Bit Twiddling Hacks

By Sean Eron Anderson
seander@cs.stanford.edu

Contents

About the operation counting methodology

When totaling the number of operations for algorithms here, any C operator is counted as one operation. Intermediate assignments, which need not be written to RAM, are not counted. Of course, this operation counting approach only serves as an approximation of the actual number of machine instructions and CPU time. All operations are assumed to take the same amount of time, which is not true in reality, but CPUs have been heading increasingly in this direction over time. There are many nuances that determine how fast a system will run a given sample of code, such as cache sizes, memory bandwidths, instruction sets, etc. In the end, benchmarking is the best way to determine whether one method is really faster than another, so consider the techniques below as possibilities to test on your target architecture.

Compute the sign of an integer

```
int v;      // we want to find the sign of v
int sign;   // the result goes here

// CHAR_BIT is the number of bits per byte (normally 8).
sign = -(v < 0);  // if v < 0 then -1, else 0.
// or, to avoid branching on CPUs with flag registers (IA32):
sign = -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT - 1));
```

```
// or, for one less instruction (but not portable):
sign = v >> (sizeof(int) * CHAR_BIT - 1);
```

The last expression above evaluates to sign = v >> 31 for 32-bit integers. This is one operation faster than the obvious way, sign = -(v < 0). This trick works because when signed integers are shifted right, the value of the far left bit is copied to the other bits. The far left bit is 1 when the value is negative and 0 otherwise; all 1 bits gives -1. Unfortunately, this behavior is architecture-specific.

Alternatively, if you prefer the result be either -1 or +1, then use:

```
sign = +1 | (v >> (sizeof(int) * CHAR_BIT - 1));  // if v < 0 then -1, else +1
```

On the other hand, if you prefer the result be either -1, 0, or +1, then use:

```
sign = (v != 0) | -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT - 1));
// Or, for more speed but less portability:
sign = (v != 0) | (v >> (sizeof(int) * CHAR_BIT - 1));  // -1, 0, or +1
// Or, for portability, brevity, and (perhaps) speed:
sign = (v > 0) - (v < 0); // -1, 0, or +1
```

If instead you want to know if something is non-negative, resulting in +1 or else 0, then use:

```
sign = 1 ^ ((unsigned int)v >> (sizeof(int) * CHAR_BIT - 1)); // if v < 0 then 0, else 1
```

Caveat: On March 7, 2003, Angus Duggan pointed out that the 1989 ANSI C specification leaves the result of signed right-shift implementation-defined, so on some systems this hack might not work. For greater portability, Toby Speight suggested on September 28, 2005 that CHAR_BIT be used here and throughout rather than assuming bytes were 8 bits long. Angus recommended the more portable versions above, involving casting on March 4, 2006. Rohit Garg suggested the version for non-negative integers on September 12, 2009.

Detect if two integers have opposite signs

```
int x, y;               // input values to compare signs

bool f = ((x ^ y) < 0); // true iff x and y have opposite signs
```

Manfred Weis suggested I add this entry on November 26, 2009.

Compute the integer absolute value (abs) without branching

```
int v;           // we want to find the absolute value of v
unsigned int r;  // the result goes here
int const mask = v >> sizeof(int) * CHAR_BIT - 1;

r = (v + mask) ^ mask;
```

Patented variation:

```
r = (v ^ mask) - mask;
```

Some CPUs don't have an integer absolute value instruction (or the compiler fails to use them). On machines where branching is expensive, the above expression can be faster than the obvious approach, r = (v < 0) ? -(unsigned)v : v, even though the number of operations is the same.

On March 7, 2003, Angus Duggan pointed out that the 1989 ANSI C specification leaves the result of signed right-shift implementation-defined, so on some systems this hack might not work. I've read that ANSI C does not require values to be represented as two's complement, so it may not work for that reason as well (on a diminishingly small number of old machines that still use one's complement). On March 14, 2004, Keith H. Duggar sent me the patented variation above; it is superior to the one I initially came up with, `r=(+1|(v>>(sizeof(int)*CHAR_BIT-1)))*v`,

because a multiply is not used. Unfortunately, this method has been patented in the USA on June 6, 2000 by Vladimir Yu Volkonsky and assigned to Sun Microsystems. On August 13, 2006, Yuriy Kaminskiy told me that the patent is likely invalid because the method was published well before the patent was even filed, such as in How to Optimize for the Pentium Processor by Agner Fog, dated November, 9, 1996. Yuriy also mentioned that this document was translated to Russian in 1997, which Vladimir could have read. Moreover, the Internet Archive also has an old link to it. On January 30, 2007, Peter Kankowski shared with me an abs version he discovered that was inspired by Microsoft's Visual C++ compiler output. It is featured here as the primary solution. On December 6, 2007, Hai Jin complained that the result was signed, so when computing the abs of the most negative value, it was still negative. On April 15, 2008 Andrew Shapira pointed out that the obvious approach could overflow, as it lacked an (unsigned) cast then; for maximum portability he suggested `(v < 0) ? (1 + ((unsigned)(-1-v))) : (unsigned)v`. But citing the ISO C99 spec on July 9, 2008, Vincent Lefèvre convinced me to remove it becasue even on non-2s-complement machines -(unsigned)v will do the right thing. The evaluation of -(unsigned)v first converts the negative value of v to an unsigned by adding 2**N, yielding a 2s complement representation of v's value that I'll call U. Then, U is negated, giving the desired result, -U = 0 - U = 2**N - U = 2**N - (v+2**N) = -v = abs(v).

Compute the minimum (min) or maximum (max) of two integers without branching

```
int x;  // we want to find the minimum of x and y
int y;
int r;  // the result goes here

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

On some rare machines where branching is very expensive and no condition move instructions exist, the above expression might be faster than the obvious approach, r = (x < y) ? x : y, even though it involves two more instructions. (Typically, the obvious approach is best, though.) It works because if x < y, then -(x < y) will be all ones, so r = y ^ (x ^ y) & ~0 = y ^ x ^ y = x. Otherwise, if x >= y, then -(x < y) will be all zeros, so r = y ^ ((x ^ y) & 0) = y. On some machines, evaluating (x < y) as 0 or 1 requires a branch instruction, so there may be no advantage.

To find the maximum, use:

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

Quick and dirty versions:

If you know that INT_MIN <= x - y <= INT_MAX, then you can use the following, which are faster because (x - y) only needs to be evaluated once.

```
r = y + ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // min(x, y)
r = x - ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // max(x, y)
```

Note that the 1989 ANSI C specification doesn't specify the result of signed right-shift, so these aren't portable. If exceptions are thrown on overflows, then the values of x and y should be unsigned or cast to unsigned for the subtractions to avoid unnecessarily throwing an exception, however the right-shift needs a signed operand to produce all one bits when negative, so cast to signed there.

On March 7, 2003, Angus Duggan pointed out the right-shift portability issue. On May 3, 2005, Randal E. Bryant alerted me to the need for the precondition, INT_MIN <= x - y <= INT_MAX, and suggested the non-quick and dirty version as a fix. Both of these issues concern only the quick and dirty version. Nigel Horspoon observed on July 6, 2005 that gcc produced the same code on a Pentium as the obvious solution because of how it evaluates (x < y). On July 9, 2008 Vincent Lefèvre pointed out the potential for overflow exceptions with subtractions in r = y + ((x - y) & -(x < y)), which was the previous version. Timothy B. Terriberry suggested using xor rather than add and subract to avoid casting and the risk of overflows on June 2, 2009.

Determining if an integer is a power of 2

```
unsigned int v; // we want to see if v is a power of 2
bool f;         // the result goes here

f = (v & (v - 1)) == 0;
```

Note that 0 is incorrectly considered a power of 2 here. To remedy this, use:

```
f = v && !(v & (v - 1));
```

Sign extending from a constant bit-width

Sign extension is automatic for built-in types, such as chars and ints. But suppose you have a signed two's complement number, x, that is stored using only b bits. Moreover, suppose you want to convert x to an int, which has more than b bits. A simple copy will work if x is positive, but if negative, the sign must be extended. For example, if we have only 4 bits to store a number, then -3 is represented as 1101 in binary. If we have 8 bits, then -3 is 11111101. The most-significant bit of the 4-bit representation is replicated sinistrally to fill in the destination when we convert to a representation with more bits; this is sign extending. In C, sign extension from a constant bit-width is trivial, since bit fields may be specified in structs or unions. For example, to convert from 5 bits to an full integer:

```
int x; // convert this from using 5 bits to a full int
int r; // resulting sign extended number goes here
struct {signed int x:5;} s;
r = s.x = x;
```

The following is a C++ template function that uses the same language feature to convert from B bits in one operation (though the compiler is generating more, of course).

```
template <typename T, unsigned B>
inline T signextend(const T x)
{
  struct {T x:B;} s;
  return s.x = x;
}

int r = signextend<signed int,5>(x);  // sign extend 5 bit number x to r
```

John Byrd caught a typo in the code (attributed to html formatting) on May 2, 2005. On March 4, 2006, Pat Wood pointed out that the ANSI C standard requires that the bitfield have the keyword "signed" to be signed; otherwise, the sign is undefined.

Sign extending from a variable bit-width

Sometimes we need to extend the sign of a number but we don't know a priori the number of bits, b, in which it is represented. (Or we could be programming in a language like Java, which lacks bitfields.)

