Joe Birch Follow
Android Engineer at Buffer , Google Developer Expert for Android—Passionate about mobile development and learning. www.joebirch.co
Mar 5, 2016 · 11 min read

Bit Manipulation

Welcome to the latest post on Software Engineering 101, the publication where I aim to post weekly(ish) deep-dive articles on algorithms, design patterns, data structures and more!

Software Engineering 101

Deep-dives into Algorithms, Design patterns and more...

medium.com

This week we're going to look at the topic of bit manipulation. This is the process of performing logical operations on bit sequences in order to reach a desired result. Bit manipulation allows us to use these operators to reach certain sequences in an a clean and efficient manner, which is just one reason why it's essential to understand what operators are available and how we can make use of them.

Operators

We can perform operations by using some basic logical operators on the bits, let's take a look at what are available for use.

XOR (^)

The **XOR** operator can be used to copy the bit value if it is set only in one operand but not both.

$$x \wedge 0 = x$$

$$x \wedge 1 = x$$

$$x \wedge x = 0$$

As you can see above, the first two lines calculate to the value of **x** because only one of the operands is set to **x**. However, both operands in the third line have the value of **x**, meaning the line equates to **0**.

AND (&)

The **AND** operator can be used to copy the bit value if it exists in both of the operands.

$$x \,\&\, 0 = 0$$

$$x \,\&\, 1 = x$$

$$x \,\&\, x = x$$

As you can see above, the first statement equates to **0** as both operands are of a different value. The second statement equates to **x** because… and finally, the third statement also equates to **x** because both operands are equal to **x**.

OR (|)

The **OR** operator can be used top copy the bit value if it exists in either

of the provided operands.

$$x \mid 0 = x$$
$$x \mid 1 = 1$$
$$x \mid x = x$$

As you can see above, the first statement equates to x because x exists in one of the two operands.

Ones Complement (~)

Ones complement is a unary operation (also known as negation) and has the effect of simply 'flipping' a bit to it's opposite value, as shown below:

$$\sim 1 = 0$$
$$\sim 0 = 1$$

As you can see above, the ones complement operator simply flips the value of the given bit to its opposite value.

Manipulation Examples

To understand bit manipulation a bit better, let's take a look at some real examples! To begin with we'll look at some addition examples:

$$0100 + 0001 = 0101$$

Here we're simply adding the two values together, **0100 = 4** and **0001 = 1**. Adding these together gives us **5**, which has a binary representation of **0101**.

$$0101 + 0001 = 0110$$

Again we're simply adding the two values together, **0101 = 5**and **0001 = 1**. Adding these together gives us **6**, which has a binary representation of **0110**.

$$0010 - 0001 = 0001$$

Here we're simply subtracting the two values, **0010 = 2** and **0001 = 1**. Subtracting these gives us **1**, which has a binary representation of **0001**.

$$1100 - 0010 = 1010$$

Again we're simply subtracting the two values, **1100 = 12** and **0010 = 2**. Subtracting these gives us **1**, which has a binary representation of **1010**.

$$0010 * 0110 = 1100$$

Here we're simply multiplying the two values, **0010 = 2** and **0110 = 6**. Multiplying these gives us **12**, which has a binary representation of

1100.

$$0011 * 0011 = 1001$$

Again we're simply multiplying the two values, **0011 = 3** and **0011 = 3**. Multiplying these gives us 9, which has a binary representation of **1001**.

$$0110 \,\&\, 0100 = 0100$$

Here we're performing the **AND** operation on the bit sequences. As we previously saw, only the bits that are both set (value of 1) remain. So here we clear all bits except the third one - as this is the only bit that has been set in both operands.

$$0100 \,|\, 0101 = 0101$$

Here we're performing the **OR** operation on the bit sequences. As we previously saw, if the left operand is not equal to the right operand then we set the value to the value of the right operand. If the operand values are equal, then the bit remains unchanged. The only change in this operation is the first bit - the left operand first bit has the value 0 whilst the right operand has the first bit value of 1, so the first bit of the result is equal to 1.

$$0100 \wedge 0101 = 0001$$

Here we're performing the **XOR** operation on the bit sequences. As we previously saw, the XOR operation means that the bit will only be set to 1 if only one of the right / left operand bits at the current position holds that value. In the example above, we clear the third bit as both the left

and right operand have the value of 1. However, the first bit gets set to 1 as only the right operand sequence holds that value at this position.

