflection | update | preview | gtd | bullet-journal | data-science | singleton | basics | powershell | terminal | security | scripting | itertools | shell | quicktip | cheatsheet | fish | art | computer-science | learning | exercism | challenge | story | standard-library | defaultdict | generative | advanced | negotiating | julia | practical | rust | iterators | cli | binary | readability | types | awk | cron | dotfiles | java | projects | brainstorming | steganography | typescript | models | architecture | devjournal | resume | career | netlify | automation | c | structs | robotics | mechanical | woodworking | books | bugs |

Like my stuff? Have questions or feedback for me? Want to mentor me or get my help with something? Get in touch! To stay updated, subscribe via RSS