```
unsigned b; // number of bits representing the number in x
int x;      // sign extend this b-bit number to r
int r;      // resulting sign-extended number
int const m = 1U << (b - 1); // mask can be pre-computed if b is fixed

x = x & ((1U << b) - 1);  // (Skip this if bits in x above position b are already zero.)
r = (x ^ m) - m;
```

The code above requires four operations, but when the bitwidth is a constant rather than variable, it requires only two fast operations, assuming the upper bits are already zeroes.

A slightly faster but less portable method that doesn't depend on the bits in x above position b being zero is:

```
int const m = CHAR_BIT * sizeof(x) - b;
r = (x << m) >> m;
```

Sean A. Irvine suggested that I add sign extension methods to this page on June 13, 2004, and he provided `m = (1 << (b - 1)) - 1; r = -(x & ~m) | x;` as a starting point from which I optimized to get m = 1U << (b - 1); r = -(x & m) | x. But then on May 11, 2007, Shay Green suggested the version above, which requires one less operation than mine. Vipin Sharma suggested I add a step to deal with situations where x had possible ones in bits other than the b bits we wanted to sign-extend on Oct. 15, 2008. On December 31, 2009 Chris Pirazzi suggested I add the faster version, which requires two operations for constant bit-widths and three for variable widths.

Sign extending from a variable bit-width in 3 operations

The following may be slow on some machines, due to the effort required for multiplication and division. This version is 4 operations. If you know that your initial bit-width, b, is greater than 1, you might do this type of sign extension in 3 operations by using r = (x * multipliers[b]) / multipliers[b], which requires only one array lookup.

```
unsigned b; // number of bits representing the number in x
int x;      // sign extend this b-bit number to r
int r;      // resulting sign-extended number
#define M(B) (1U << ((sizeof(x) * CHAR_BIT) - B)) // CHAR_BIT=bits/byte
static int const multipliers[] =
{
  0,     M(1),  M(2),  M(3),  M(4),  M(5),  M(6),  M(7),
  M(8),  M(9),  M(10), M(11), M(12), M(13), M(14), M(15),
  M(16), M(17), M(18), M(19), M(20), M(21), M(22), M(23),
  M(24), M(25), M(26), M(27), M(28), M(29), M(30), M(31),
  M(32)
}; // (add more if using more than 64 bits)
static int const divisors[] =
{
  1,    ~M(1),  M(2),  M(3),  M(4),  M(5),  M(6),  M(7),
  M(8),  M(9),  M(10), M(11), M(12), M(13), M(14), M(15),
  M(16), M(17), M(18), M(19), M(20), M(21), M(22), M(23),
  M(24), M(25), M(26), M(27), M(28), M(29), M(30), M(31),
  M(32)
}; // (add more for 64 bits)
#undef M
r = (x * multipliers[b]) / divisors[b];
```

The following variation is not portable, but on architectures that employ an arithmetic right-shift, maintaining the sign, it should be fast.

```
const int s = -b; // OR:  sizeof(x) * CHAR_BIT - b;
r = (x << s) >> s;
```

Randal E. Bryant pointed out a bug on May 3, 2005 in an earlier version (that used multipliers[] for divisors[]), where it failed on the case of x=1 and b=1.

Conditionally set or clear bits without branching

```
bool f;         // conditional flag
unsigned int m; // the bit mask
unsigned int w; // the word to modify:  if (f) w |= m; else w &= ~m;

w ^= (-f ^ w) & m;

// OR, for superscalar CPUs:
w = (w & ~m) | (-f & m);
```

On some architectures, the lack of branching can more than make up for what appears to be twice as many operations. For instance, informal speed tests on an AMD Athlon™ XP 2100+ indicated it was 5-10% faster. An Intel Core 2 Duo ran the superscalar version about 16% faster than the first. Glenn Slayden informed me of the first expression on December 11, 2003. Marco Yu shared the superscalar version with me on April 3, 2007 and alerted me to a typo 2 days later.

Conditionally negate a value without branching

If you need to negate only when a flag is false, then use the following to avoid branching:

```
bool fDontNegate;  // Flag indicating we should not negate v.
int v;             // Input value to negate if fDontNegate is false.
int r;             // result = fDontNegate ? v : -v;

r = (fDontNegate ^ (fDontNegate - 1)) * v;
```

If you need to negate only when a flag is true, then use this:

```
bool fNegate;  // Flag indicating if we should negate v.
int v;         // Input value to negate if fNegate is true.
int r;         // result = fNegate ? -v : v;

r = (v ^ -fNegate) + fNegate;
```

Avraham Plotnitzky suggested I add the first version on June 2, 2009. Motivated to avoid the multiply, I came up with the second version on June 8, 2009. Alfonso De Gregorio pointed out that some parens were missing on November 26, 2009, and received a bug bounty.

Merge bits from two values according to a mask

```
unsigned int a;    // value to merge in non-masked bits
unsigned int b;    // value to merge in masked bits
unsigned int mask; // 1 where bits from b should be selected; 0 where from a.
unsigned int r;    // result of (a & ~mask) | (b & mask) goes here

r = a ^ ((a ^ b) & mask);
```

This shaves one operation from the obvious way of combining two sets of bits according to a bit mask. If the mask is a constant, then there may be no advantage.

Ron Jeffery sent this to me on February 9, 2006.

Counting bits set (naive way)

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

for (c = 0; v; v >>= 1)
{
  c += v & 1;
}
```

The naive approach requires one iteration per bit, until no more bits are set. So on a 32-bit word with only the high set, it will go through 32 iterations.

Counting bits set by lookup table

```
static const unsigned char BitsSetTable256[256] =
{
#   define B2(n) n,     n+1,     n+1,     n+2
#   define B4(n) B2(n), B2(n+1), B2(n+1), B2(n+2)
#   define B6(n) B4(n), B4(n+1), B4(n+1), B4(n+2)
    B6(0), B6(1), B6(1), B6(2)
};

unsigned int v; // count the number of bits set in 32-bit value v
unsigned int c; // c is the total bits set in v

// Option 1:
c = BitsSetTable256[v & 0xff] +
    BitsSetTable256[(v >> 8) & 0xff] +
    BitsSetTable256[(v >> 16) & 0xff] +
    BitsSetTable256[v >> 24];

// Option 2:
unsigned char * p = (unsigned char *) &v;
c = BitsSetTable256[p[0]] +
    BitsSetTable256[p[1]] +
    BitsSetTable256[p[2]] +
    BitsSetTable256[p[3]];


// To initially generate the table algorithmically:
BitsSetTable256[0] = 0;
for (int i = 0; i < 256; i++)
{
  BitsSetTable256[i] = (i & 1) + BitsSetTable256[i / 2];
}
```

On July 14, 2009 Hallvard Furuseth suggested the macro compacted table.

Counting bits set, Brian Kernighan's way

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v
for (c = 0; v; c++)
{
  v &= v - 1; // clear the least significant bit set
}
```

Brian Kernighan's method goes through as many iterations as there are set bits. So if we have a 32-bit word with only the high bit set, then it will only go once through the loop.

Published in 1988, the C Programming Language 2nd Ed. (by Brian W. Kernighan and Dennis M. Ritchie) mentions this in exercise 2-9. On April 19, 2006 Don Knuth pointed out to me that this method "was first published by Peter Wegner in CACM 3 (1960), 322. (Also discovered independently by Derrick Lehmer and published in 1964 in a book edited by Beckenbach.)"

Counting bits set in 14, 24, or 32-bit words using 64-bit instructions

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

// option 1, for at most 14-bit values in v:
c = (v * 0x200040008001ULL & 0x111111111111111ULL) % 0xf;

// option 2, for at most 24-bit values in v:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
```

```
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL)
     % 0x1f;

// option 3, for at most 32-bit values in v:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL) %
     0x1f;
c += ((v >> 24) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
```

This method requires a 64-bit CPU with fast modulus division to be efficient. The first option takes only 3 operations; the second option takes 10; and the third option takes 15.

Rich Schroeppel originally created a 9-bit version, similiar to option 1; see the Programming Hacks section of Beeler, M., Gosper, R. W., and Schroeppel, R. HAKMEM. MIT AI Memo 239, Feb. 29, 1972. His method was the inspiration for the variants above, devised by Sean Anderson. Randal E. Bryant offered a couple bug fixes on May 3, 2005. Bruce Dawson tweaked what had been a 12-bit version and made it suitable for 14 bits using the same number of operations on Feburary 1, 2007.

Counting bits set, in parallel

```
unsigned int v; // count bits set in this (32-bit value)
unsigned int c; // store the total here
static const int S[] = {1, 2, 4, 8, 16}; // Magic Binary Numbers
static const int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF, 0x0000FFFF};

c = v - ((v >> 1) & B[0]);
c = ((c >> S[1]) & B[1]) + (c & B[1]);
c = ((c >> S[2]) + c) & B[2];
c = ((c >> S[3]) + c) & B[3];
c = ((c >> S[4]) + c) & B[4];
```

The B array, expressed as binary, is:

```
B[0] = 0x55555555 = 01010101 01010101 01010101 01010101
B[1] = 0x33333333 = 00110011 00110011 00110011 00110011
B[2] = 0x0F0F0F0F = 00001111 00001111 00001111 00001111
B[3] = 0x00FF00FF = 00000000 11111111 00000000 11111111
B[4] = 0x0000FFFF = 00000000 00000000 11111111 11111111
```

We can adjust the method for larger integer sizes by continuing with the patterns for the *Binary Magic Numbers*, B and S. If there are k bits, then we need the arrays S and B to be ceil(lg(k)) elements long, and we must compute the same number of expressions for c as S or B are long. For a 32-bit v, 16 operations are used.

