Bitwise operation

In computer programming, a **bitwise operation** operates on a bit string, a bit array or a binary numeral (considered as a bit string) at the level of its individual bits. It is a fast and simple action, basic to the higher-level arithmetic operations and directly supported by the processor. Most bitwise operations are presented as two-operand instructions where the result replaces one of the input operands.

On simple low-cost processors, typically, bitwise operations are substantially faster than division, several times faster than multiplication, and sometimes significantly faster than addition. While modern processors usually perform addition and multiplication just as fast as bitwise operations due to their longer instruction pipelines and other architectural design choices, bitwise operations do commonly use less power because of the reduced use of resources.[1]

Bitwise operators

In the explanations below, any indication of a bit's position is counted from the right (least significant) side, advancing left. For example, the binary value 0001 (decimal 1) has zeroes at every position but the first (i.e., the rightmost) one.

NOT

The **bitwise NOT**, or **bitwise complement**, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Bits that are 0 become 1, and those that are 1 become 0. For example:

```
NOT 0111  (decimal 7)
  = 1000  (decimal 8)
```

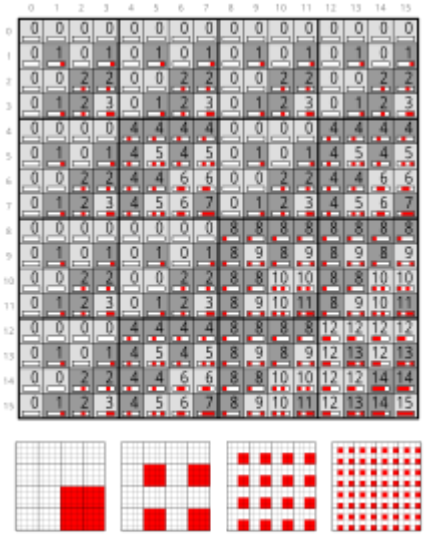
```
NOT 10101011  (decimal 171)
  = 01010100  (decimal 84)
```

The result is equal to the two's complement of the value minus one. If two's complement arithmetic is used, then `NOT x = -x – 1`.

For unsigned integers, the bitwise complement of a number is the "mirror reflection" of the number across the half-way point of the unsigned integer's range. For example, for 8-bit unsigned integers, `NOT x = 255 - x`, which can be visualized on a graph as a downward line that effectively "flips" an increasing range from 0 to 255, to a decreasing range from 255 to 0. A

simple but illustrative example use is to invert a grayscale image where each pixel is stored as an unsigned integer.

AND



Bitwise AND of 4-bit integers

A **bitwise AND** is a binary operation that takes two equal-length binary representations and performs the logical AND operation on each pair of the corresponding bits. Thus, if both bits in the compared position are 1, the bit in the resulting binary representation is 1 (1 × 1 = 1); otherwise, the result is 0 (1 × 0 = 0 and 0 × 0 = 0). For example:

```
    0101 (decimal 5)
AND 0011 (decimal 3)
  = 0001 (decimal 1)
```

The operation may be used to determine whether a particular bit is *set* (1) or *cleared* (0). For example, given a bit pattern 0011 (decimal 3), to determine whether the second bit is set we use a bitwise AND with a bit pattern containing 1 only in the second bit:

```
    0011 (decimal 3)
AND 0010 (decimal 2)
  = 0010 (decimal 2)
```

Because the result 0010 is non-zero, we know the second bit in the original pattern was set. This is often called *bit masking*. (By analogy, the use of masking tape covers, or *masks*, portions that should not be altered or portions that are not of interest. In this case, the 0 values mask the bits that are not of interest.)

The bitwise AND may be used to clear selected bits (or flags) of a register in which each bit represents an individual Boolean state. This technique is an efficient way to store a number of

Boolean values using as little memory as possible.

For example, 0110 (decimal 6) can be considered a set of four flags numbered from right to left, where the first and fourth flags are clear (0), and the second and third flags are set (1). The third flag may be cleared by using a bitwise AND with the pattern that has a zero only in the third bit:

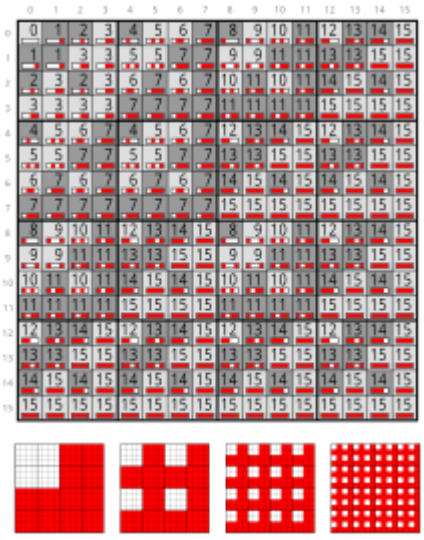
```
    0110 (decimal 6)
AND 1011 (decimal 11)
  = 0010 (decimal 2)
```

