Open in app

Joshua Parker

Follow 9 Followers About

Practical bit manipulation in JavaScript

Joshua Parker Oct 4, 2018 · 12 min read

An introduction to bitwise operators (and how to put them to work).

"person holding calendar at January" by Brooke Lark on Unsplash

What follows, is a short introduction to JavaScript's handling of binary numbers and bitwise logical operations, followed by 13 bit-sized lessons. Enjoy!

The strange (but necessary) bit

All numbers in JavaScript are 64 bit floating point. However, when bitwise operators are used, JavaScript, internally converts the them into truncated 32 bit signed integers in 2's complement, then performs the bitwise operation on them before converting the result back to a 64 bit double precision number. You can think of it like this:

```
5 << 1                                  // valid expression
00000000000000000000000000000101 << 1   // (whats basically going on
00000000000000000000000000001010        // …behind the scenes)
10                                      // return value
```

If you want to work *in* binary, you might expect there to be a nice way to represent numbers in binary, and… you would be right! ES6 introduced binary literals, which take the form `0b` followed by a series of 1's and 0's, e.g., `0b1001` represents the number represented by the decimal integer 9. Like all numeric literals in JavaScript, they simply *express* 64 bit double precision numbers in another way, for convenience. They are certainly no "closer to the metal", and JavaScript still needs to do the actual conversion to/from 32 bit integers behind the scenes. So there is no time saved by using binary literals over their decimal or hexadecimal equivalents. That being said, bitwise solutions to a given problem are typically faster and more space-efficient than their non-bitwise counterparts.

—

Note, if we desired to work in base 2 pre-ES6, the best we could do was parse a binary string in base(radix) 2, then perform the bitwise operation on the return value.

```
0b101                      // using ES6 binary literal
parseInt('101',2)          // before binary literals
```

Of course, there *are* still times when its useful to convert to/from binary string representations.

```
const b2d = x => parseInt(x,2)     // binary to decimal
const d2b = x => x.toString(2)     // decimal to binary

b2d(101)                           // 5
d2b(5)                             // 101
```

13 things to know [and love] about bits.

1. Bitwise left shift `<< x` is equivalent to multiplying by 2**x

`<<` is the left shift operator. It shifts a number to the left by the specified number of bits, discarding bits that are shifted off.

```
0b0001 << 1     // 2 (0010)
0b0001 << 2     // 4 (0100)
0b0001 << 3     // 8 (1000)
```

Recall, that for base 10, every time we move a decimal place to the left we multiply by 10, so if we move x digits, we multiply by `10**x` .

```
0    0    1    0        # 10
10^3 10^2 10^1 10^0

1    0    0    0        # 10 << 2 = 10 x 10^2 = 1000
10^3 10^2 10^1 10^0
```

This works for any base! For base 2, left shifting x digits to the left is equivalent to multiplying by `2**x` .

```
0b0010              // 0010 = 2
0b0011              // 0011 = 3

0b0010 << 1         // 0100 = 4
0b0011 << 1         // 0110 = 6

0b0010 << 2         // 1000 = 8
0b0011 << 2         // 1100 = 12
```

2. Bitwise right shift `>> x` is equivalent to floor division by 2**x

If a left shift by x results in a multiplication by `2**x` , we can expect a right shift by x bits to result in a division by `2**x` , which it does,… almost. This operation results in a *floor* division by `2**x` . The rounding occurs because we are shifting the right-most bits off and discarding them.

```
0b1001              // 1001 = 9
0b1000              // 1000 = 8

0b1001 >> 1         // 0100 = 4 = Math.floor( 9/2 )
0b1000 >> 1         // 0100 = 4 = Math.floor( 8/2 )

0b1001 >> 2         // 0010 = 2 = Math.floor( 9/4 )
0b1000 >> 2         // 0010 = 2 = Math.floor( 8/4 )

0b1001 >> 3         // 0001 = 1 = Math.floor( 9/8 )
0b1000 >> 3         // 0001 = 1 = Math.floor( 8/8 )
```