Two's Complement

Two's complement allows us to represent negative values in the form of binary sequences, this is done by using either a **1** (for negative) or **0** (for positive) as the sign bit value to indicate whether the value is positive or not. An **N-bit** number (excluding the sign bit) is represented as it's two's complement value with respect to **2^N**.

For example, the 4-bit integer value of -6 would be represented as two parts using the first bit for the sign bit and the remaining 3 for the value itself. With respect to 2^N (where N = 3) which equals 8, the complement to 6 (the absolute value of -6) is 2 - which in binary is **010**. As a binary representation, using the first bit as the sign bit **-6** is equal to **1010**.

Another way of doing this is by simply flipping the current representation and adding the value of 1 to the result, which would look like so:

$$0110 = 6$$

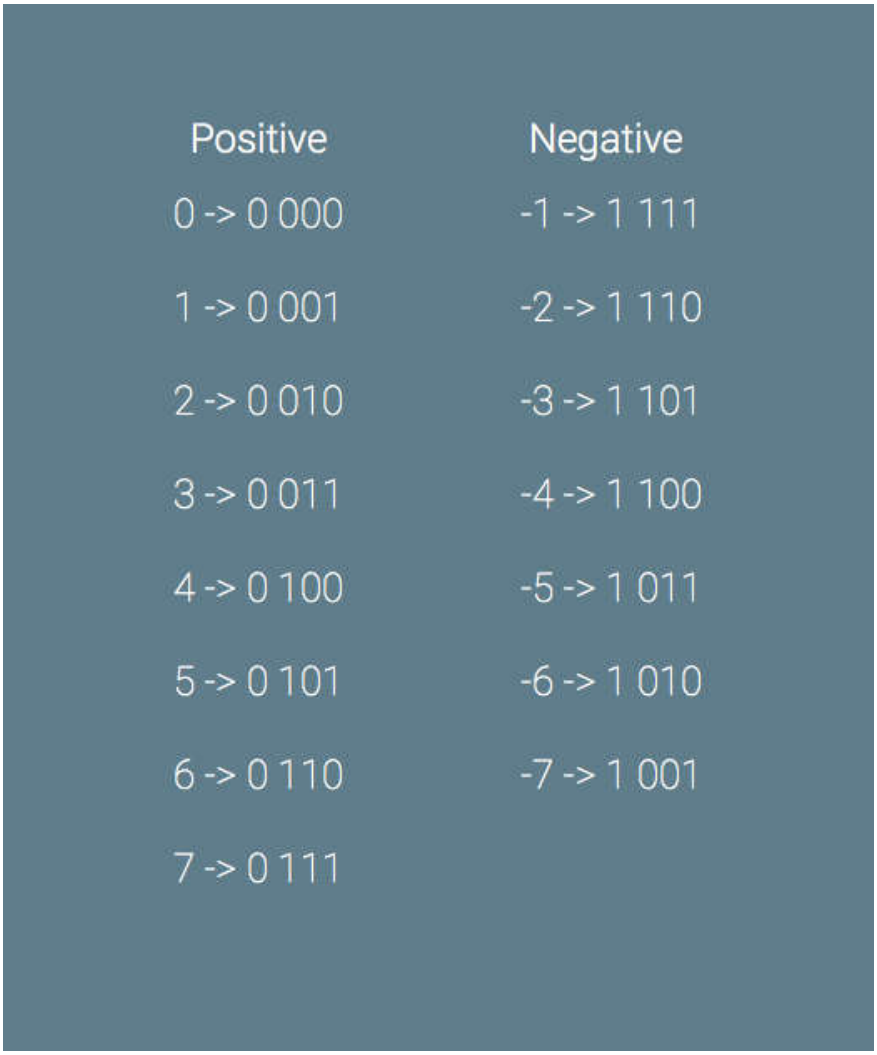
We begin with our initial binary representation of 6, we wish to turn this into the representation for -6. So next, we flip the bits to their opposite representation:

$$1001$$

This result gives our representation a sign bit, making it a negative value. However, we still need to carry out the last step of adding the value 1 to our binary sum:

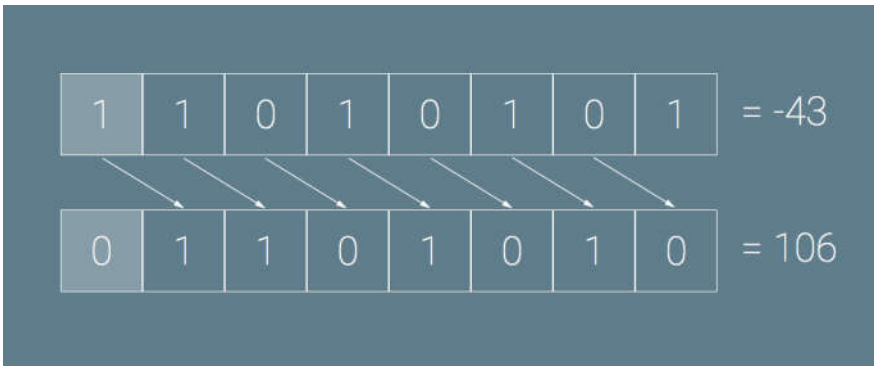
$$1010 = -6$$

Some further examples of positive and negative representations are as so:



Logical Shift

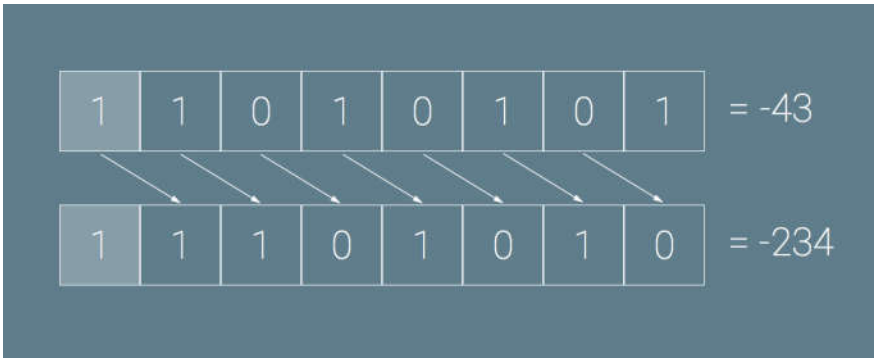
A logical shift is the process of shifting **all** bits in the desired direction (left **or** right) and place a **0** in the most significant bit - this process is represented by the >>> operator. For example, a logical right shift on the binary sequence below would look like so:



You can see here we've physically shifted all of the bits to the right, simply filling in the dominant position with a 0. This has transformed our binary representation from negative to positive and increased it's value dramatically.

Arithmetic Shift

The arithmetic shift is the process of again shifting **all** bits in the desired direction (left **or** right) but this time we place a the **value of the sign bit** in the most significant bit—this process is represented by the >> operator. For example, an arithmetic logical right shift on the same binary sequence would look like so:



Because our sign bit is equal to 1, after performing the shift we fill in the position of the dominant bit with a 1 value. In this case, the binary sequence remains a negative representation but our number has increased in value greatly.

Retrieving a Bit

If we want to retrieve the bit from a specific position in our binary sequence, then we can do with the use of a simple method:

```
public boolean getBit(int n, int i) {
    return ((n & (1 << i)) != 0);
}
```

To begin with, we take the value 1 and perform a left-shift operation using the **i** parameter as the right operand. We AND this result with our **n** parameter to remove all other bits other than the one at position i. This remaining bit is then compared to 0, returning true if the bit is equal to **1**.

Example

Let's start with a binary sequence:

$$n = 0110$$

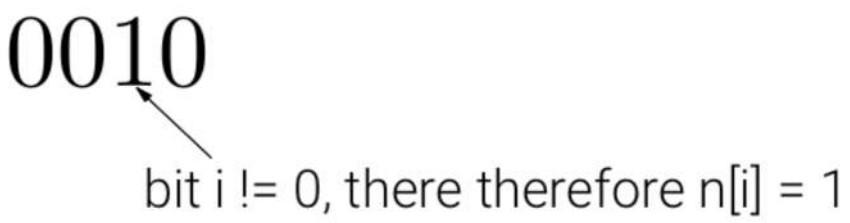
Let's say we want to get the second bit in the sequence, making **i = 2**. We begin by shifting the value one by i, giving us:

$$1 << i = 0010$$

We then **AND** this result with our sequence, **n**:

$$0010 \,\&\, 0010 = 0010$$

The final stage is simply comparing bit **i** of this result to **0**, if it does not equal **0** then bit **i** must equal **1** (and vice versa).



