8-bitwise-operations

Contents

  - Get bit
  - Set bit
  - Clear bit
  - Toggle bit
- Tricks
  - Power of two
  - Max number
  - Min number
  - Multiply by 2^n
  - Divide by 2^n
  - Check equals
  - Check odd
  - Swap
  - Check sign
  - Flip sign

Introduction

Bit-wise operations are usually used to optimize some solutions or problems, to tightly pack data or quickly do some computations. There are various tricks achieved with bit wise operations, below are the most common ones, and the most important ones.

Representation

This is one of the most important crucial details to understand, how binary numbers are represented and how the same representation is extended, to also allow for representing negative numbers or even decimal / real ones as well

Basics

Binary number are represented as a chain or a string of of 1s and 0s. This string is enumerated (usually) from right to left, the right most bit being the least significant and the leftmost being the most significant. The numbering starts from 0 up to number of bits - 1. Some common types and their sizes, note that the size is platform dependent but the types and the sizes listed below are the ones most commonly seen

- byte = 8 bits = 1 byte
- short = 16 bits = 2 bytes
- integer = 32 bits = 4 bytes
- long = 64 bits = 8 bytes
- double = 64 = 8 bytes
- long double 80 bits = 10 bytes

As some of the sizes defined above vary from platform to platform, and they are implementation dependent, for example on some platform long and int might have the same size (4 bytes - x86), on other platforms integer could be defined as 16 bits. Older x86 systems usually define long as 32 bits, while long long is 64, on newer x64-86 systems these are usually both 64 bits wide. There are many variations but what is important, to remember is the general stride / size of each of those primitives.

```
// unsigned representation
0b1011010111101011
```

```
    // signed representation
    1b1011010111101011
```

Negative

To represent a negative number we use what is called 2s complement, negative numbers reserve the top most significant bit to represent the sign, which means that the total range that can be represented is halved, for example an 8 bit positive unsigned number can store 256 values, from 0 to 255, however 8 bit signed number can store values from -128, 0 & up to 127. The total numbers that are represented are still 256, but the range is halved in two effectively

The trick with 2s complement is that negative numbers are represented as the complement of their positive representation. The table below gives us an example of a 4bit wide number, which can represent the numbers from the range `[-8, 0, 7]`

Number	Positive	Negative
0	0000	—-
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	—-	1000

Take a good look at how the negative numbers mirror the positive ones, for example 1 is `0b001` while -1 is `1b111`. Two's complement, is calculated by taking the number and subtracting it from the max stride. For 4 bit width, we can represent a total of 16 numbers in total, that is `2^4`, therefore to represent negative 1, we have `-1 = 2^(4-1)- 1 = 7`, meaning the binary representation of `-1` is the same as the normal positive representation of the number `7` which is `0b111`, with the difference being only the sign bit therefore `-1 = 1 b111`. Note that above we take `2^(4-1)== 2^3`, since the top most bit is reserved for sign, effectively halving the range we can represent.

Note how 0 and -8 are actually also mirrored, just as every other number in the range form `[7 to -7]` is, the only difference being the sign bit, and in total we still have 16 numbers we can represent. (from `-1 to -7, plus 1 to 7 = 14`, then `0 and 8 = 2`, totaling 16 unique numbers)

This allows us to exploit negative numbers, which are very often used in bit wise operations to create masks such as:

- mask with none bits set -> 0 = 0b0000
- mask with only one bit set -> 1 = 0b0001
- mask with with all bits set -> -1 = 0b1111

Real

Also called floating point representation, the floating in the name suggests that the integer part of the real number can have a variable width of bits it can occupy. This allows us to represent fractions with more precision after the floating point number, or less precision but have more bits for the integer part of the floating number, e.g.

- more precision `1.23946792387456923`
- less precision `23984.2874`

Endianess

The endianess of a memory defines how at the machine level the memory is represented

- little endian - stores the least significant byte of information at the lowest smallest / address - `0x00FF = 255`
- big endian - stores the most significant byte of the information at the lowest / smallest address - `0xFF00 = 255`

Historically, Intel processors have always been little endian, while processors like the 6502, Motorolla, have always been employing big endian byte order. This is mostly relevant information when the same software has to be deployed on platforms with differing endianess. For example something developed on an Intel platform must be carefully ported over to another platform if it is using big endian byte order

Operations

There are just a few generally applicable operations on per bits basis. They are often used in conjunction to build composite operations to mutate and transform a bit set or a number of bits into some other state.