3. Turn bits off with `& 0`

(and leave the rest unchanged with `& 1`)

Given two binary digits, or *bits*, `a` and `b` , `a AND b` yields a `1` if *and only if* both bits evaluate to `1` . Another way to say the same thing is, `a & 0 -> 0` and `a & 1 -> a` . Yet another way, is `& 0` turns a bit off (if it is not already off) and `& 1` does, well,.. nothing at all.

```
a   b   a & b
0   0   0          // a & 0 = 0
0   1   0          // a & 1 = a
1   0   0          // a & 0 = 0
1   1   1          // a & 1 = a
```

With Javascript, we are not operating on single bits, but 32 bits at a time, and our concern is with turning off some bits while leaving others alone. To this end, we can define another 32 bit number, called a mask, such that for every bit we want to turn off we set the corresponding bit in the mask to 0, and set all other bits to 1. When we apply the mask with the bitwise AND operator, input bits that correspond to 0's in mask will be turned off (`& 0` turns bits off), and remaining bits will be unchanged (`& 1` does nothing at all).

```
      v
    101 (input)
AND 011 (mask)
-------
    001 (result)
```

In JavaScript, using our helper function, it looks like this:

```
a =    0b101         // 101
mask = 0b011         // 011 mask 3rd bit
a = a & mask         // 001 turn 3rd bit off

// turn ith bit off
const turnOFF = (num, i) => num & ~(1 << i - 1)
```

4. Turn bits on with ` |1 `

(and leave the rest unchanged with ` | 0`)

`a OR b` yields a `1` provided either *bit,* `a` or `b`, evaluates to `1`. This lazy gate-keeping can be expressed as `a | 1 -> 1` and `a | 0 -> a`, and in turn as `| 1` turns a bit on and `| 0` does not a thing.

```
a   b   a | b
0   0   0          // a | 0 = a
0   1   1          // a | 1 = 1
1   0   1          // a | 0 = a
1   1   1          // a | 1 = 1
```

As we did with bitwise `AND`, we define a 32 bit mask to turn some bits on while leaving others unchanged. This time the mask bits that correspond to input bits we want to turn on, will be set to 1, and remaining bits set to 0. When we apply the mask with the bitwise OR operator, input bits that correspond to 1's in mask will be turned on (`| 1` turns bits on), and remaining bits will be unchanged (`| 0` does nothing).

```
     v
   010 (input)
OR 100 (mask)
-------
   110 (result)
```

And in JavaScript:

```
a =    0b010         // 010
mask = 0b100         // 100 mask 3rd bit
a = a | mask         // 110 turn 3rd bit on

// turn ith bit on
const turnON = (num, i) => num | (1 << i - 1)
```

5. Flip bits with ` ^1 `

(and leave rest unchanged with `^0`)

`a XOR b` yields a `1` if either bits, `a` or `b`, *but not both*, evaluate to `1`. In this way, `XOR` "eXcludes" the cases in which the operands are equal in value, and it follows that `a ^ 1 -> !a` and `a ^ 0 -> a`, or `^ 1` toggles a bit, and `^ 0` does, surprise, nothing. `!a` used here to denote the toggled bit value of a.

```
a   b   a ^ b
0   0   0           // a ^ 0 = a
0   1   1           // a ^ 1 = !a
1   0   1           // a ^ 0 = a
1   1   0           // a ^ 1 = !a
```

Now, lets define a 32 bit mask to *toggle* a subset of one or more bits while leaving the rest unchanged. Bits that correspond to input bits we want to toggle, will be set to 1, and remaining bits set to 0. When we apply the mask with the bitwise XOR operator, input bits that correspond to 1's in mask will be toggled, and remaining bits will be left alone.

```
      v
    110
XOR 100 (mask)
-------
    010
```

In Javascript:

```
a =    0b110
mask = 0b100         // mask 3rd bit
a = a ^ mask         // flip 3rd bit

// flip ith bit
const flip = (num, i) => num ^ (1 << i - 1)
```