The best method for counting bits in a 32-bit integer v is the following:

```
v = v - ((v >> 1) & 0x55555555);                    // reuse input as temporary
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);     // temp
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24; // count
```

The best bit counting method takes only 12 operations, which is the same as the lookup-table method, but avoids the memory and potential cache misses of a table. It is a hybrid between the purely parallel method above and the earlier methods using multiplies (in the section on counting bits with 64-bit instructions), though it doesn't use 64-bit instructions. The counts of bits set in the bytes is done in parallel, and the sum total of the bits set in the bytes is computed by multiplying by 0x1010101 and shifting right 24 bits.

A generalization of the best bit counting method to integers of bit-widths upto 128 (parameterized by type T) is this:

```
v = v - ((v >> 1) & (T)~(T)0/3);                           // temp
v = (v & (T)~(T)0/15*3) + ((v >> 2) & (T)~(T)0/15*3);      // temp
v = (v + (v >> 4)) & (T)~(T)0/255*15;                      // temp
c = (T)(v * ((T)~(T)0/255)) >> (sizeof(T) - 1) * CHAR_BIT; // count
```

See Ian Ashdown's nice newsgroup post for more information on counting the number of bits set (also known as *sideways addition*). The best bit counting method was brought to my attention on October 5, 2005 by Andrew Shapira; he found it in pages 187-188 of Software Optimization Guide for AMD Athlon™ 64 and Opteron™ Processors. Charlie Gordon suggested a way to shave off one operation from the purely parallel version on December 14, 2005, and Don Clugston trimmed three more from it on December 30, 2005. I made a typo with Don's suggestion that Eric Cole spotted on January 8, 2006. Eric later suggested the arbitrary bit-width generalization to the best method on November 17, 2006. On April 5, 2007, Al Williams observed that I had a line of dead code at the top of the first method.

Count bits set (rank) from the most-significant bit upto a given position

The following finds the the rank of a bit, meaning it returns the sum of bits that are set to 1 from the most-signficant bit downto the bit at the given position.

```
uint64_t v;       // Compute the rank (bits set) in v from the MSB to pos.
unsigned int pos; // Bit position to count bits upto.
uint64_t r;       // Resulting rank of bit at pos goes here.

// Shift out bits after given position.
r = v >> (sizeof(v) * CHAR_BIT - pos);
// Count set bits in parallel.
// r = (r & 0x5555...) + ((r >> 1) & 0x5555...);
r = r - ((r >> 1) & ~0UL/3);
// r = (r & 0x3333...) + ((r >> 2) & 0x3333...);
r = (r & ~0UL/5) + ((r >> 2) & ~0UL/5);
// r = (r & 0x0f0f...) + ((r >> 4) & 0x0f0f...);
r = (r + (r >> 4)) & ~0UL/17;
// r = r % 255;
r = (r * (~0UL/255)) >> ((sizeof(v) - 1) * CHAR_BIT);
```

Juha Järvi sent this to me on November 21, 2009 as an inverse operation to the computing the bit position with the given rank, which follows.

Select the bit position (from the most-significant bit) with the given count (rank)

The following 64-bit code selects the position of the r^{th} 1 bit when counting from the left. In other words if we start at the most significant bit and proceed to the right, counting the number of bits set to 1 until we reach the desired rank, r, then the position where we stop is returned. If the rank requested exceeds the count of bits set, then 64 is returned. The code may be modified for 32-bit or counting from the right.

```
uint64_t v;          // Input value to find position with rank r.
unsigned int r;      // Input: bit's desired rank [1-64].
unsigned int s;      // Output: Resulting position of bit with rank r [1-64]
uint64_t a, b, c, d; // Intermediate temporaries for bit count.
unsigned int t;      // Bit count temporary.

// Do a normal parallel bit count for a 64-bit integer,
// but store all intermediate steps.
// a = (v & 0x5555...) + ((v >> 1) & 0x5555...);
a =  v - ((v >> 1) & ~0UL/3);
// b = (a & 0x3333...) + ((a >> 2) & 0x3333...);
b = (a & ~0UL/5) + ((a >> 2) & ~0UL/5);
// c = (b & 0x0f0f...) + ((b >> 4) & 0x0f0f...);
c = (b + (b >> 4)) & ~0UL/0x11;
// d = (c & 0x00ff...) + ((c >> 8) & 0x00ff...);
d = (c + (c >> 8)) & ~0UL/0x101;
t = (d >> 32) + (d >> 48);
// Now do branchless select!
s  = 64;
// if (r > t) {s -= 32; r -= t;}
```

```
  s -= ((t - r) & 256) >> 3; r -= (t & ((t - r) >> 8));
  t  = (d >> (s - 16)) & 0xff;
  // if (r > t) {s -= 16; r -= t;}
  s -= ((t - r) & 256) >> 4; r -= (t & ((t - r) >> 8));
  t  = (c >> (s - 8)) & 0xf;
  // if (r > t) {s -= 8; r -= t;}
  s -= ((t - r) & 256) >> 5; r -= (t & ((t - r) >> 8));
  t  = (b >> (s - 4)) & 0x7;
  // if (r > t) {s -= 4; r -= t;}
  s -= ((t - r) & 256) >> 6; r -= (t & ((t - r) >> 8));
  t  = (a >> (s - 2)) & 0x3;
  // if (r > t) {s -= 2; r -= t;}
  s -= ((t - r) & 256) >> 7; r -= (t & ((t - r) >> 8));
  t  = (v >> (s - 1)) & 0x1;
  // if (r > t) s--;
  s -= ((t - r) & 256) >> 8;
  s = 65 - s;
```

If branching is fast on your target CPU, consider uncommenting the if-statements and commenting the lines that follow them.

Juha Järvi sent this to me on November 21, 2009.

Computing parity the naive way

```
unsigned int v;       // word value to compute the parity of
bool parity = false;  // parity will be the parity of v

while (v)
{
  parity = !parity;
  v = v & (v - 1);
}
```

The above code uses an approach like Brian Kernigan's bit counting, above. The time it takes is proportional to the number of bits set.

Compute parity by lookup table

```
static const bool ParityTable256[256] =
{
#   define P2(n) n, n^1, n^1, n
#   define P4(n) P2(n), P2(n^1), P2(n^1), P2(n)
#   define P6(n) P4(n), P4(n^1), P4(n^1), P4(n)
    P6(0), P6(1), P6(1), P6(0)
};

unsigned char b;  // byte value to compute the parity of
bool parity = ParityTable256[b];

// OR, for 32-bit words:
unsigned int v;
v ^= v >> 16;
v ^= v >> 8;
bool parity = ParityTable256[v & 0xff];

// Variation:
unsigned char * p = (unsigned char *) &v;
parity = ParityTable256[p[0] ^ p[1] ^ p[2] ^ p[3]];
```

Randal E. Bryant encouraged the addition of the (admittedly) obvious last variation with variable p on May 3, 2005. Bruce Rawles found a typo in an instance of the table variable's name on September 27, 2005, and he received a $10 bug bounty. On October 9, 2006, Fabrice Bellard suggested the 32-bit variations above, which require only one table lookup; the previous version had four lookups (one per byte) and were slower. On July 14, 2009 Hallvard Furuseth suggested the macro compacted table.

Compute parity of a byte using 64-bit multiply and modulus division

```
unsigned char b;  // byte value to compute the parity of
bool parity =
  (((b * 0x0101010101010101ULL) & 0x8040201008040201ULL) % 0x1FF) & 1;
```

The method above takes around 4 operations, but only works on bytes.

Compute parity of word with a multiply

The following method computes the parity of the 32-bit value in only 8 operations using a multiply.

```
unsigned int v; // 32-bit word
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x11111111U) * 0x11111111U;
return (v >> 28) & 1;
```

Also for 64-bits, 8 operations are still enough.

```
unsigned long long v; // 64-bit word
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x1111111111111111UL) * 0x1111111111111111UL;
return (v >> 60) & 1;
```

Andrew Shapira came up with this and sent it to me on Sept. 2, 2007.

Compute parity in parallel

```
unsigned int v;  // word value to compute the parity of
v ^= v >> 16;
v ^= v >> 8;
v ^= v >> 4;
v &= 0xf;
return (0x6996 >> v) & 1;
```

The method above takes around 9 operations, and works for 32-bit words. It may be optimized to work just on bytes in 5 operations by removing the two lines immediately following "unsigned int v;". The method first shifts and XORs the eight nibbles of the 32-bit value together, leaving the result in the lowest nibble of v. Next, the binary number 0110 1001 1001 0110 (0x6996 in hex) is shifted to the right by the value represented in the lowest nibble of v. This number is like a miniature 16-bit parity-table indexed by the low four bits in v. The result has the parity of v in bit 1, which is masked and returned.

Thanks to Mathew Hendry for pointing out the shift-lookup idea at the end on Dec. 15, 2002. That optimization shaves two operations off using only shifting and XORing to find the parity.

Swapping values with subtraction and addition

```
#define SWAP(a, b) ((&(a) == &(b)) || \
                    (((a) -= (b)), ((b) += (a)), ((a) = (b) - (a))))
```

This swaps the values of a and b *without using a temporary variable*. The initial check for a and b being the same location in memory may be omitted when you know this can't happen. (The compiler may omit it anyway as an optimization.) If you enable overflows exceptions, then pass unsigned values so an exception isn't thrown. The XOR method that follows may be slightly faster on some machines. Don't use this with floating-point numbers (unless you operate on their raw integer representations).

Sanjeev Sivasankaran suggested I add this on June 12, 2007. Vincent Lefèvre pointed out the potential for overflow exceptions on July 9, 2008

Swapping values with XOR

```
#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))
```

This is an old trick to exchange the values of the variables a and b *without using extra space for a temporary variable*.