General Operations

Operation	Description	Example		
AND	1 when both bits are 1	1 & 1 = 1	1 & 0 = 0	0 & 0 = 0
XOR	1 when both bits are different	1 ^ 1 = 0	1 ^ 0 = 1	0 ^ 0 = 0
OR	1 when one or both bits are 1	1 | 1 = 1	1 | 0 = 1	0 | 0 = 0
NOT	1 when bit is 0, 0 when bit is 1	~1 = 0	~0 = 1	———-
LEFT SHIFT	shifts a bit into a position to the left	1 « 5 = 32	———-	———-
RIGHT SHIFT	shifts a bit into a position to the right	32 » 5 = 1	———-	———-

Logical Shifts

During the left and right bit shifts, `essentially move the bits that many positions in each direction`, taking the example from above, shifting left the number `1` means the bit which sits at position 0, will be moved 5 positions to the left, going into position 5 (where position of the bit is actually an index, starting from 0). Therefore to shift by 5 positions, we can start `counting from 0, 5 positions forward`, start counting `from current position of the 1st set bit` in the number (in our example the number is 1, the 1st set bit is at index 0)

To summarize this in a formula, `take the index / position of the first set bit, and simply add the number by which the target is being shifted`. The inverse is applicable for shifting in the other direction, but instead of addition we `subtract from the bit's index / position`

```
    0b0000001 - 1st set bit at index 0, therefore 0 + 5 = 5
    0b0100000 - the result is the bit is now moved at index 5
```

Arithmetic Shifts

There are the arithmetic left and right shifts, which work a bit differently than the logical ones, while in logical right and left shifts, we simply move all bits to the left or right by a specified offset, in arithmetic shifts in addition to what happens during logical shift, the most significant bits that are freed up are filled with the sign bit of. This has the effect of `dividing` a number by two, rounding down, for odd numbers

```
--------------
10110101 = -75
--------------
11011010 = -38
```

Note in the example above shifting to the right is accompanied by filling up the most significant bits which are freed up (and in a logical shift would otherwise be a 0), by the sign or to be more precise by value of the most significant bit of the number, in this case 1.

Both the logical shift operation and arithmetic ones are used to divide or multiply a number by the `number shifted bits * 2`. Shifts can be useful as efficient ways to perform multiplication or division of signed or unsigned integers by powers of two.

Arithmetic shifting left by n bits on a signed or unsigned binary number has the effect of multiplying it by 2n. While logical shifting has the same effect as arithmetic, however only valid when applied on unsigned integers

Get bit

The approach here, knowing the index of the target bit, is to move that specific bit at the least position, after that simply check if that bit is a 0 or a 1 by bit wise AND against 0b01. Note that the bit argument is and index, meaning it ranges for normal 4 byte integers, from 0 to 31 inclusive, for a total of 32 bits

```
    int getBit(int value, int bit) {
        // to get the value of a bit, we need to consider what that entails,
           to first shift the bit we want to the beginning of the
        // bit wise order, meaning we shift to the right the original value
           exactly 'bits' number of bits, then we can bit wise and this
        // value with the value of 1, the value of 1 has just one bit set,
           the 0th bit, the rest are all 0s, AND it with the shifted
        // number would leave only that 0th bit - bit(3) -> value(10110) >> 3
            -> shifted(00010) & mask(00001) -> result(00000). Note
        // that the value of the bit is an index, bit(3), means the 4th bit
           in the number
        int shifted = value >> bit;
        return shifted & 0x01;
    }
```

Set bit

To set a bit at a specific position we have to build a mask where only that specific bit at that position is set to 1, then we simply bit wise OR against the original number. If the bit at that position in the number was originally 0, it would become a 1, if it was 1, it is already set.

```
    int setBit(int value, int bit) {
        // to set a bit, what we need to consider is that the specific bit
           should become 1, while the rest must remain as they are, to
        // do this at the bit position we have to have a 1, and the rest of
           the bits can be 0, but instead of AND, we do a bit wise OR
        // between the mask and the value - bit(3) -> value (10110) | mask
           (01000) = result(11110). Note that the value of the bit is an
        // index, bit(3) means the 4th bit in the number
        int mask = (1 << bit);
        return value | mask;
    }
```

Clear bit

To clear a bit, means to set it to 0, this is the inverse operation of setting a bit. What that means is that we can simply extend the set bit logic, by simply applying a bit wise not on the mask and then apply AND bit wise against the original number, the mask would be only 1s, and a single 0, at the bit position we want to clear.

```
    int clearBit(int value, int bit) {
        // to clear a bit, what we need to consider is that the specific bit
           should become 0, while the reset must remain as they are,
        // to do this at the bit position we have to have a 0 value, and the
           rest of the mask should be all 1s, then we and the value
        // and the mask - bit(2) -> value (10110) & mask(11011) = result
           (10010). Note that the value of the bit is an index, bit(2)
        // means the 3rd bit in the number
        int mask = ~(1 << bit);
        return value & mask;
    }
```

Toggle bit

To toggle the bit means to flip it, if it was 1 it would become a 0, and vice versa. The approach here is to make a mask that is all 0s, and have a single bit set to 1, at the position we want to flip the bit. Then XOR the mask and the number. That xor operation would make sure to keep the 1s and 0s from the original number in place, and only ‘flip’ the set bit at the position, from the mask

```
    int toggleBit(int value, int bit) {
        // to toggle a bit we first create a mask with the specific bit set
           to 1, the rest are going to be 0s. To toggle the bit we can
        // simply xor the mask and the original value. What happens, is that
           the 0s in the mask would keep the 1s from the value, since
        // and the 0s from the mask would keep the 0s from the original value
            too. Since XOR (1 ^ 0 = 1) and XOR (0 ^ 0 = 0), therefore
        // - bit(2) -> value (10110) ^ mask(00100) = result(10010). Note that
            the value of the bit is an index, bit(2)
        int mask = (1 << bit);
        return value ^ mask;
    }
```

