Bitwise Operators and Bitboards

Maxim Rebguns

November 2023

This handout is for my LASA Computer Science Club captain presentation. It provides an overview of the importance of bitwise operators, and uses of the bitboard data structure. It serves as a reference and high-resolution accompaniment to my presentation.

Binary data

When representing data on computers, a computer architect has to work with restrictions imposed by electricity. These include current, voltage, speed, size, and other factors that could influence the capabilities of the device being made. In terms of representing data, electronics are generally classified into two types: analog and digital.

Figure 1: In analog electronics, we represent data continuously.

In *analog electronics*, data is represented within a continuous range of values. We might use current, voltage, frequency, or another property of electricity to represent a signal. For example, if we have a signal with a maximum voltage of 5 volts, we might represent any number from 1 to 10 as a fraction of those 5 volts. Thus, we could represent 0 with 0 volts, 5 with 2.5 volts, and 10 with 5 volts. The main drawback of analog electronics is that the signal is prone to fluctuations in real life, and slight changes in current or voltage would cause errors in the represented data.

Figure 2: In digital electronics, we represent data discretely.

In *digital electronics*, data is represented with two values: on or off. The maximum voltage[1] represents the "on" state, while the minimum voltage represents the "off" state. We can also denote these states also as 0/1 and true/false. This type of data is known as *binary data*, since we are using only two values. The benefit of this form of data is that slight fluctuations in voltage will be a lot less likely to result in errors in the data, since there are only two states.

[1] Voltage is a common property of electricity to use for representing data. However, we could also use others like current and frequency. We don't even need to use electricity, although we're focused on *elec*tronics.

Binary numbers

On	Off
True	False
1	0

Table 1: Binary states

Representing data in binary requires us to devise a system that only uses two states: on or off. A common way to represent integers in binary is what's known as *positional* or *place-value notation*. This notation parallels the notation we use every day—base-10 place-value notation.

To represent base-10 numbers, we use ten different symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. To represent numbers greater than 9, we now need a tens place, where the digit represents the number of tens in the number. However, we can only count up to 99. From there, we

add another place—the hundreds place, and so on. Notice how each place represents a magnitude of 10.

The same can be done with only two digits. We start out, as before, with the ones place. We can count 0, 1, but we've already run out of digits. Thus, to represent 2, we need a twos place. Then, we will need a fours place, an eights place, and so on. Notice how each place now represents a power of 2, rather than 10.

0	0	1	3
10^3	10^2	10^1	10^0
thousands	hundreds	tens	ones
1	1	0	1
2^3	2^2	2^1	2^0
eights	fours	twos	ones

Table 2: Place value representation of 13 in decimal (13) and binary (1101).

Other ways to represent data in binary

On a low level, computers don't really care what the zeroes and ones in memory represent. It is up to humans to devise encodings to represent data such as integers and strings using only those two values. We already saw one way to do this—using place-value representation. However, there are also other ways, suited to different types of data. Since each method has its own uses and implementations, I will list various ways to represent data on a computer. You may or may not have heard of one or more of these, but I will not delve into them here.

Wikipedia is an excellent source of background information on various data representation formats.

To represent integers, we might use place-value or two's complement [6, 4]. To store real numbers, computer scientists have devised floating-point, logarithmic, and fixed-point arithmetic. For characters, we could use something like ASCII or UTF-8. These are just some of the many types of data we need to represent on a computer. How would we represent classes, objects, pointers, arrays, or other data structures? Examining this question is essential when creating a new data format or programming language.

Bitwise operators

While we think of operations such as addition and subtraction as being fundamental in computer programming, there are more fundamental operators we use to manipulate bits on a lower level. A *bit*, or *b*inary dig*it*, is the fundamental unit of data on a computer, and has two states: 0 or 1. We use *bitwise operators* to manipulate individual bits. Bitwise operations are usually the fastest operations a computer can perform, and are sometimes even faster than addition, subtraction, multiplication, and division.

Manipulating bits is important in many different applications [1]. Since bits represent true or false data, we can use binary numbers to store multiple booleans. Computer programs often use a single integer to store multiple true/false values as *flags*.[2] In *embedded systems*, or small computers used everywhere from smart fridges to aircraft, we often have to manipulate individual bits to update certain regis-

[2] For example, the 8-bit integer `10110100` may represent 8 different true/false values. This is useful because we can pass a single integer into a function, while representing 8 boolean options at once.

ters and perform operations efficiently. On top of that, cryptography, compression, and computer graphics all rely on bitwise operations in their algorithms. Finally, bitwise operations are lightning fast and can be used in data structures such as *bitboards*, which we will explore next.