On January 20, 2005, Iain A. Fleming pointed out that the macro above doesn't work when you swap with the same memory location, such as SWAP(a[i], a[j]) with i == j. So if that may occur, consider defining the macro as (((a) == (b)) || ((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))). On July 14, 2009, Hallvard Furuseth suggested that on some machines, (((a) ^ (b)) && ((b) ^= (a) ^= (b), (a) ^= (b))) might be faster, since the (a) ^ (b) expression is reused.

Swapping individual bits with XOR

```
unsigned int i, j; // positions of bit sequences to swap
unsigned int n;    // number of consecutive bits in each sequence
unsigned int b;    // bits to swap reside in b
unsigned int r;    // bit-swapped result goes here

unsigned int x = ((b >> i) ^ (b >> j)) & ((1U << n) - 1); // XOR temporary
r = b ^ ((x << i) | (x << j));
```

As an example of swapping ranges of bits suppose we have have b = **001**0**111**1 (expressed in binary) and we want to swap the n = 3 consecutive bits starting at i = 1 (the second bit from the right) with the 3 consecutive bits starting at j = 5; the result would be r = **111**0**001**1 (binary).

This method of swapping is similar to the general purpose XOR swap trick, but intended for operating on individual bits. The variable x stores the result of XORing the pairs of bit values we want to swap, and then the bits are set to the result of themselves XORed with x. Of course, the result is undefined if the sequences overlap.

On July 14, 2009 Hallvard Furuseth suggested that I change the 1 << n to 1U << n because the value was being assigned to an unsigned and to avoid shifting into a sign bit.

Reverse bits the obvious way

```
unsigned int v;     // input bits to be reversed
unsigned int r = v; // r will be reversed bits of v; first get LSB of v
int s = sizeof(v) * CHAR_BIT - 1; // extra shift needed at end

for (v >>= 1; v; v >>= 1)
{
  r <<= 1;
  r |= v & 1;
  s--;
```

```
}
r <<= s; // shift when v's highest bits are zero
```

On October 15, 2004, Michael Hoisie pointed out a bug in the original version. Randal E. Bryant suggested removing an extra operation on May 3, 2005. Behdad Esfabod suggested a slight change that eliminated one iteration of the loop on May 18, 2005. Then, on February 6, 2007, Liyong Zhou suggested a better version that loops while v is not 0, so rather than iterating over all bits it stops early.

Reverse bits in word by lookup table

```
static const unsigned char BitReverseTable256[256] =
{
#   define R2(n)     n,     n + 2*64,     n + 1*64,     n + 3*64
#   define R4(n) R2(n), R2(n + 2*16), R2(n + 1*16), R2(n + 3*16)
#   define R6(n) R4(n), R4(n + 2*4 ), R4(n + 1*4 ), R4(n + 3*4 )
    R6(0), R6(2), R6(1), R6(3)
};

unsigned int v; // reverse 32-bit value, 8 bits at time
unsigned int c; // c will get v reversed

// Option 1:
c = (BitReverseTable256[v & 0xff] << 24) |
    (BitReverseTable256[(v >> 8) & 0xff] << 16) |
    (BitReverseTable256[(v >> 16) & 0xff] << 8) |
    (BitReverseTable256[(v >> 24) & 0xff]);

// Option 2:
unsigned char * p = (unsigned char *) &v;
unsigned char * q = (unsigned char *) &c;
q[3] = BitReverseTable256[p[0]];
q[2] = BitReverseTable256[p[1]];
q[1] = BitReverseTable256[p[2]];
q[0] = BitReverseTable256[p[3]];
```

The first method takes about 17 operations, and the second takes about 12, assuming your CPU can load and store bytes easily.

On July 14, 2009 Hallvard Furuseth suggested the macro compacted table.

Reverse the bits in a byte with 3 operations (64-bit multiply and modulus division):

```
unsigned char b; // reverse this (8-bit) byte

b = (b * 0x0202020202ULL & 0x010884422010ULL) % 1023;
```

The multiply operation creates five separate copies of the 8-bit byte pattern to fan-out into a 64-bit value. The AND operation selects the bits that are in the correct (reversed) positions, relative to each 10-bit groups of bits. The multiply and the AND operations copy the bits from the original byte so they each appear in only one of the 10-bit sets. The reversed positions of the bits from the original byte coincide with their relative positions within any 10-bit set. The last step, which involves modulus division by 2^10 - 1, has the effect of merging together each set of 10 bits (from positions 0-9, 10-19, 20-29, ...) in the 64-bit value. They do not overlap, so the addition steps underlying the modulus division behave like or operations.

This method was attributed to Rich Schroeppel in the Programming Hacks section of Beeler, M., Gosper, R. W., and Schroeppel, R. HAKMEM. MIT AI Memo 239, Feb. 29, 1972.

Reverse the bits in a byte with 4 operations (64-bit multiply, no division):

```
unsigned char b; // reverse this byte

b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;
```

The following shows the flow of the bit values with the boolean variables `a, b, c, d, e, f, g,` and `h`, which comprise an 8-bit byte. Notice how the first multiply fans out the bit pattern to multiple copies, while the last multiply combines them in the fifth byte from the right.

```
                                                                                                  abcd efgh (-> hgfe dcba)
*                                                 1000 0000  0010 0000  0000 1000  0000 0010 (0x80200802)
--------------------------------------------------------------------------------------------------------
                                        0abc defg  h00a bcde  fgh0 0abc  defg h00a  bcde fgh0
&                                       0000 1000  1000 0100  0100 0010  0010 0001  0001 0000 (0x0884422110)
--------------------------------------------------------------------------------------------------------
                                        0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
*                                       0000 0001  0000 0001  0000 0001  0000 0001  0000 0001 (0x0101010101)
--------------------------------------------------------------------------------------------------------
                                        0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                              0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                    0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
          0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
--------------------------------------------------------------------------------------------------------
0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba  hgfe 0cba  0gfe 00ba  00fe 000a  000e 0000
>> 32
--------------------------------------------------------------------------------------------------------
                                        0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba
&                                                                                   1111 1111
--------------------------------------------------------------------------------------------------------
                                                                                    hgfe dcba
```

Note that the last two steps can be combined on some processors because the registers can be accessed as bytes; just multiply so that a register stores the upper 32 bits of the result and the take the low byte. Thus, it may take only 6 operations.

Devised by Sean Anderson, July 13, 2001.

Reverse the bits in a byte with 7 operations (no 64-bit):

```
b = ((b * 0x0802LU & 0x22110LU) | (b * 0x8020LU & 0x88440LU)) * 0x10101LU >> 16;
```

Make sure you assign or cast the result to an unsigned char to remove garbage in the higher bits. Devised by Sean Anderson, July 13, 2001. Typo spotted and correction supplied by Mike Keith, January 3, 2002.

Reverse an N-bit quantity in parallel in 5 * lg(N) operations:

```
unsigned int v; // 32-bit word to reverse bit order

// swap odd and even bits
v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
// swap consecutive pairs
v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
// swap nibbles ...
v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
// swap bytes
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
// swap 2-byte long pairs
v = ( v >> 16             ) | ( v               << 16);
```

The following variation is also O(lg(N)), however it requires more operations to reverse v. Its virtue is in taking less slightly memory by computing the constants on the fly.

```
unsigned int s = sizeof(v) * CHAR_BIT; // bit size; must be power of 2
unsigned int mask = ~0;
while ((s >>= 1) > 0)
{
  mask ^= (mask << s);
```

```
  v = ((v >> s) & mask) | ((v << s) & ~mask);
}
```

These methods above are best suited to situations where N is large. If you use the above with 64-bit ints (or larger), then you need to add more lines (following the pattern); otherwise only the lower 32 bits will be reversed and the result will be in the lower 32 bits.

See Dr. Dobb's Journal 1983, Edwin Freed's article on Binary Magic Numbers for more information. The second variation was suggested by Ken Raeburn on September 13, 2005. Veldmeijer mentioned that the first version could do without ANDS in the last line on March 19, 2006.

Compute modulus division by 1 << s without a division operator

```
const unsigned int n;          // numerator
const unsigned int s;
const unsigned int d = 1U << s; // So d will be one of: 1, 2, 4, 8, 16, 32, ...
unsigned int m;                // m will be n % d
m = n & (d - 1);
```

Most programmers learn this trick early, but it was included for the sake of completeness.

Compute modulus division by (1 << s) - 1 without a division operator

```
unsigned int n;                     // numerator
const unsigned int s;               // s > 0
const unsigned int d = (1 << s) - 1; // so d is either 1, 3, 7, 15, 31, ...).
unsigned int m;                     // n % d goes here.

for (m = n; n > d; n = m)
{
  for (m = 0; n; n >>= s)
  {
    m += n & d;
  }
}
// Now m is a value from 0 to d, but since with modulus division
// we want m to be 0 when it is d.
m = m == d ? 0 : m;
```

This method of modulus division by an integer that is one less than a power of 2 takes at most 5 + (4 + 5 * ceil(N / s)) * ceil(lg(N / s)) operations, where N is the number of bits in the numerator. In other words, it takes at most O(N * lg(N)) time.

Devised by Sean Anderson, August 15, 2001. Before Sean A. Irvine corrected me on June 17, 2004, I mistakenly commented that we could alternatively assign `m = ((m + 1) & d) - 1;` at the end. Michael Miller spotted a typo in the code April 25, 2005.