Setting a Bit

If we want to set a bit at a specific position in our binary sequence, then we can do with the use of a simple method where n is our binary sequence and i is the bit we wish to set:

```
public int setBit(int n, int i) {
    return n | (1 << i);
}
```

To begin with, we again perform a left-shift operation on the value 1 using the i parameter as the right operand. Next, we perform an OR operation using this value and our n parameter - this will cause the bit at i (and only i) to change. The other values of our mask are all 0 so they will not have any effect on our sequence n.

Example

Let's start again with the same binary sequence from the last example:

$$n = 0110$$

Let's say we want to set the fourth bit in the sequence, making **i = 4**. We begin by shifting the value one by i, giving us:

$$1 << i = 1000$$

We then use this result to perform an **OR** operation with our **n** sequence, giving:

$$0110 \mid 1000 = 1110$$

We can see from this that the 4th bit from our n sequence has been set.

Clearing a Bit

This function behaves in the opposite way to setting a bit, we can clear a bit by using the following function:

```
public int clearBit(int n, int i) {
    int mask = ~(1 << i);
    return num & mask;
}
```

To begin with, we again perform a left-shift operation on the value 1 using the i parameter as the right operand. However, this time we reverse the value representation by flipping all of the bits using the ones complement operator. Next, we perform an **AND** operation using this value and our n parameter—this will cause the bit at i (and only i) to be cleared. The other values of our mask are all 0 so they will not have any effect on our sequence n.

Example

Let's start again with the same binary sequence from the last example:

$$n = 0110$$

Let's say we want to clear the third bit in the sequence, making **i = 3**.

We begin by shifting the value one by i, giving us:

$$1 << i = 0100$$

Next, using the one's complement operator transforms this result to:

$$\sim 0100 = 1011$$

We next **AND** this result with our **n** sequence like so:

$$0110 \,\&\, 1011 = 0010$$

We can see from this that the 3rd bit from our n sequence has been cleared.

Updating a Bit

We can update a bit to a desired value by using the following function:

```
public int updateBit(int n, int i, boolean shouldSetBit) {
    int value = shouldSetBit ? 1 : 0;
    int mask = ~(1 << i);
    return (num & mask) | (value << i);
}
```

This function adds the ability to declare whether a bit should have the set value or not, using the flag **shouldSetBit**. We begin by clearing the value at position i with the use of a mask sequence. Once that's done, we perform an AND operation on our n and mask sequence. We then

perform a shift operation on **value** using **i** as the right operand, which is then used for an OR operation with the previously created value. This then updates the bit at the given index.

Example

Let's start again with the same binary sequence from the last example:

$$n = 0110$$

Let's say we want to update the first bit in the sequence, making **i = 1**. We begin by shifting the value one by i, giving us:

$$1 << i = 0001$$

Next, using the one's complement operator transforms this result to:

$$\sim 0001 = 1110$$

Now with a **mask** value of **1110** we can **AND** this result with our **n** sequence:

$$0110 \,\&\, 1110 = 0110$$

We next need to perform the shift operation on our **value** variable:

$$1 << i = 0001$$

Finally, we can perform the OR operation on these two calculated values:

$$0110 \mid 0001 = 0111$$

As you can see from this result, the **first** bit in our **n** sequence has been updated with our desired value of **1**.

And that's it!

We've learnt what bit manipulation is and how to use the operators involved with bitwise operations. Whether you're learning from scratch or refreshing your knowledge on data structures, I hope this deep-dive has been a good companion in understanding how bit manipulation works.

If you have any questions, feel free to to leave a response or drop me a tweet!

Joe Birch (@hitherejoe) | Twitter

The latest Tweets from Joe Birch (@hitherejoe). Android, photography, bass and a lot of running. Android Developer...

twitter.com

> *Check out more of my projects at hitherejoe.com*