Bitwise NOT

The simplest operation we can do on a set of bits is to simply reverse all the bits (turn 0 into 1 and 1 into 0). This is known as *bitwise NOT* [2, 3]. For example, the bits `011010100` would become `100101011`. In most programming languages, bitwise not is denoted with a tilde (~).

We can use a *truth table* to show how input bits correspond to output bits after the NOT operation has been applied. In this case, the truth table is very simple: 0 becomes 1 and 1 becomes 0.

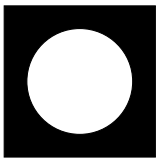

Figure 3: NOT operation (logical negation)

A	NOT A
`0`	`1`
`1`	`0`

Table 3: Truth table for NOT.

Bitwise AND, OR, and XOR

Next, we will explore bitwise operations that require two inputs. If you have done programming before, you will notice that bitwise operators do the same thing as logical operators (which you use in things like if-statements), except they work on individual bits.

When we apply *bitwise AND* on two series of bits, the resulting series of bits will only have 1 bits where both inputs have a 1 bit in that location [2, 3]. For example, `011010100 & 101001101 = 001000100`. The symbol for bitwise AND is typically an ampersand (&).

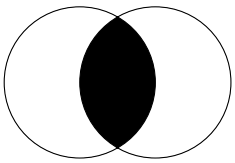

Figure 4: AND operation (logical intersection)

Bitwise OR is another operator that takes two inputs. The resulting series of bits will have 1 bits where either input or both have a 1 bit at that location [2, 3]. Thus, `011010100 | 101001101 = 111011101`. Bitwise OR is typically represented with a pipe (|).

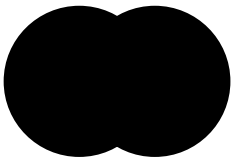

Figure 5: OR operation (logical union)

The final common two-input bitwise operator is *bitwise exclusive or (XOR)*. This operator is exactly like bitwise OR, except it requires only one input bit to be 1 for the output to be 1 [2, 3]. If both input bits at a certain position are 1, then the output is 0. For example, `011010100 ^ 101001101 = 110011001`. Bitwise XOR usually represented by a caret (^).

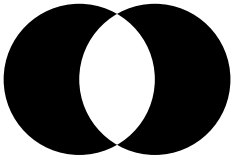

Figure 6: XOR operation (logical symmetric difference)

Bitwise left/right shift

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 4: Truth table for AND, OR, and XOR.

In many cases, we might want to "move" bits to the left or to the right. This is exactly what bit shift operators do. Both the left and right shift operators take two inputs: the value to be shifted, and a number of bits the value should be shifted by [2, 3].

The *bitwise left shift* moves all bits in a variable to the left, truncating the leftmost bits and padding the right side with zeroes. Thus, `0010110 >> 10 = 0000101`. The *bitwise right shift* does the same thing, but in the opposite direction. Rightmost bits are truncated, while the right side is padded with zeroes. Thus, `0010110 << 10 = 1011000`.

`0010110 >> 10 = 0000101`

Figure 7: Visual example of a right shift by 2. Bold shows padding and italic shows truncated digits.

Note that in the example, we are shifting to the right by 2, since 10 is 2 in binary. For the sake of consistency, I used binary for the shift value as well, but this isn't as common in programming, where we can explicitly denote the base of a number.

Bitboards

Now that you understand the utility of bitwise operators and how they work, let's look at an application of them in game development. When coding games such as chess and tic-tac-toe, it is necessary to represent the game board.

We might represent the board as an array. For the given board (Table 5), we can represent it with the following array:

o	x	x
	x	
o	o	

Table 5: The tic-tac-toe board.

For code examples, I am using C, since it is a language commonly used for low-level programming.

```
typedef enum { MOVE_EMPTY, MOVE_X, MOVE_O } move;
move board[][] = {
    {MOVE_O, MOVE_X, MOVE_X},
    {MOVE_EMPTY, MOVE_X, MOVE_EMPTY},
    {MOVE_O, MOVE_O, MOVE_EMPTY}
};
```

We define three possible states, and then fill them into a two-dimensional array. We could also do the same thing with a one-dimensional array. Either way, using an array to represent a game board like this is a bad idea.

First of all, for just three states (empty, x, and o), we are using an integer, which can take up over a million different values. This is a clear waste of memory. On top of that, we would have to navigate the board through iteration. Loops add time complexity and can become unwieldy, especially if we are looking for things like diagonal wins. Additionally, for more complex games like chess, performing move operations on a board array would require a lot of changing of array values, which is not the fastest operation on a computer.