Compute modulus division by (1 << s) - 1 in parallel without a division operator

```
// The following is for a word size of 32 bits!

static const unsigned int M[] =
{
  0x00000000, 0x55555555, 0x33333333, 0xc71c71c7,
  0x0f0f0f0f, 0xc1f07c1f, 0x3f03f03f, 0xf01fc07f,
  0x00ff00ff, 0x07fc01ff, 0x3ff003ff, 0xffc007ff,
  0xff000fff, 0xfc001fff, 0xf0003fff, 0xc0007fff,
```

```
    0x0000ffff, 0x0001ffff, 0x0003ffff, 0x0007ffff,
    0x000fffff, 0x001fffff, 0x003fffff, 0x007fffff,
    0x00ffffff, 0x01ffffff, 0x03ffffff, 0x07ffffff,
    0x0fffffff, 0x1fffffff, 0x3fffffff, 0x7fffffff
  };

  static const unsigned int Q[][6] =
  {
    { 0,  0,  0,  0,  0,  0}, {16,  8,  4,  2,  1,  1}, {16,  8,  4,  2,  2,  2},
    {15,  6,  3,  3,  3,  3}, {16,  8,  4,  4,  4,  4}, {15,  5,  5,  5,  5,  5},
    {12,  6,  6,  6 , 6,  6}, {14,  7,  7,  7,  7,  7}, {16,  8,  8,  8,  8,  8},
    { 9,  9,  9,  9,  9,  9}, {10, 10, 10, 10, 10, 10}, {11, 11, 11, 11, 11, 11},
    {12, 12, 12, 12, 12, 12}, {13, 13, 13, 13, 13, 13}, {14, 14, 14, 14, 14, 14},
    {15, 15, 15, 15, 15, 15}, {16, 16, 16, 16, 16, 16}, {17, 17, 17, 17, 17, 17},
    {18, 18, 18, 18, 18, 18}, {19, 19, 19, 19, 19, 19}, {20, 20, 20, 20, 20, 20},
    {21, 21, 21, 21, 21, 21}, {22, 22, 22, 22, 22, 22}, {23, 23, 23, 23, 23, 23},
    {24, 24, 24, 24, 24, 24}, {25, 25, 25, 25, 25, 25}, {26, 26, 26, 26, 26, 26},
    {27, 27, 27, 27, 27, 27}, {28, 28, 28, 28, 28, 28}, {29, 29, 29, 29, 29, 29},
    {30, 30, 30, 30, 30, 30}, {31, 31, 31, 31, 31, 31}
  };

  static const unsigned int R[][6] =
  {
    {0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000},
    {0x0000ffff, 0x000000ff, 0x0000000f, 0x00000003, 0x00000001, 0x00000001},
    {0x0000ffff, 0x000000ff, 0x0000000f, 0x00000003, 0x00000003, 0x00000003},
    {0x00007fff, 0x0000003f, 0x00000007, 0x00000007, 0x00000007, 0x00000007},
    {0x0000ffff, 0x000000ff, 0x0000000f, 0x0000000f, 0x0000000f, 0x0000000f},
    {0x00007fff, 0x0000001f, 0x0000001f, 0x0000001f, 0x0000001f, 0x0000001f},
    {0x00000fff, 0x0000003f, 0x0000003f, 0x0000003f, 0x0000003f, 0x0000003f},
    {0x00003fff, 0x0000007f, 0x0000007f, 0x0000007f, 0x0000007f, 0x0000007f},
    {0x0000ffff, 0x000000ff, 0x000000ff, 0x000000ff, 0x000000ff, 0x000000ff},
    {0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff},
    {0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff},
    {0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff},
    {0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff},
    {0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff},
    {0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff},
    {0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff},
    {0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff},
    {0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff},
    {0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff},
    {0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff},
    {0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff},
    {0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff},
    {0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff},
    {0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff},
    {0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff},
    {0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff},
    {0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff},
    {0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff},
    {0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff},
    {0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff},
    {0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff},
    {0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff}
  };

  unsigned int n;       // numerator
  const unsigned int s; // s > 0
  const unsigned int d = (1 << s) - 1; // so d is either 1, 3, 7, 15, 31, ...).
  unsigned int m;       // n % d goes here.

  m = (n & M[s]) + ((n >> s) & M[s]);

  for (const unsigned int * q = &Q[s][0], * r = &R[s][0]; m > d; q++, r++)
  {
    m = (m >> *q) + (m & *r);
```

```
}
m = m == d ? 0 : m; // OR, less portably: m = m & -((signed)(m - d) >> s);
```

This method of finding modulus division by an integer that is one less than a power of 2 takes at most O(lg(N)) time, where N is the number of bits in the numerator (32 bits, for the code above). The number of operations is at most 12 + 9 * ceil(lg(N)). The tables may be removed if you know the denominator at compile time; just extract the few relevent entries and unroll the loop. It may be easily extended to more bits.

It finds the result by summing the values in base (1 << s) in parallel. First every other base (1 << s) value is added to the previous one. Imagine that the result is written on a piece of paper. Cut the paper in half, so that half the values are on each cut piece. Align the values and sum them onto a new piece of paper. Repeat by cutting this paper in half (which will be a quarter of the size of the previous one) and summing, until you cannot cut further. After performing lg(N/s/2) cuts, we cut no more; just continue to add the values and put the result onto a new piece of paper as before, while there are at least two s-bit values.

Devised by Sean Anderson, August 20, 2001. A typo was spotted by Randy E. Bryant on May 3, 2005 (after pasting the code, I had later added "unsinged" to a variable declaration). As in the previous hack, I mistakenly commented that we could alternatively assign `m = ((m + 1) & d) - 1;` at the end, and Don Knuth corrected me on April 19, 2006 and suggested `m = m & -((signed)(m - d) >> s)`. On June 18, 2009 Sean Irvine proposed a change that used `((n >> s) & M[s])` instead of `((n & ~M[s]) >> s)`, which typically requires fewer operations because the M[s] constant is already loaded.

Find the log base 2 of an integer with the MSB N set in O(N) operations (the obvious way)

```
unsigned int v; // 32-bit word to find the log base 2 of
unsigned int r = 0; // r will be lg(v)

while (v >>= 1) // unroll for more speed...
{
  r++;
}
```

The log base 2 of an integer is the same as the position of the highest bit set (or most significant bit set, MSB). The following log base 2 methods are faster than this one.

Find the integer log base 2 of an integer with an 64-bit IEEE float

```
int v; // 32-bit integer to find the log base 2 of
int r; // result of log_2(v) goes here
union { unsigned int u[2]; double d; } t; // temp

t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] = 0x43300000;
t.u[__FLOAT_WORD_ORDER!=LITTLE_ENDIAN] = v;
t.d -= 4503599627370496.0;
r = (t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] >> 20) - 0x3FF;
```

The code above loads a 64-bit (IEEE-754 floating-point) double with a 32-bit integer (with no paddding bits) by storing the integer in the mantissa while the exponent is set to 2^{52}. From this newly minted double, 2^{52} (expressed as a double) is subtracted, which sets the resulting exponent to the log base 2 of the input value, v. All that is left is shifting the exponent bits into position (20 bits right) and subtracting the bias, 0x3FF (which is 1023 decimal). This technique only takes 5 operations, but many CPUs are slow at manipulating doubles, and the endianess of the architecture must be accommodated.

Eric Cole sent me this on January 15, 2006. Evan Felix pointed out a typo on April 4, 2006. Vincent Lefèvre told me on July 9, 2008 to change the endian check to use the float's endian, which could differ from the integer's endian.

Find the log base 2 of an integer with a lookup table

```
static const char LogTable256[256] =
{
#define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
    -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
    LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
    LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
};

unsigned int v; // 32-bit word to find the log of
unsigned r;     // r will be lg(v)
register unsigned int t, tt; // temporaries

if (tt = v >> 16)
{
  r = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
}
else
{
  r = (t = v >> 8) ? 8 + LogTable256[t] : LogTable256[v];
}
```

The lookup table method takes only about 7 operations to find the log of a 32-bit value. If extended for 64-bit quantities, it would take roughly 9 operations. Another operation can be trimmed off by using four tables, with the possible additions incorporated into each. Using int table elements may be faster, depending on your architecture.

The code above is tuned to uniformly distributed *output* values. If your *inputs* are evenly distributed across all 32-bit values, then consider using the following:

```
if (tt = v >> 24)
{
  r = 24 + LogTable256[tt];
}
else if (tt = v >> 16)
{
  r = 16 + LogTable256[tt];
}
else if (tt = v >> 8)
{
  r = 8 + LogTable256[tt];
}
else
{
  r = LogTable256[v];
}
```

To initially generate the log table algorithmically:

```
LogTable256[0] = LogTable256[1] = 0;
for (int i = 2; i < 256; i++)
{
  LogTable256[i] = 1 + LogTable256[i / 2];
}
LogTable256[0] = -1; // if you want log(0) to return -1
```

Behdad Esfahbod and I shaved off a fraction of an operation (on average) on May 18, 2005. Yet another fraction of an operation was removed on November 14, 2006 by Emanuel Hoogeveen. The variation that is tuned to evenly distributed input values was suggested by David A. Butterfield on September 19, 2008. Venkat Reddy told me on January 5, 2009 that log(0) should return -1 to indicate an error, so I changed the first entry in the table to that.