Because of this property, it becomes easy to check the parity of a binary number by checking the value of the lowest valued bit. Using the example above:

```
    0110 (decimal 6)
AND 0001 (decimal 1)
  = 0000 (decimal 0)
```

Because 6 AND 1 is zero, 6 is divisible by two and therefore even.

OR



Bitwise OR of 4-bit integers

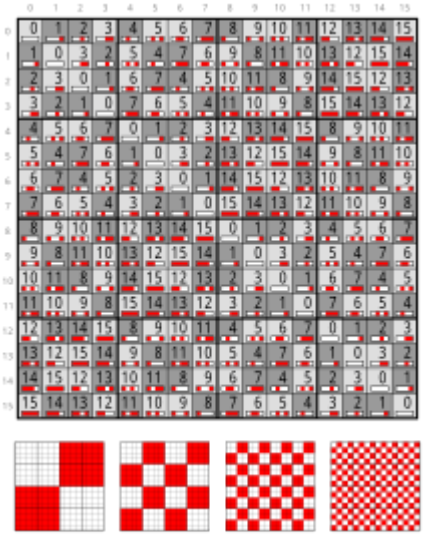
A **bitwise OR** is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits. The result in each position is 0 if both bits are 0, while otherwise the result is 1. For example:

```
   0101 (decimal 5)
OR 0011 (decimal 3)
 = 0111 (decimal 7)
```

The bitwise OR may be used to set to 1 the selected bits of the register described above. For example, the fourth bit of 0010 (decimal 2) may be set by performing a bitwise OR with the pattern with only the fourth bit set:

```
   0010 (decimal 2)
OR 1000 (decimal 8)
 = 1010 (decimal 10)
```

XOR



Bitwise XOR of 4-bit integers

A **bitwise XOR** is a binary operation that takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits. The result in each position is 1 if only one of the bits is 1, but will be 0 if both are 0 or both are 1. In this we perform the comparison of two bits, being 1 if the two bits are different, and 0 if they are the same. For example:

```
    0101 (decimal 5)
XOR 0011 (decimal 3)
  = 0110 (decimal 6)
```

The bitwise XOR may be used to invert selected bits in a register (also called toggle or flip). Any bit may be toggled by XORing it with 1. For example, given the bit pattern 0010 (decimal 2) the second and fourth bits may be toggled by a bitwise XOR with a bit pattern containing 1 in the second and fourth positions:

```
    0010 (decimal 2)
XOR 1010 (decimal 10)
  = 1000 (decimal 8)
```

This technique may be used to manipulate bit patterns representing sets of Boolean states.

Assembly language programmers and optimizing compilers sometimes use XOR as a short-cut to setting the value of a register to zero. Performing XOR on a value against itself always yields zero, and on many architectures this operation requires fewer clock cycles and less memory than loading a zero value and saving it to the register.

If the set of bit strings of fixed length n (i.e. machine words) is thought of as an n-dimensional vector space $\mathbf{F}_2^n$ over the field $\mathbf{F}_2$, then vector addition corresponds to the bitwise XOR.

Mathematical equivalents

Assuming $x \geq y$, for the non-negative integers, the bitwise operations can be written as follows:

$$\operatorname{NOT} x = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 + 1 \right) \bmod 2 \right] = 2^{\lfloor \log_2(x) \rfloor + 1} - 1 - x$$

$$x \operatorname{AND} y = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right)$$

$$x \operatorname{OR} y = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left(\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) - \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right)$$

$$x \operatorname{XOR} y = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left(\left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right] \bmod 2 \right) = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor + \left\lfloor \frac{y}{2^n} \right\rfloor \right) \bmod 2 \right]$$

Truth table for all binary logical operators

There are 16 possible truth functions of two binary variables; this defines a truth table.