6. Query a bit with ` & 1 `

(by turning rest of bits off with `& 0`)

We know that `x & 1` leaves the bit `x` unchanged. Put another way, the state of a bit x *is* `x & 1`. For multiple bits, we can infer the state of a bit by turning all other bits off and observing that state of the bit in question is 1 only if the result is > 0.

```
// input
a = 0b101

// masks
1st_bit = 0b001
2nd_bit = 0b010
3rd_bit = 0b100

// queries
a & 1st_bit > 0       // true (1st bit is on)
a & 2nd_bit > 0       // false (2nd bit is off)
a & 3rd_bit > 0       // true (1st bit is on)
```

Of course, inside a JavaScript conditional we can omit the comparison operator and coerce to true/false.

```
mask = 0b100
if (0b110 & mask) {
    console.log('third bit is on')
}
```

And to query the ith bit, a left shift by `i — 1` will yield a mask with ith bit set to 1.

```
// query ith bit
const query = (num, i) => num & (1 << i - 1)
```

7. Slide a '1' into place with ` 1 << n-1 `

(slide a '0' into place by toggling the result of sliding a 1 into place.)

Its very often the case that we want to target nth bit in a number programmatically, e.g., to toggle it or set it. By far, the easiest way to do this is to left shift `1` by `n-1` , producing a mask with the nth bit set to 1. We can then use this mask to toggle or set the corresponding bit.

```
a = 0b10001          // 00001     input
mask = (1 << 4)      // 10000     mask 5th bit with 1
a = a ^ mask         // 10001     toggle 5th bit
a = a | mask         // 00001     turn 5th bit on

// slide a 1 into ith place (mask ith bit to 1)
const mask = (i) => (1 << i - 1)

// slide a 0 into ith place (mask ith bit to 0)
const mask = (i) => ~(1 << i - 1)
```

8. Create n 1's with ` (1 << n) - 1 `

Previously, we created a mask and used it to toggle and set the 5th bit with XOR and OR respectively, but what if we want to turn the 5th bit on. We can do this with AND if we had a bit mask of the form `01111` . We can use bitwise not, but in the event that our tilda key is broken, this pattern can also be produced by XOR-ing `10000` with `11111` , ie toggle all the bits. We know how to slide a 1 into nth position, i.e., `1 << n-1` , but how do we produce a mask with n 1's. Answer: by shifting 1 into the `n+1` position and subtracting 1, i.e., `(1 << n) — 1` . Looks like this:

```
a = 0b11001              // 11001     input
mask = (1 << 4)          // 10000     mask 5th bit with 1
ones = (1 << 5) - 1      // 11111     generate all ones
mask = mask ^ ones       // 01111     inverted mask
a = a & mask             // 01001     turn 5th bit off!
```

This works because the base 2 number with 1 at nth position represents the decimal number `2**(n-1)` , so `2**(n-1)-1` is the highest number that you can represent with `n-1` bits (because we start at zero). That number corresponds to `n-1` 1's!

```
n         2^(n-1)   2^(n-1)     2^(n-1)-1
2         2         10          01
3         4         100         011
4         8         1000        0111
```

9. Use `~i` to flip all the bits and use `~i + 1` to negate a number

Two's complement, which JavaScript used to represent negative numbers, is equivalent to flipping the bits, and adding 1. It is important to distinguish it from bitwise not `~i` . The bitwise NOT, also called the complement, or one's complement, is the result of just flipping the bits. Note, we could have used it in the previous example to flip bits in the mask and save ourself some steps.

```
a = 0b11001              // 11001     input
mask = ~(1 << 4)         // 01111     mask 5th bit with 1
a = a & mask             // 01001     turn 5th bit off!
```

(Want to learn more about two's complement? Start here. Trust me.)

