

[PRACTICE](#)[COMPETE](#)[JOBS](#)[LEADERBOARD](#)[Search](#)[Dashboard](#) > [Tutorials](#) > [10 Days of Javascript](#) > [Day 6: Bitwise Operators](#)

Day 6: Bitwise Operators

by [AvmnuSng](#)[Problem](#)[Submissions](#)[Leaderboard](#)[Discussions](#)[Editorial](#) [Tutorial](#)

Bitwise Operators in JavaScript

Bitwise Operators

Before discussing bitwise operators, let's review the following:

We use cookies to ensure you have the best browsing experience on our website. Please read our [cookie policy](#) for more information about how we use cookies.

[OK](#)

- Base-**10** (decimal number) to base-**2** (binary number) conversions
- Base-**2** (binary number) to base-**10** (decimal number) conversions
- Representing negative base-**10** numbers in base-**2**

-

RECOMMENDED ARTICLE

This article provides some additional background on these topics.

Binary Number Basics

- The binary, or base-**2**, numeral system is a way for us to express numbers. It's called binary because it only uses *two* symbols, **0** and **1**, to express these numbers. Examples of binary numbers are **1011**, **100011**, and **111**.
- The number of symbols in a number system is called its *base* or *radix*. This is why we often see binary numbers referred to as base-**2** (because each digit is in $\{0, 1\}$), and decimal numbers referred to as base-**10** (because each digit is in $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$).
- We use the notation $(?)_b$ to discuss numbers with different radices, where $?$ is the number and b is the base. For example, $(1101)_2$ is the binary equivalent of the decimal number $(13)_{10}$.
- Each digit in a binary number is called a *bit*.

We use the following algorithm to convert a decimal integer to a binary number:

1. Take the decimal integer, divide it by **2**, and record the quotient (the number of times **2** divided the integer) and the remainder (the number of units left over from the division, which will always be **0** or **1**).
2. Repeat step **1** on the quotient until the quotient becomes **0**.
3. Look at the sequence of remainders. The remainder from the first division operation corresponds to the binary number's *least significant bit* (LSB) and the remainder from the last division operation corresponds to the number's *most significant bit* (MSB). To get our binary number, we simply need to concatenate these remainder bits from most to least significant.

Examples

The table below depicts the conversion from $(71)_{10}$ to $(?)_2$:

n	$\frac{n}{2}$	Remainder	Significance
71	35	1	Least
35	17	1	↑
17	8	1	↑
8	4	0	↑
4	2	0	↑

n	$\frac{n}{2}$	Remainder	Significance
1	0	1	Most

When we concatenate our remainder bits from the bottom up (i.e., *most* to *least* significant), we get $(71)_{10} \Rightarrow (1000111)_2$.

The table below depicts the conversion from $(129)_{10}$ to $(?)_2$:

n	$\frac{n}{2}$	Remainder	Significance
129	64	1	Least
64	32	0	↑
32	16	0	↑
16	8	0	↑
8	4	0	↑
4	2	0	↑
2	1	0	↑
1	0	1	Most

When we concatenate our remainder bits from the bottom up (i.e., *most* to *least* significant), we get $(129)_{10} \Rightarrow (10000001)_2$.

Page 9 (Binary) to Page 10 (Decimal) Conversions

Let's say we have a binary number with d bits we can express as $b_{d-1}b_{d-2}b_{d-3} \dots b_1b_0$.
 We use the following summation to calculate its base-10 integer value:

$$\sum_{i=0}^{d-1} (b_i \times 2^i)$$

Note that b_0 is the LSB and b_{d-1} is the MSB.