Tricks

Power of two

This is a pretty commonly used approach to check if a number if a power of 2, how does it work, think about how a power of two number is represented, it has only one 1 bit set in its representation,

For example the number 32, which is 2^5, which looks like: 0b100000 == 1 « 5. If we subtract 1 from that number we would get 0b011111 == 31. If we bit wise AND those two numbers, we will get exactly and always 0. Any other number that is not a power of two would not produce a 0 when we bit wise AND them with the same (number - 1)

```
    boolean pow2(int n) {
        return (n & (n - 1)) == 0;
    }
```

Max number

```
    int max(int a, int b) {
    }
```

Min number

```
    int min(int a, int b) {
    }
```

Multiply by 2^n

Logical and arithmetic shifts have the property of multiplying the number by `2^n` when shifting to the left

```
    int mult2(int x, int n) {
        return x << n;
    }
```

Divide by 2^n

Logical and arithmetic shifts have the property of dividing the number by `2^n` when shifting to the right

```
    int div2(int x, int n) {
        return x >> n;
    }
```

Check equals

To verify two numbers are equal, apply the XOR operations between them, if that produces 0, then all bits were exactly the same in all positions between the two input numbers, if there was one or more bit position which was different XOR would have produced a 1 in that position, and the result would not equal exactly 0.

```
    int equals(int a, int b) {
        return (a ^ b) == 0;
    }
```

Check odd

Remember that there only two types of numbers, it can either be odd one, or even.

To check if a number is odd, we need to realize that odd numbers are basically even numbers + 1. That is each even number added with 1 produces an odd number. Each odd number + 1 produces an even number.

Now going back to the binary representation, if the least significant bit is 1, that is the 0th bit, we can guarantee that whatever number it is, it is always going to be odd, we AND against a mask of 1, that will extract the 0th bit, and if it is 1, we know the number is odd

A simple example below demonstrates, how odd numbers all have 1 in their least significant bit position (the 0th bit index), conversely, the even numbers all have 0 in their least significant bit position.

Number	Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

This assumption & logical deduction, comes from the fact that if we try to convert a number from binary, where the 0th position is one, that would produce the following result `..... + 1 * 2^0`.

```
boolean odd(int a) {
    return (a & 1) == 1;
}
```

Swap

To swap two numbers without using a temporary variable. If we take the example of the numbers a `0b101` and b `0b010`.

```
a = (a) 0b101 ^ (b) 0b010 = 0b111
b = (a) 0b111 ^ (b) 0b010 = 0b101
a = (a) 0b111 ^ (b) 0b101 = 0b010
```

What we can take from the example above, is that the first XOR operation creates a special `mask`. That mask is used in two more xor operations against the original numbers to produce the other one - `b = b ^ mask` then `a = b ^ mask`

When applying XOR between number `b` and the `mask`, we would zero out all bits that are the same between the `mask` and the number `b`, the ones that remain 1s, essentially are leaving / producing the original number `a` as a result.

When applying the XOR between the same `mask` and the number `a` (which is now assigned to the variable `b`, from the previous step), we do the exact same operation, zero out all bits that are the same between `mask` and the number `a`, the ones that remain 1s, produce the original number `b` as a result

```
a ^= b;
b ^= a;
a ^= b;
```

Check sign

Checking the sign of two numbers by XOR operation, since we only care about the most significant bit, we know that if the bits are the same, XOR would produce a 0, in the most significant bit position, therefore the number would be 0 or positive, if the numbers are different we would get a negative mask by XOR between a and b

```
    boolean sign(int a, int b) {
        return (a ^ b) >= 0;
    }
```

Flip sign

To flip the sign of a number we have to consider what that means, to keep the original value of the number while getting it's negative representation, think about the number `6 = 0b110`, the negative representation of `6 = 1b010`, or in other words `2^3 - 6 = 2` and indeed that is the number `2 = 0b010` with the most significant bit set to 1, that is why we get `1b010`.

However, when we bit wise not a number, and we think about it in the context of the 2's complement representation, we actually get the opposite number but with one less (for positive) and one more (for negative). Going back to the example `6`, bit wise not 6 `0b110` is `1b001`, which is actually the number `-7`, if we add 1 to it, we will get exactly `-6`

The same logic would actually apply too if the number was originally negative, if we have `-6 = 1b010`, we do a flip we get `0b101`, which is 5, we add one and we get `6 = 0b110`

Why is the number off by 1, when we do a flip? That is due to the fact we can represent one more special type of number that is neither positive nor negative, and that is the number zero `0`. This is why in two's complement the numbers we can represent (for 4 bit numbers) are from `-(2^3)through 0 to +(2^3)-1`, that is 8 negative numbers, 7 positive and 0, in other words the range `[-8, 0, 7]`, that is total of 16 numbers that is exactly how many numbers 4 bits can hold `16 = (2^4)`

```
  int flip(int a) {
      return ~a + 1;
  }
```