Here is the bitwise equivalent operations of two bits P and Q:

p	*q*	F[0]	NOR[1]	Xq[2]	¬p[3]	↛[4]	¬q[5]	XOR[6]	NAND[7]	AND[8]	XNOR[9]	q[10]	If/then[11]	p[12]	Then/if[13]	OR[14]	T
1	**1**	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	
1	**0**	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	
0	**1**	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	
0	**0**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
Bitwise equivalents		**0**	**NOT (p OR q)**	**(NOT p) AND q**	**NOT p**	**p AND (NOT q)**	**NOT q**	**p XOR q**	**NOT (p AND q)**	**p AND q**	**NOT (p XOR q)**	**q**	**(NOT p) OR q**	**p**	**p OR (NOT q)**	**p OR q**	

Bit shifts

The **bit shifts** are sometimes considered bitwise operations, because they treat a value as a series of bits rather than as a numerical quantity. In these operations, the digits are moved, or *shifted*, to the left or right. Registers in a computer processor have a fixed width, so some bits will be "shifted out" of the register at one end, while the same number of bits are "shifted in" from the other end; the differences between bit shift operators lie in how they determine the values of the shifted-in bits.

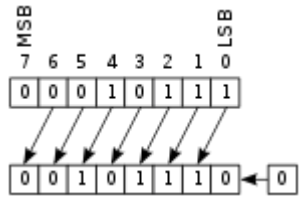
Bit addressing

If the width of the register (frequently 32 or even 64) is larger than the number of bits (usually 8) of the smallest addressable unit, frequently called byte, the shift operations induce an addressing scheme from the bytes to the bits. Thereby the orientations "left" and "right" are taken from the standard writing of numbers in a place-value notation, such that a left shift increases and a right shift decreases the value of the number — if the left digits are read first, this makes up a big-endian orientation. Disregarding the boundary effects at both ends of the register, arithmetic and logical shift operations behave the same, and a shift by 8 bit positions transports the bit pattern by 1 byte position in the following way:

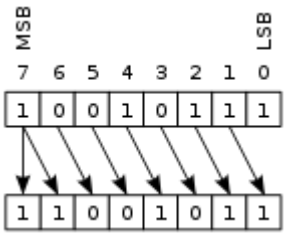
Little-endian ordering: a left shift by 8 positions increases the byte address by 1,
a right shift by 8 positions decreases the byte address by 1.

Big-endian ordering: a left shift by 8 positions decreases the byte address by 1,
a right shift by 8 positions increases the byte address by 1.

Arithmetic shift



Left arithmetic shift



Right arithmetic shift

In an *arithmetic shift*, the bits that are shifted out of either end are discarded. In a left arithmetic shift, zeros are shifted in on the right; in a right arithmetic shift, the sign bit (the MSB in two's complement) is shifted in on the left, thus preserving the sign of the operand.

This example uses an 8-bit register, interpreted as two's complement:

```
   00010111 (decimal +23) LEFT-SHIFT
=  00101110 (decimal +46)
```

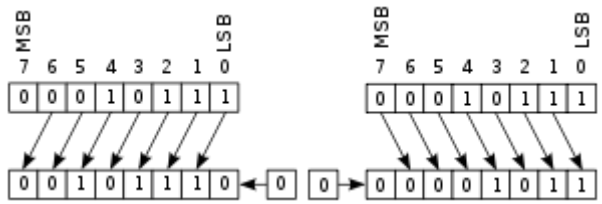
```
   10010111 (decimal −105) RIGHT-SHIFT
=  11001011 (decimal −53)
```

In the first case, the leftmost digit was shifted past the end of the register, and a new 0 was shifted into the rightmost position. In the second case, the rightmost 1 was shifted out (perhaps into the carry flag), and a new 1 was copied into the leftmost position, preserving the sign of the number. Multiple shifts are sometimes shortened to a single shift by some number of digits. For example:

```
   00010111 (decimal +23) LEFT-SHIFT-BY-TWO
=  01011100 (decimal +92)
```

A left arithmetic shift by n is equivalent to multiplying by 2^n (provided the value does not overflow), while a right arithmetic shift by n of a two's complement value is equivalent to taking the floor of division by 2^n. If the binary number is treated as ones' complement, then the same right-shift operation results in division by 2^n and rounding toward zero.

Logical shift



Left logical shift Right logical shift

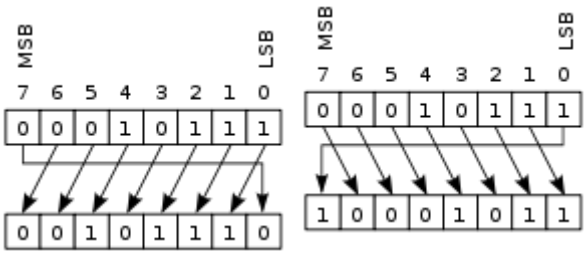
In a *logical shift*, zeros are shifted in to replace the discarded bits. Therefore, the logical and arithmetic left-shifts are exactly the same.