Example

The table below depicts the conversion from $(111000001)_2$ to $(?)_{10}$:

Significance	b_i	i	2^i	$b_i \cdot 2^i$
Least	1	0	1	1
↑	0	1	2	0
↑	0	2	4	0
↑	0	3	8	0
↑	0	4	16	0
↑	0	5	32	0
↑	1	6	64	64
↑	1	7	128	128
Most	1	8	256	256

Representing Negative Base-10 Numbers in Base-2

In this explanation, we're representing our integers as **32-bit** signed binary numbers. To represent an integer, $-n$, in binary, we perform the following steps:

1. Find the **32-bit** binary representation of n .
2. Take the **1's** complement. We do this by inverting all the binary number's bits (i.e., every **0** becomes a **1**, and every **1** becomes a **0**).
3. Take the **2's** complement by adding **1** to the **1's** complement.

The **2's** complement is the binary representation of $-n$.

Examples

Let's look at the binary representation of $(-12)_{10}$:

1. First, let's look at $n = (12)_{10}$. When we convert it to binary, we get $(00000000000000000000000000001100)_2$.
2. The **1's** complement of $(00000000000000000000000000001100)_2$ is $(111111111111111111111111110011)_2$.
3. Then we add **1** to $(111111111111111111111111110011)_2$ to get n 's **2's** complement, $(111111111111111111111111110100)_2$, which is the value of $-n = (-12)_{10}$.

1. $n = (314)_{10} \Rightarrow (000000000000000000000000100111010)_2$
2. $(\sim 314)_{10} = (11111111111111111111111011000101)_2$ is n 's 1's complement.
3. $(-314)_{10} = (\sim 314)_{10} + 1 = (11111111111111111111111011000110)_2$ is n 's 2's complement (i.e., $-n$).

Bitwise Operation Conventions

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to **32**-bit integers, meaning they're expressed as sequences of **32** zeroes and ones. Any number larger than **32** bits is reduced to **32** bits by cutting off and discarding its excess most significant bits. The example below shows a binary integer before and after it's converted to a **32**-bit integer:

```
Before: 111001101111101000000000000000110000000000001
After:   101000000000000000110000000000001
```

- Each bit in the first operand is paired with the corresponding bit in the second operand from least to most significant. In other words, the first LSB matches the first LSB, the second LSB matches the second LSB, and so on.
- The operator is applied to each pair of bits so that the resulting number is constructed bitwise (i.e., bit-by-bit).

This operator performs the *AND* operation on each pair of bits. Given two binary numbers, a and b , the result of an AND operation on the corresponding bits at each position i (i.e., $a_i \& b_i$) is **1** if and only if both a_i and b_i are **1**. The truth table for the bitwise AND operation is:

a_i	b_i	$a_i \& b_i$
0	0	0
0	1	0
1	0	0
1	1	1

For example, $101 \& 110 = 100$.

Bitwise OR (|)

This operator performs the *OR* operation on each pair of bits. Given two binary numbers, a and b , the result of an OR operation on the corresponding bits at each position i (i.e., $a_i | b_i$) is **1** if a_i and/or b_i are **1**. The truth table for the bitwise OR operation is:

a_i	b_i	$a_i b_i$
0	0	0
0	1	1
1	0	1
1	1	1

For example, $0101 \mid 0110 = 0111$.

Bitwise XOR (^)

This operator performs the *XOR* operation on each pair of bits. Given two binary numbers, a and b , the result of an XOR operation on the corresponding bits at each position i (i.e., $a_i \wedge b_i$) is **1** if *either* a_i *or* b_i is **1** (i.e., the values of the two operands are different). The truth table for the bitwise XOR operation is:

a_i	b_i	$a_i \wedge b_i$
0	0	0
0	1	1
1	0	1
1	1	0

For example, $0101 \wedge 0110 = 0011$.

Bitwise NOT (~)

This operator performs the *NOT* operation on each pair of bits in a number. Given a binary number, a , the NOT operation (i.e., $\sim a$) inverts each bit in the number. The truth table for the bitwise NOT operation is:



[Contest Calendar](#) | [Interview Prep](#) | [Blog](#) | [Scoring](#) | [Environment](#) | [FAQ](#) | [About Us](#) | [Support](#) | [Careers](#) | [Terms Of Service](#) | [Privacy Policy](#) | [Request a Feature](#)