Find the log base 2 of an N-bit integer in O(lg(N)) operations

```
unsigned int v;  // 32-bit value to find the log2 of
const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000};
const unsigned int S[] = {1, 2, 4, 8, 16};
int i;

register unsigned int r = 0; // result of log2(v) will go here
for (i = 4; i >= 0; i--) // unroll for speed...
{
  if (v & b[i])
  {
    v >>= S[i];
    r |= S[i];
  }
}


// OR (IF YOUR CPU BRANCHES SLOWLY):

unsigned int v;          // 32-bit value to find the log2 of
register unsigned int r; // result of log2(v) will go here
register unsigned int shift;

r =     (v > 0xFFFF) << 4; v >>= r;
shift = (v > 0xFF  ) << 3; v >>= shift; r |= shift;
shift = (v > 0xF   ) << 2; v >>= shift; r |= shift;
shift = (v > 0x3   ) << 1; v >>= shift; r |= shift;
                                        r |= (v >> 1);


// OR (IF YOU KNOW v IS A POWER OF 2):

unsigned int v;  // 32-bit value to find the log2 of
static const unsigned int b[] = {0xAAAAAAAA, 0xCCCCCCCC, 0xF0F0F0F0,
                                 0xFF00FF00, 0xFFFF0000};
register unsigned int r = (v & b[0]) != 0;
for (i = 4; i > 0; i--) // unroll for speed...
{
  r |= ((v & b[i]) != 0) << i;
}
```

Of course, to extend the code to find the log of a 33- to 64-bit number, we would append another element, 0xFFFFFFFF00000000, to b, append 32 to S, and loop from 5 to 0. This method is much slower than the earlier table-lookup version, but if you don't want big table or your architecture is slow to access memory, it's a good choice. The second variation involves slightly more operations, but it may be faster on machines with high branch costs (e.g. PowerPC).

The second version was sent to me by Eric Cole on January 7, 2006. Andrew Shapira subsequently trimmed a few operations off of it and sent me his variation (above) on Sept. 1, 2007. The third variation was suggested to me by John Owens on April 24, 2002; it's faster, but *it is only suitable when the input is known to be a power of 2*. On May 25, 2003, Ken Raeburn suggested improving the general case by using smaller numbers for b[], which load faster on some architectures (for instance if the word size is 16 bits, then only one load instruction may be needed). These values work for the general version, but not for the special-case version below it, where v is a power of 2; Glenn Slayden brought this oversight to my attention on December 12, 2003.

Find the log base 2 of an N-bit integer in O(lg(N)) operations with multiply and lookup

```
uint32_t v; // find the log base 2 of 32-bit v
int r;      // result goes here

static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 9, 1, 10, 13, 21, 2, 29, 11, 14, 16, 18, 22, 25, 3, 30,
  8, 12, 20, 28, 15, 17, 24, 7, 19, 27, 23, 6, 26, 5, 4, 31
```

```
};

v |= v >> 1; // first round down to one less than a power of 2
v |= v >> 2;
v |= v >> 4;
v |= v >> 8;
v |= v >> 16;

r = MultiplyDeBruijnBitPosition[(uint32_t)(v * 0x07C4ACDDU) >> 27];
```

The code above computes the log base 2 of a 32-bit integer with a small table lookup and multiply. It requires only 13 operations, compared to (up to) 20 for the previous method. The purely table-based method requires the fewest operations, but this offers a reasonable compromise between table size and speed.

If you know that v is a power of 2, then you only need the following:

```
static const int MultiplyDeBruijnBitPosition2[32] =
{
  0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
  31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
};
r = MultiplyDeBruijnBitPosition2[(uint32_t)(v * 0x077CB531U) >> 27];
```

Eric Cole devised this January 8, 2006 after reading about the entry below to round up to a power of 2 and the method below for computing the number of trailing bits with a multiply and lookup using a DeBruijn sequence. On December 10, 2009, Mark Dickinson shaved off a couple operations by requiring v be rounded up to one less than the next power of 2 rather than the power of 2.

Find integer log base 10 of an integer

```
unsigned int v; // non-zero 32-bit integer value to compute the log base 10 of
int r;          // result goes here
int t;          // temporary

static unsigned int const PowersOf10[] =
    {1, 10, 100, 1000, 10000, 100000,
     1000000, 10000000, 100000000, 1000000000};

t = (IntegerLogBase2(v) + 1) * 1233 >> 12; // (use a lg2 method from above)
r = t - (v < PowersOf10[t]);
```

The integer log base 10 is computed by first using one of the techniques above for finding the log base 2. By the relationship $\log_{10}(v) = \log_2(v) / \log_2(10)$, we need to multiply it by $1/\log_2(10)$, which is approximately 1233/4096, or 1233 followed by a right shift of 12. Adding one is needed because the IntegerLogBase2 rounds down. Finally, since the value t is only an approximation that may be off by one, the exact value is found by subtracting the result of v < PowersOf10[t].

This method takes 6 more operations than IntegerLogBase2. It may be sped up (on machines with fast memory access) by modifying the log base 2 table-lookup method above so that the entries hold what is computed for t (that is, pre-add, -mulitply, and -shift). Doing so would require a total of only 9 operations to find the log base 10, assuming 4 tables were used (one for each byte of v).

Eric Cole suggested I add a version of this on January 7, 2006.

Find integer log base 10 of an integer the obvious way

```
unsigned int v; // non-zero 32-bit integer value to compute the log base 10 of
int r;          // result goes here
```

```
r = (v >= 1000000000) ? 9 : (v >= 100000000) ? 8 : (v >= 10000000) ? 7 :
    (v >= 1000000) ? 6 : (v >= 100000) ? 5 : (v >= 10000) ? 4 :
    (v >= 1000) ? 3 : (v >= 100) ? 2 : (v >= 10) ? 1 : 0;
```

This method works well when the input is uniformly distributed over 32-bit values because 76% of the inputs are caught by the first compare, 21% are caught by the second compare, 2% are caught by the third, and so on (chopping the remaining down by 90% with each comparision). As a result, less than 2.6 operations are needed on average.

On April 18, 2007, Emanuel Hoogeveen suggested a variation on this where the conditions used divisions, which were not as fast as simple comparisons.

Find integer log base 2 of a 32-bit IEEE float

```
const float v; // find int(log2(v)), where v > 0.0 && finite(v) && isnormal(v)
int c;         // 32-bit int c gets the result;

c = *(const int *) &v;  // OR, for portability:  memcpy(&c, &v, sizeof c);
c = (c >> 23) - 127;
```

The above is fast, but IEEE 754-compliant architectures utilize *subnormal* (also called *denormal*) floating point numbers. These have the exponent bits set to zero (signifying pow(2,-127)), and the mantissa is not normalized, so it contains leading zeros and thus the log2 must be computed from the mantissa. To accomodate for subnormal numbers, use the following:

```
const float v;              // find int(log2(v)), where v > 0.0 && finite(v)
int c;                      // 32-bit int c gets the result;
int x = *(const int *) &v;  // OR, for portability:  memcpy(&x, &v, sizeof x);

c = x >> 23;

if (c)
{
  c -= 127;
}
else
{ // subnormal, so recompute using mantissa: c = intlog2(x) - 149;
  register unsigned int t; // temporary
  // Note that LogTable256 was defined earlier
  if (t = x >> 16)
  {
    c = LogTable256[t] - 133;
  }
  else
  {
    c = (t = x >> 8) ? LogTable256[t] - 141 : LogTable256[x] - 149;
  }
}
```

On June 20, 2004, Sean A. Irvine suggested that I include code to handle subnormal numbers. On June 11, 2005, Falk Hüffner pointed out that ISO C99 6.5/7 specified undefined behavior for the common type punning idiom *(int *)&, though it has worked on 99.9% of C compilers. He proposed using memcpy for maximum portability or a union with a float and an int for better code generation than memcpy on some compilers.

Find integer log base 2 of the pow(2, r)-root of a 32-bit IEEE float (for unsigned integer r)

```
const int r;
const float v; // find int(log2(pow((double) v, 1. / pow(2, r)))),
               // where isnormal(v) and v > 0
int c;         // 32-bit int c gets the result;
```

```
c = *(const int *) &v;  // OR, for portability:  memcpy(&c, &v, sizeof c);
c = ((((c - 0x3f800000) >> r) + 0x3f800000) >> 23) - 127;
```

So, if r is 0, for example, we have c = int(log2((double) v)). If r is 1, then we have c = int(log2(sqrt((double) v))). If r is 2, then we have c = int(log2(pow((double) v, 1./4))).

On June 11, 2005, Falk Hüffner pointed out that ISO C99 6.5/7 left the type punning idiom *(int *)& undefined, and he suggested using memcpy.

Count the consecutive zero bits (trailing) on the right linearly

```
unsigned int v;  // input to count trailing zero bits
int c;  // output: c will count v's trailing zero bits,
        // so if v is 1101000 (base 2), then c will be 3
if (v)
{
  v = (v ^ (v - 1)) >> 1;  // Set v's trailing 0s to 1s and zero rest
  for (c = 0; v; c++)
  {
    v >>= 1;
  }
}
else
{
  c = CHAR_BIT * sizeof(v);
}
```

The average number of trailing zero bits in a (uniformly distributed) random binary number is one, so this O(trailing zeros) solution isn't that bad compared to the faster methods below.

Jim Cole suggested I add a linear-time method for counting the trailing zeros on August 15, 2007. On October 22, 2007, Jason Cunningham pointed out that I had neglected to paste the unsigned modifier for v.