However, as the logical right-shift inserts value 0 bits into the most significant bit, instead of copying the sign bit, it is ideal for unsigned binary numbers, while the arithmetic right-shift is ideal for signed two's complement binary numbers.

Circular shift

Another form of shift is the *circular shift*, *bitwise rotation* or *bit rotation*.

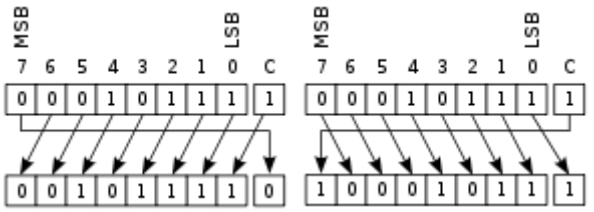
Rotate



Left circular shift or rotate Right circular shift or rotate

In this operation, sometimes called *rotate no carry*, the bits are "rotated" as if the left and right ends of the register were joined. The value that is shifted into the right during a left-shift is whatever value was shifted out on the left, and vice versa for a right-shift operation. This is useful if it is necessary to retain all the existing bits, and is frequently used in digital cryptography.

Rotate through carry



Left rotate through carry Right rotate through carry

Rotate through carry is a variant of the rotate operation, where the bit that is shifted in (on either end) is the old value of the carry flag, and the bit that is shifted out (on the other end) becomes the new value of the carry flag.

A single *rotate through carry* can simulate a logical or arithmetic shift of one position by setting up the carry flag beforehand. For example, if the carry flag contains 0, then `x RIGHT-ROTATE-THROUGH-CARRY-BY-ONE` is a logical right-shift, and if the carry flag contains a copy of the sign bit, then `x RIGHT-ROTATE-THROUGH-CARRY-BY-ONE` is an arithmetic right-shift. For this reason, some microcontrollers such as low end PICs just have *rotate* and *rotate through carry*, and don't bother with arithmetic or logical shift instructions.

Rotate through carry is especially useful when performing shifts on numbers larger than the processor's native word size, because if a large number is stored in two registers, the bit that is shifted off one end of the first register must come in at the other end of the second. With rotate-

through-carry, that bit is "saved" in the carry flag during the first shift, ready to shift in during the second shift without any extra preparation.

In high-level languages

In C family of languages

In C and C++ languages, the logical shift operators are " `<<` " for left shift and " `>>` " for right shift. The number of places to shift is given as the second argument to the operator. For example,

```
x = y << 2;
```

assigns `x` the result of shifting `y` to the left by two bits, which is equivalent to a multiplication by four.

Shifts can result in implementation-defined behavior or undefined behavior, so care must be taken when using them. The result of shifting by a bit count greater than or equal to the word's size is undefined behavior in C and C++.[2][3] Right-shifting a negative value is implementation-defined and not recommended by good coding practice;[4] the result of left-shifting a signed value is undefined if the result cannot be represented in the result type.[2]

In C#, the right-shift is an arithmetic shift when the first operand is an int or long. If the first operand is of type uint or ulong, the right-shift is a logical shift.[5]

Circular shifts

The C-family of languages lack a rotate operator (although C++20 provides `std::rotl` and `std::rotr`), but one can be synthesized from the shift operators. Care must be taken to ensure the statement is well formed to avoid undefined behavior and timing attacks in software with security requirements.[6] For example, a naive implementation that left-rotates a 32-bit unsigned value `x` by `n` positions is simply

```
uint32_t x = ..., n = ...;
uint32_t y = (x << n) | (x >> (32 - n));
```

However, a shift by `0` bits results in undefined behavior in the right-hand expression `(x >> (32 - n))` because `32 - 0` is `32`, and `32` is outside the range 0–31 inclusive. A second try might result in

```
uint32_t x = ..., n = ...;
uint32_t y = n ? (x << n) | (x >> (32 - n)) : x;
```

where the shift amount is tested to ensure that it does not introduce undefined behavior. However, the branch adds an additional code path and presents an opportunity for timing analysis and attack, which is often not acceptable in high-integrity software.[6] In addition, the code compiles to multiple machine instructions, which is often less efficient than the processor's native instruction.

