NOTE: use 1L for shifting with larger sized numbers

Hexadecimal

```
0xF -> 1111
0x7 -> 0111
0x3 -> 0011
0x1 -> 0001
```

Websites

https://graphics.stanford.edu/~seander/bithacks.html http://bits.stephan-brumme.com/ http://aggregate.org/MAGIC/

Basic bitwise

Left shift - multiplies by 2

Right shift - divides by 2

Even

```
~(x & 1)
```

Is power of 2

```
x && !(x & x - 1)
```

All 1's

```
~0
```

All 1's unsigned

```
~0u
```

Turn off bits (leaving remainder untouched)

```
x & 0x0000FFFF
```

Turn on bits

```
x | 0x...
```

Create a mask with 0's in the rightmost n bits

```
~0 << n
```

Create a mask with 0's in the leftmost n bits

```
~0 >> n
```

Create a mask with 1's in the rightmost n bits

```
~(~0 << n)
```

Create a mask with 1's in the leftmost n bits

```
~(~0 >> n)
```

Move a desired range to the right end of a word (0 indexed p)

```
(considers index p and n elements to its right and moves to rightmost position, i.
x >> (p + 1 − n)
```

Set a bit (0 indexed)

```
x |= (1 << n)
```

Clear a bit (0 indexed)

```
x &= ~(1 << n)
```

Toggle a bit (0 indexed)

```
x ^= (1 << n)
```

Test a bit (0 indexed)

```
x & (1 << n)
```

Naive left shift with rotate

```
unsigned y = (x << n) | (x >> (32 - n))
```

Safe left shift with rotate

```
unsigned y = (x << n) | (x >> (-n & 31))
```

Clear least significant bit

```
v &= v - 1
```

Mask of least significant bit

```
x &= ~(x - 1)
```

Swap bits i & j

```
x ^= (1L << i) | (1L << j)
```

Count number of set bits (Hamming Weight for bit strings or popcount/population count)

```
// Kernighan
for (c = 0; v; c++) {
    v &= v - 1;
}
```

Sign of an int

```
int sign = -(v < 0);
// not portable
int sign = v >> (sizeof(int) * CHAR_BIT - 1);
```

Detect if integers have opposite signs

```
return ((x ^ y) < 0);
```

Absolute value of an int

```
const int mask = v >> sizeof(int) * CHAR_BIT - 1;
return (x ^ mask) - mask;
```

Find the minimum

```
return y ^ ((x ^ y) & -(x < y)); // min(x, y)
```

Find the maximum

```
return x ^ ((x ^ y) & -(x < y)); // max(x, y)
```

Is power of two

```
return v && !(v & (v - 1));
```

Round up to next power of 2

```
--x;
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
++x;

return x;
```

Properties of XOR

Identity -> a number XOR'ed with 0 returns the number

```
x ^ 0 == x
```

Bitwise negation

```
x ^ ~0 == ~x
```

Inverting the identity

```
x ^ x == 0
```

Associativity

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

Commutativity

```
x ^ y == y ^ x
```

Swap

```
x ^= y
y ^= x
x ^= y

// or
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Bitwise XOR equivalent

```
x ^ y == (~x & y) | (x & ~y)
```

One's complement

```
0010    2
0001    1
0000    0
1111   -0
1110   -1
```

Two's complement

```
0001    1
0000    0
1111   -1
1110   -2
```

One's complement of a number (operator ~)

```
0101 -> 1010
```

Two's complement of a number

```
bitwise NOT (~) of number then add 1 (ignore overflow of two's complement of 0)
returns the numerical complement
```

Hexadecimal

All 1's

All 1's unsigned

Turn off bits (leaving remainder untouched)

Turn on bits

Create a mask with 0's in the rightmost n bits

Create a mask with 0's in the leftmost n bits

Create a mask with 1's in the rightmost n bits

Create a mask with 1's in the leftmost n bits

Move a range field to the right end of a word

Set a bit

Clear a bit

Toggle a bit

Test a bit

Naive left shift with rotate

Safe left shift with rotate

Drop lowest set bit

Clear least significant bit

Swap bits i & j

Count number of set bits (Hamming Weight for bit strings or popcount/population count)

Absolute value of an int

Round up to next power of 2

Properties of XOR

Identity -> a number XOR'ed with 0 returns the number

Bitwise negation

Inverting the identity

Associativity

Commutativity

Swap

Bitwise XOR equivalent

One's complement

Two's complement