Count the consecutive zero bits (trailing) on the right in parallel

```
unsigned int v;      // 32-bit word input to count zero bits on right
unsigned int c = 32; // c will be the number of zero bits on the right
v &= -signed(v);
if (v) c--;
if (v & 0x0000FFFF) c -= 16;
if (v & 0x00FF00FF) c -= 8;
if (v & 0x0F0F0F0F) c -= 4;
if (v & 0x33333333) c -= 2;
if (v & 0x55555555) c -= 1;
```

Here, we are basically doing the same operations as finding the log base 2 in parallel, but we first isolate the lowest 1 bit, and then proceed with c starting at the maximum and decreasing. The number of operations is at most 3 * lg(N) + 4, roughly, for N bit words.

Bill Burdick suggested an optimization, reducing the time from 4 * lg(N) on February 4, 2011.

Count the consecutive zero bits (trailing) on the right by binary search

```
unsigned int v;     // 32-bit word input to count zero bits on right
unsigned int c;     // c will be the number of zero bits on the right,
                    // so if v is 1101000 (base 2), then c will be 3
// NOTE: if 0 == v, then c = 31.
if (v & 0x1)
```

```
{
  // special case for odd v (assumed to happen half of the time)
  c = 0;
}
else
{
  c = 1;
  if ((v & 0xffff) == 0)
  {
    v >>= 16;
    c += 16;
  }
  if ((v & 0xff) == 0)
  {
    v >>= 8;
    c += 8;
  }
  if ((v & 0xf) == 0)
  {
    v >>= 4;
    c += 4;
  }
  if ((v & 0x3) == 0)
  {
    v >>= 2;
    c += 2;
  }
  c -= v & 0x1;
}
```

The code above is similar to the previous method, but it computes the number of trailing zeros by accumulating c in a manner akin to binary search. In the first step, it checks if the bottom 16 bits of v are zeros, and if so, shifts v right 16 bits and adds 16 to c, which reduces the number of bits in v to consider by half. Each of the subsequent conditional steps likewise halves the number of bits until there is only 1. This method is faster than the last one (by about 33%) because the bodies of the if statements are executed less often.

Matt Whitlock suggested this on January 25, 2006. Andrew Shapira shaved a couple operations off on Sept. 5, 2007 (by setting c=1 and unconditionally subtracting at the end).

Count the consecutive zero bits (trailing) on the right by casting to a float

```
unsigned int v;           // find the number of trailing zeros in v
int r;                    // the result goes here
float f = (float)(v & -v); // cast the least significant bit in v to a float
r = (*(uint32_t *)&f >> 23) - 0x7f;
```

Although this only takes about 6 operations, the time to convert an integer to a float can be high on some machines. The exponent of the 32-bit IEEE floating point representation is shifted down, and the bias is subtracted to give the position of the least significant 1 bit set in v. If v is zero, then the result is -127.

Count the consecutive zero bits (trailing) on the right with modulus division and lookup

```
unsigned int v;  // find the number of trailing zeros in v
int r;           // put the result in r
static const int Mod37BitPosition[] = // map a bit value mod 37 to its position
{
  32, 0, 1, 26, 2, 23, 27, 0, 3, 16, 24, 30, 28, 11, 0, 13, 4,
  7, 17, 0, 25, 22, 31, 15, 29, 10, 12, 6, 0, 21, 14, 9, 5,
  20, 8, 19, 18
};
r = Mod37BitPosition[(-v & v) % 37];
```

The code above finds the number of zeros that are trailing on the right, so binary 0100 would produce 2. It makes use of the fact that the first 32 bit position values are relatively prime with 37, so performing a modulus division with 37 gives a unique number from 0 to 36 for each. These numbers may then be mapped to the number of zeros using a small lookup table. It uses only 4 operations, however indexing into a table and performing modulus division may make it unsuitable for some situations. I came up with this independently and then searched for a subsequence of the table values, and found it was invented earlier by Reiser, according to Hacker's Delight.

Count the consecutive zero bits (trailing) on the right with multiply and lookup

```
unsigned int v;  // find the number of trailing zeros in 32-bit v
int r;           // result goes here
static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
  31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
};
r = MultiplyDeBruijnBitPosition[((uint32_t)((v & -v) * 0x077CB531U)) >> 27];
```

Converting bit vectors to indices of set bits is an example use for this. It requires one more operation than the earlier one involving modulus division, but the multiply may be faster. The expression (v & -v) extracts the least significant 1 bit from v. The constant 0x077CB531UL is a de Bruijn sequence, which produces a unique pattern of bits into the high 5 bits for each possible bit position that it is multiplied against. When there are no bits set, it returns 0. More information can be found by reading the paper Using de Bruijn Sequences to Index 1 in a Computer Word by Charles E. Leiserson, Harald Prokof, and Keith H. Randall.

On October 8, 2005 Andrew Shapira suggested I add this. Dustin Spicuzza asked me on April 14, 2009 to cast the result of the multiply to a 32-bit type so it would work when compiled with 64-bit ints.

Round up to the next highest power of 2 by float casting

```
unsigned int const v; // Round this 32-bit value to the next highest power of 2
unsigned int r;       // Put the result here. (So v=3 -> r=4; v=8 -> r=8)

if (v > 1)
{
  float f = (float)v;
  unsigned int const t = 1U << ((*(unsigned int *)&f >> 23) - 0x7f);
  r = t << (t < v);
}
else
{
  r = 1;
}
```

The code above uses 8 operations, but works on all v <= (1<<31).

Quick and dirty version, for domain of 1 < v < (1<<25):

```
float f = (float)(v - 1);
r = 1U << ((*(unsigned int*)(&f) >> 23) - 126);
```

Although the quick and dirty version only uses around 6 operations, it is roughly three times slower than the technique below (which involves 12 operations) when benchmarked on an Athlon™ XP 2100+ CPU. Some CPUs will fare better with it, though.

On September 27, 2005 Andi Smithers suggested I include a technique for casting to floats to find the lg of a number for rounding up to a power of 2. Similar to the quick and dirty version here, his version worked with values less than (1<<25), due to mantissa rounding, but it used one more operation.

Round up to the next highest power of 2

```
unsigned int v; // compute the next highest power of 2 of 32-bit v

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

In 12 operations, this code computes the next highest power of 2 for a 32-bit integer. The result may be expressed by the formula 1U << (lg(v - 1) + 1). Note that in the edge case where v is 0, it returns 0, which isn't a power of 2; you might append the expression v += (v == 0) to remedy this if it matters. It would be faster by 2 operations to use the formula and the log base 2 method that uses a lookup table, but in some situations, lookup tables are not suitable, so the above code may be best. (On a Athlon™ XP 2100+ I've found the above shift-left and then OR code is as fast as using a single BSR assembly language instruction, which scans in reverse to find the highest set bit.) It works by copying the highest set bit to all of the lower bits, and then adding one, which results in carries that set all of the lower bits to 0 and one bit beyond the highest set bit to 1. If the original number was a power of 2, then the decrement will reduce it to one less, so that we round up to the same original value.

You might alternatively compute the next higher power of 2 in only 8 or 9 operations using a lookup table for floor(lg(v)) and then evaluating 1<<(1+floor(lg(v))); Atul Divekar suggested I mention this on September 5, 2010.

Devised by Sean Anderson, Sepember 14, 2001. Pete Hart pointed me to a couple newsgroup posts by him and William Lewis in February of 1997, where they arrive at the same algorithm.

Interleave bits the obvious way

```
unsigned short x;   // Interleave bits of x and y, so that all of the
unsigned short y;   // bits of x are in the even positions and y in the odd;
unsigned int z = 0; // z gets the resulting Morton Number.

for (int i = 0; i < sizeof(x) * CHAR_BIT; i++) // unroll for more speed...
{
  z |= (x & 1U << i) << i | (y & 1U << i) << (i + 1);
}
```

Interleaved bits (aka Morton numbers) are useful for linearizing 2D integer coordinates, so x and y are combined into a single number that can be compared easily and has the property that a number is usually close to another if their x and y values are close.

Interleave bits by table lookup

```
static const unsigned short MortonTable256[256] =
{
  0x0000, 0x0001, 0x0004, 0x0005, 0x0010, 0x0011, 0x0014, 0x0015,
  0x0040, 0x0041, 0x0044, 0x0045, 0x0050, 0x0051, 0x0054, 0x0055,
  0x0100, 0x0101, 0x0104, 0x0105, 0x0110, 0x0111, 0x0114, 0x0115,
  0x0140, 0x0141, 0x0144, 0x0145, 0x0150, 0x0151, 0x0154, 0x0155,
  0x0400, 0x0401, 0x0404, 0x0405, 0x0410, 0x0411, 0x0414, 0x0415,
  0x0440, 0x0441, 0x0444, 0x0445, 0x0450, 0x0451, 0x0454, 0x0455,
  0x0500, 0x0501, 0x0504, 0x0505, 0x0510, 0x0511, 0x0514, 0x0515,
  0x0540, 0x0541, 0x0544, 0x0545, 0x0550, 0x0551, 0x0554, 0x0555,
  0x1000, 0x1001, 0x1004, 0x1005, 0x1010, 0x1011, 0x1014, 0x1015,
  0x1040, 0x1041, 0x1044, 0x1045, 0x1050, 0x1051, 0x1054, 0x1055,
  0x1100, 0x1101, 0x1104, 0x1105, 0x1110, 0x1111, 0x1114, 0x1115,
```