To avoid the undefined behavior and branches under GCC and Clang, the following is recommended. The pattern is recognized by many compilers, and the compiler will emit a single rotate instruction:[7][8][9]

```
uint32_t x = ..., n = ...;
uint32_t y = (x << n) | (x >> (-n & 31));
```

There are also compiler-specific intrinsics implementing circular shifts, like _rotl8, _rotl16 (http://msdn.microsoft.com/en-us/library/t5e2f3sc(VS.80).aspx) , _rotr8, _rotr16 (http://msdn.microsoft.com/en-us/library/yy0728bz(VS.80).aspx) in Microsoft Visual C++. Clang provides some rotate intrinsics for Microsoft compatibility that suffers the problems above.[9] GCC does not offer rotate intrinsics. Intel also provides x86 intrinsics (https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=rot&techs=Other) .

Java

In Java, all integer types are signed, so the " `<<` " and " `>>` " operators perform arithmetic shifts. Java adds the operator " `>>>` " to perform logical right shifts, but since the logical and arithmetic left-shift operations are identical for signed integer, there is no " `<<<` " operator in Java.

More details of Java shift operators:[10]

- The operators `<<` (left shift), `>>` (signed right shift), and `>>>` (unsigned right shift) are called the *shift operators*.
- The type of the shift expression is the promoted type of the left-hand operand. For example, `aByte >>> 2` is equivalent to `((int) aByte) >>> 2`.
- If the promoted type of the left-hand operand is int, only the five lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator & with the mask value 0x1f (0b11111).[11] The shift distance actually used is therefore always in the range 0 to 31, inclusive.
- If the promoted type of the left-hand operand is long, then only the six lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator & with the mask value 0x3f (0b111111).[11] The shift distance actually used is therefore always in the range 0 to 63, inclusive.

- The value of `n >>> s` is *n* right-shifted *s* bit positions with zero-extension.
- In bit and shift operations, the type `byte` is implicitly converted to `int`. If the byte value is negative, the highest bit is one, then ones are used to fill up the extra bytes in the int. So `byte b1 = -5; int i = b1 | 0x0200;` will result in `i == -5`.

JavaScript

JavaScript uses bitwise operations to evaluate each of two or more units place to 1 or 0.[12]

Pascal

In Pascal, as well as in all its dialects (such as Object Pascal and Standard Pascal), the logical left and right shift operators are "`shl`" and "`shr`", respectively. Even for signed integers, `shr` behaves like a logical shift, and does not copy the sign bit. The number of places to shift is given as the second argument. For example, the following assigns *x* the result of shifting *y* to the left by two bits:

```
x := y shl 2;
```

Other

- popcount, used in cryptography
- count leading zeros

Applications

Bitwise operations are necessary particularly in lower-level programming such as device drivers, low-level graphics, communications protocol packet assembly, and decoding.

Although machines often have efficient built-in instructions for performing arithmetic and logical operations, all these operations can be performed by combining the bitwise operators and zero-testing in various ways.[13] For example, here is a pseudocode implementation of ancient Egyptian multiplication showing how to multiply two arbitrary integers `a` and `b` (`a` greater than `b`) using only bitshifts and addition:

```
c ← 0
while b ≠ 0
    if (b and 1) ≠ 0
        c ← c + a
    left shift a by 1
    right shift b by 1
return c
```

Another example is a pseudocode implementation of addition, showing how to calculate a sum of two integers `a` and `b` using bitwise operators and zero-testing:

```
while a ≠ 0
    c ← b and a
    b ← b xor a
    left shift c by 1
    a ← c
return b
```

Boolean algebra

Sometimes it is useful to simplify complex expressions made up of bitwise operations, for example when writing compilers. The goal of a compiler is to translate a high level programming language into the most efficient machine code possible. Boolean algebra is used to simplify complex bitwise expressions.

AND

- `x & y = y & x`
- `x & (y & z) = (x & y) & z`
- `x & 0xFFFF = x` [14]
- `x & 0 = 0`
- `x & x = x`

OR

- `x | y = y | x`
- `x | (y | z) = (x | y) | z`
- `x | 0 = x`
- `x | 0xFFFF = 0xFFFF`
- `x | x = x`

NOT

- `~(~x) = x`

XOR

- `x ^ y = y ^ x`
- `x ^ (y ^ z) = (x ^ y) ^ z`

- `x ^ 0 = x`
- `x ^ y ^ y = x`
- `x ^ x = 0`
- `x ^ 0xFFFF = ~x`

Additionally, XOR can be composed using the 3 basic operations (AND, OR, NOT)