We can do the same exact board using just two integers known as *bitboards*. A bitboard is an integer representation of a game board, where each bit represents the existence of a piece at a certain position [5]. Since bits can only be 1 or 0, we need two bitboards to represent x and o positions separately.

```
typedef uint16_t board;
//                   123456789
board x_positions = 0b011100000;
board o_positions = 0b100000110;
```

In this code, we define the type `board` to be a 16-bit integer. The board itself is only 9 bits, but that is just over the size of an 8-bit integer. Thus, 7 bits are wasted, but this is a lot less than before.[3] Instead of thinking the board as an integer, we think of it as a structure of bits that can be manipulated with bitwise operators. Note that what square we think of as 1 (the most significant bit) and what square we think of as 9 (the least significant bit) don't really matter. In this case, we made the top left square 1, and move in row major order down to the bottom right square at 9.

[3] Chess is convenient in that the board has 64 squares, which can be perfectly represented in a 64-bit integer.

1	2	3
4	5	6
7	8	9

Table 6: Our board numbering system.

Representing the board in this way unlocks the speed and power of bitwise operators. Becase a bitboard is represented as an integer, you can use bitwise operators to create new bitboards, pass them into functions, and make copies of them, worrying a lot less about memory usage and speed.

Bitboard operations

Let's say we now have two bitboards representing x and o positions in tic-tac-toe. If we need a bitboard representing all taken positions, all we have to do is apply a bitwise OR to them:

```
board taken_positions = x_positions | o_positions;
```

0	1	1		1	0	0		1	1	1
0	1	0	|	0	0	0	=	0	1	0
0	0	0		1	1	0		1	1	0

Table 7: ORing the x and o positions gives a bitboard of all taken positions.

We can now use this board of taken positions to determine, for example, if a player's move is valid. We can invert the taken positions to get a bitboard of all positions that haven't been taken (since every 1 becomes a 0, and 0 becomes a 1).

	1	1	1		0	0	0
~	0	1	0	=	1	0	1
	1	1	0		0	0	1

Table 8: Inverting the taken positions to get a bitboard of not taken (valid) positions.

We can then AND this bitboard with the bitboard representing the player's move (it is full of zeroes except for a single 1 at the position where the player wants to move). This resulting board will have a 1 somewhere if the player's move overlaps with a valid position.

However, if the player's move doesn't overlap with a valid position, then we get an empty bitboard. Since an empty bitboard represents the integer value 0, we know that the move is invalid only if the resulting bitboard is equal to 0, meaning that they player's move did not overlap any of the allowed positions.

0	0	0		0	0	0		0	0	0
0	0	0	&	1	0	1	=	0	0	0
0	0	1		0	0	1		0	0	1

Table 9: The player tries to move to the bottom-right corner, which is a valid move. Therefore, the resulting bitboard is non-zero.

1	0	0		0	0	0		0	0	0
0	0	0	&	1	0	1	=	0	0	0
0	0	0		0	0	1		0	0	0

Table 10: The player tries to move to the top-left corner, which is already taken. Therefore, the resulting bitboard is 0.

```
// Generating a bitboard of valid positions.
board valid_positions = ~taken_positions;
// Using that bitboard to validate the player's move.
bool is_valid = (move & valid_positions) != 0;
```

We could also do things like checking if somebody has won. Instead of using loops, we can simply encode the eight possible win conditions in tic-tac-toe as eight different bitboards, and then AND them with the player's board to see if the result matches the win condition:

```
bool has_won = ((board & l_vert_win) == l_vert_win)
               || (board & m_vert_win) == m_vert_win)
               || ...);
```

Of course, the possibilities are endless. For a complex game like chess, we may want to use clever bitwise tactics, such as shifts to apply different moves. It may seem counter-intuitive, but it is extremely speed-efficient. Speed is a necessary trait for things like chess AIs, which need to cycle through hundreds of possible states to choose the most optimal one.

See the Chess Programming Wiki (`https://www.chessprogramming.org`) for a whole bunch of bitboard tips and tricks, as well as other clever ways to represent game boards and implement a chess program.

Conclusion

Modern digital computing revolves around the notion of two states: 0 and 1, which can be encoded in electricity to represent data. There are many ways to represent different types of data in binary, including bitboards. The good thing about bitboards is that they're a great way to represent game boards in chess, checkers, and tic-tac-toe. We can use bitwise operators to manipulate individual bits, which is broadly applicable to many computer science fields. In the case of bitboards, bitwise operators allow us to manipulate bits quickly and efficiently.

References

[1] Real world use cases of bitwise operators.

[2] Wikipedia Contributors. Bitwise operation, November 2023.

[3] Wikipedia Contributors. Bitwise operations in C, November 2023.

[4] Wikipedia Contributors. Floating-point arithmetic, November 2023.

[5] Gerd Isenberg and Chess Programming Wiki Contributors. Bitboards, March 2022.

[6] John Zahorjan. Binary Representation, 2022.