10. Set a bit by first clearing it with `& 0`, then setting it with `| val`

So far, we can turn a bit on, off and toggle it, but in order to set it to a value `v` , where v can be `0` or `1` , we must first reset the bit to a known value, `0` or `1` . If we reset to `0` , we can then set to `v` with `OR v` . If we reset to `1` , we can then set to `v` with `AND v` . Either works. I like the latter.

```
// set ith bit to val
const set = (num, i, val) => {
  let mask = ~(1 << i - 1)        // mask ith bit
  num = num & mask                // reset ith to 1
  val = val << i - 1              // shift val by i
  num = num | val                 // set ith to val
}
```

Of course if we want to set multiple bits, we have to mask multiple bits. A common interview problem takes the following form: You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to insert M into N such that M starts at bit j and ends at bit i. Compare the solution to the set function above.

```
const insert_m_into_n(n, m, i, j) {

  // mask bits i thru j
  ones = (1 << (j - i + 1)) - 1
  mask = ~(ones << i - 1)

  // reset bits i thru j to one
  n = n & mask

  // shift m by i
  m = m << i - 1

  // merge i thru j
  return n | m
}
```

11. Check if number is odd with `& 1`

(odd if the last bit is 1)

An integer is odd if the last bit is 1. So query it with `& 1` . If result is 1, then the last bit is 1 → number is odd. If result is 0 then the last bit is 0 → number is even.

```
    00101011        01100010
&   00000001    &   00000001
    --------        --------
    00000001        00000000
```

In JavaScript `0` and `1` are falsy / truthy values so we can coerce them to `true / false` with `!`

```
const is_even = (x) => {
    return !(x & 1)
}
const is_odd = (x) => {
    return !!(x & 1)
}
```

12. Swap bits with 3 XORs

If we want to swap two integers without creating a temp var, and without the spread operator (thats cheating), there is a very cool bit of bit magic that will accomplish this: three XOR's. Can step through it to see how it works, but really it just works.

```
              a     b     a     b
a  |  b  | a^b | a^b | a^b |
0     0     0     0     0     0
0     1     1     0     1     0
1     0     1     1     0     1
1     1     0     1     1     1
```

JavaScript:

```
let a = 0b0011          // a = 0011 (3)
let b = 0b0101          // b = 0101 (5)

a = a^b                 // a = 0110
b = a^b                 // b = 0011
a = a^b                 // a = 0101

a.toString(2)           // a = 0101 (5)
b.toString(2)           // b = 0011 (3)
```

13. Filter a set with with `>> 1` and `& 1`

One of the most useful things to do with bits is use them as booleans. Real Booleans in JavaScript are not 1 bit, they are 32 bits or maybe 64. I don’t really know. But definitely not 1, but bits are (1 bit that is), so lets use them instead. Say you have a set of things. Now lets say you have lots of subsets. Lots of ways to handle this. Could duplicate the data, or maintain a dictionary for each subset, where keys map to true/false values that represent inclusion in the base set. Or represent subsets as boolean arrays. Thats not bad. Here is another way: you could represent each subset as just a single number, that when expressed in base 2, individual bits correspond to items in the set, with 1’s representing inclusion, and 0’s non-inclusion!

```
                            i
                            v
set:    {  'a',  'b',  'c',  'd'  }
msk:        0     1     0     1
subset: {        'b',        'd'  }
```

Then if we want to bake the subset. We can filter the base set by repeatedly shift and test the first bit for inclusion with `>> 1` and `& 1` . Recall that `& 1` is used to query the first bit and `>> 1` shifts the first bit off, moving the second bit into the first position.

This is the bitwise equivalent of iterating through an array. bits are not indexed, so we can't increment a pointer. We could shift a mask bit left and query each bit in turn, but its easier to incrementally shift bits off and continue to query only the first bit.

```
i = set.length - 1
subset = new Set()
while (msk > 0) {
 if (msk & 1) subset.add(set[i])
 msk >> 1
 i -= 1
}
```

References

1. https://graphics.stanford.edu/~seander/bithacks.html#SwappingValuesXOR

2. https://bits.stephan-brumme.com/

3. http://aggregate.org/MAGIC/

—

Thanks, welcome any/all discussion in the comments. Happy to do a follow-up post if any interest…

JavaScript Bitwise Numbers