- `a ^ b = (a | b) & (~a | ~b)`
- `a ^ b = (a & ~b) | (~a & b)`

Others

- `x | (x & y) = x`
- `x & (x | y) = x`
- `~(x | y) = ~x & ~y`
- `~(x & y) = ~x | ~y`
- `x | (y & z) = (x | y) & (x | z)`
- `x & (y | z) = (x & y) | (x & z)`
- `x & (y ^ z) = (x & y) ^ (x & z)`
- `x + y = (x ^ y) + ((x & y) << 1)`
- `x - y = ~(~x + y)`

Inverses and solving equations

It can be hard to solve for variables in Boolean algebra, because unlike regular algebra, several operations do not have inverses. Operations without inverses lose some of the original data bits when they are performed, and it is not possible to recover this missing information.

- Has inverse
  - NOT
  - XOR
  - Rotate left
  - Rotate right
- No inverse
  - AND
  - OR

- Shift left
- Shift right

Order of operations

Operations at the top of this list are executed first. See the main article for a more complete list.

- `( )`
- `~ -` [15]
- `* / %`
- `+ -` [16]
- `<< >>`
- `&`
- `^`
- `|`

See also

- Arithmetic logic unit
- Bit manipulation
- Bitboard
- Bitwise operations in C
- Double dabble
- Find first set
- Karnaugh map
- Logic gate
- Logical operator
- Primitive data type

References

1. "CMicrotek Low-power Design Blog" (http://cmicrotek.com/wordpress_159256135/) . CMicrotek. Retrieved 2015-08-12.
2. JTC1/SC22/WG14 N843 "C programming language" (http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n843.htm) , section 6.5.7

3. "Arithmetic operators - cppreference.com" (http://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_shift_operators) . *en.cppreference.com*. Retrieved 2016-07-06.
4. "INT13-C. Use bitwise operators only on unsigned operands" (https://www.securecoding.cert.org/confluence/display/c/INT13-C.+Use+bitwise+operators+only+on+unsigned+operands) . *CERT: Secure Coding Standards*. Software Engineering Institute, Carnegie Mellon University. Retrieved 2015-09-07.
5. "Operator (C# Reference)" (http://msdn.microsoft.com/en-us/library/xt18et0d%28v=vs.110%29.aspx) . Microsoft. Retrieved 2013-07-14.
6. "Near constant time rotate that does not violate the standards?" (https://stackoverflow.com/q/31387778) . Stack Exchange Network. Retrieved 2015-08-12.
7. "Poor optimization of portable rotate idiom" (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=57157) . GNU GCC Project. Retrieved 2015-08-11.
8. "Circular rotate that does not violate C/C++ standard?" (https://software.intel.com/en-us/forums/topic/580884) . Intel Developer Forums. Retrieved 2015-08-12.
9. "Constant not propagated into inline assembly, results in "constraint 'I' expects an integer constant expression"" (https://llvm.org/bugs/show_bug.cgi?id=24226) . LLVM Project. Retrieved 2015-08-11.
10. The Java Language Specification, section 15.19. Shift Operators (http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.19)
11. "Chapter 15. Expressions" (http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.22.1) . *oracle.com*.
12. "JavaScript Bitwise" (https://www.w3schools.com/js/js_bitwise.asp) . *W3Schools.com*.
13. "Synthesizing arithmetic operations using bit-shifting tricks" (http://bisqwit.iki.fi/story/howto/bitmath/) . Bisqwit.iki.fi. 2014-02-15. Retrieved 2014-03-08.
14. Throughout this article, 0xFFFF means that all the bits in your data type need to be set to 1. The exact number of bits depends on the width of the data type.
15. - is negation here, not subtraction
16. - is subtraction here, not negation

External links

- Online Bitwise Calculator (http://www.miniwebtool.com/bitwise-calculator/) supports Bitwise AND, OR and XOR
- XORcat (https://gitlab.com/viablu/xorcat) , a tool for bitwise-XOR files/streams
- Division using bitshifts (http://www.cs.uiowa.edu/~jones/bcd/divide.html)
- "Bitwise Operations Mod N (https://web.archive.org/web/20150125062918/http://demonstrations.wolfram.com/BitwiseOperationsModN/) " by Enrique Zeleny, Wolfram Demonstrations Project.

- "Plots Of Compositions Of Bitwise Operations (http://demonstrations.wolfram.com/PlotsOfCompositionsOfBitwiseOperations/) " by Enrique Zeleny, The Wolfram Demonstrations Project.