```
  0x1140, 0x1141, 0x1144, 0x1145, 0x1150, 0x1151, 0x1154, 0x1155,
  0x1400, 0x1401, 0x1404, 0x1405, 0x1410, 0x1411, 0x1414, 0x1415,
  0x1440, 0x1441, 0x1444, 0x1445, 0x1450, 0x1451, 0x1454, 0x1455,
  0x1500, 0x1501, 0x1504, 0x1505, 0x1510, 0x1511, 0x1514, 0x1515,
  0x1540, 0x1541, 0x1544, 0x1545, 0x1550, 0x1551, 0x1554, 0x1555,
  0x4000, 0x4001, 0x4004, 0x4005, 0x4010, 0x4011, 0x4014, 0x4015,
  0x4040, 0x4041, 0x4044, 0x4045, 0x4050, 0x4051, 0x4054, 0x4055,
  0x4100, 0x4101, 0x4104, 0x4105, 0x4110, 0x4111, 0x4114, 0x4115,
  0x4140, 0x4141, 0x4144, 0x4145, 0x4150, 0x4151, 0x4154, 0x4155,
  0x4400, 0x4401, 0x4404, 0x4405, 0x4410, 0x4411, 0x4414, 0x4415,
  0x4440, 0x4441, 0x4444, 0x4445, 0x4450, 0x4451, 0x4454, 0x4455,
  0x4500, 0x4501, 0x4504, 0x4505, 0x4510, 0x4511, 0x4514, 0x4515,
  0x4540, 0x4541, 0x4544, 0x4545, 0x4550, 0x4551, 0x4554, 0x4555,
  0x5000, 0x5001, 0x5004, 0x5005, 0x5010, 0x5011, 0x5014, 0x5015,
  0x5040, 0x5041, 0x5044, 0x5045, 0x5050, 0x5051, 0x5054, 0x5055,
  0x5100, 0x5101, 0x5104, 0x5105, 0x5110, 0x5111, 0x5114, 0x5115,
  0x5140, 0x5141, 0x5144, 0x5145, 0x5150, 0x5151, 0x5154, 0x5155,
  0x5400, 0x5401, 0x5404, 0x5405, 0x5410, 0x5411, 0x5414, 0x5415,
  0x5440, 0x5441, 0x5444, 0x5445, 0x5450, 0x5451, 0x5454, 0x5455,
  0x5500, 0x5501, 0x5504, 0x5505, 0x5510, 0x5511, 0x5514, 0x5515,
  0x5540, 0x5541, 0x5544, 0x5545, 0x5550, 0x5551, 0x5554, 0x5555
};

unsigned short x; // Interleave bits of x and y, so that all of the
unsigned short y; // bits of x are in the even positions and y in the odd;
unsigned int z;   // z gets the resulting 32-bit Morton Number.

z = MortonTable256[y >> 8]   << 17 |
    MortonTable256[x >> 8]   << 16 |
    MortonTable256[y & 0xFF] <<  1 |
    MortonTable256[x & 0xFF];
```

For more speed, use an additional table with values that are MortonTable256 pre-shifted one bit to the left. This second table could then be used for the y lookups, thus reducing the operations by two, but almost doubling the memory required. Extending this same idea, four tables could be used, with two of them pre-shifted by 16 to the left of the previous two, so that we would only need 11 operations total.

Interleave bits with 64-bit multiply

In 11 operations, this version interleaves bits of two bytes (rather than shorts, as in the other versions), but many of the operations are 64-bit multiplies so it isn't appropriate for all machines. The input parameters, x and y, should be less than 256.

```
unsigned char x;  // Interleave bits of (8-bit) x and y, so that all of the
unsigned char y;  // bits of x are in the even positions and y in the odd;
unsigned short z; // z gets the resulting 16-bit Morton Number.

z = ((x * 0x0101010101010101ULL & 0x8040201008040201ULL) *
     0x0102040810204081ULL >> 49) & 0x5555 |
    ((y * 0x0101010101010101ULL & 0x8040201008040201ULL) *
     0x0102040810204081ULL >> 48) & 0xAAAA;
```

Holger Bettag was inspired to suggest this technique on October 10, 2004 after reading the multiply-based bit reversals here.

Interleave bits by Binary Magic Numbers

```
static const unsigned int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF};
static const unsigned int S[] = {1, 2, 4, 8};

unsigned int x; // Interleave lower 16 bits of x and y, so the bits of x
```

```
unsigned int y; // are in the even positions and bits from y in the odd;
unsigned int z; // z gets the resulting 32-bit Morton Number.
                // x and y must initially be less than 65536.

x = (x | (x << S[3])) & B[3];
x = (x | (x << S[2])) & B[2];
x = (x | (x << S[1])) & B[1];
x = (x | (x << S[0])) & B[0];

y = (y | (y << S[3])) & B[3];
y = (y | (y << S[2])) & B[2];
y = (y | (y << S[1])) & B[1];
y = (y | (y << S[0])) & B[0];

z = x | (y << 1);
```

Determine if a word has a zero byte

```
// Fewer operations:
unsigned int v; // 32-bit word to check if any 8-bit byte in it is 0
bool hasZeroByte = ~((((v & 0x7F7F7F7F) + 0x7F7F7F7F) | v) | 0x7F7F7F7F);
```

The code above may be useful when doing a fast string copy in which a word is copied at a time; it uses 5 operations. On the other hand, testing for a null byte in the obvious ways (which follow) have at least 7 operations (when counted in the most sparing way), and at most 12.

```
// More operations:
bool hasNoZeroByte = ((v & 0xff) && (v & 0xff00) && (v & 0xff0000) && (v & 0xff000000))
// OR:
unsigned char * p = (unsigned char *) &v;
bool hasNoZeroByte = *p && *(p + 1) && *(p + 2) && *(p + 3);
```

The code at the beginning of this section (labeled "Fewer operations") works by first zeroing the high bits of the 4 bytes in the word. Subsequently, it adds a number that will result in an overflow to the high bit of a byte if any of the low bits were initialy set. Next the high bits of the original word are ORed with these values; thus, the high bit of a byte is set iff any bit in the byte was set. Finally, we determine if any of these high bits are zero by ORing with ones everywhere except the high bits and inverting the result. Extending to 64 bits is trivial; simply increase the constants to be 0x7F7F7F7F7F7F7F7F.

For an additional improvement, a fast pretest that requires only 4 operations may be performed to determine if the word *may* have a zero byte. The test also returns true if the high byte is 0x80, so there are occasional false positives, but the slower and more reliable version above may then be used on candidates for an overall increase in speed with correct output.

```
bool hasZeroByte = ((v + 0x7efefeff) ^ ~v) & 0x81010100;
if (hasZeroByte) // or may just have 0x80 in the high byte
{
  hasZeroByte = ~((((v & 0x7F7F7F7F) + 0x7F7F7F7F) | v) | 0x7F7F7F7F);
}
```

There is yet a faster method — use hasless(v, 1), which is defined below; it works in 4 operations and requires no subsquent verification. It simplifies to

```
#define haszero(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
```

The subexpression (v - 0x01010101UL), evaluates to a high bit set in any byte whenever the corresponding byte in v is zero or greater than 0x80. The sub-expression ~v & 0x80808080UL evaluates to high bits set in bytes where the byte of v doesn't have its high bit set (so the byte was less than 0x80). Finally, by ANDing these two sub-expressions the result is the high bits set where the bytes in v were zero, since the high bits set due to a value greater than 0x80 in the first sub-expression are masked off by the second.

Paul Messmer suggested the fast pretest improvement on October 2, 2004. Juha Järvi later suggested `hasless(v, 1)` on April 6, 2005, which he found on Paul Hsieh's Assembly Lab; previously it was written in a newsgroup post on April 27, 1987 by Alan Mycroft.

Determine if a word has a byte equal to n

We may want to know if any byte in a word has a specific value. To do so, we can XOR the value to test with a word that has been filled with the byte values in which we're interested. Because XORing a value with itself results in a zero byte and nonzero otherwise, we can pass the result to `haszero`.

```
#define hasvalue(x,n) \
(haszero((x) ^ (~0UL/255 * (n))))
```

Stephen M Bennet suggested this on December 13, 2009 after reading the entry for `haszero`.

Determine if a word has a byte less than n

Test if a word x contains an unsigned byte with value < n. Specifically for n=1, it can be used to find a 0-byte by examining one long at a time, or any byte by XORing x with a mask first. Uses 4 arithmetic/logical operations when n is constant.

Requirements: x>=0; 0<=n<=128

```
#define hasless(x,n) (((x)-~0UL/255*(n))&~(x)&~0UL/255*128)
```

To count the number of bytes in x that are less than n in 7 operations, use

```
#define countless(x,n) \
(((~0UL/255*(127+(n))-((x)&~0UL/255*127))&~(x)&~0UL/255*128)/128%255)
```

Juha Järvi sent this clever technique to me on April 6, 2005. The `countless` macro was added by Sean Anderson on April 10, 2005, inspired by Juha's `countmore`, below.

Determine if a word has a byte greater than n

Test if a word x contains an unsigned byte with value > n. Uses 3 arithmetic/logical operations when n is constant.

Requirements: x>=0; 0<=n<=127

```
#define hasmore(x,n) (((x)+~0UL/255*(127-(n))|(x))&~0UL/255*128)
```

To count the number of bytes in x that are more than n in 6 operations, use:

```
#define countmore(x,n) \
(((((x)&~0UL/255*127)+~0UL/255*(127-(n))|(x))&~0UL/255*128)/128%255)
```

The macro `hasmore` was suggested by Juha Järvi on April 6, 2005, and he added `countmore` on April 8, 2005.

Determine if a word has a byte between m and n

When m < n, this technique tests if a word x contains an unsigned byte value, such that m < value < n. It uses 7 arithmetic/logical operations when n and m are constant.

Note: Bytes that equal n can be reported by `likelyhasbetween` as false positives, so this should be checked by character if a certain result is needed.

Requirements: x>=0; 0<=m<=127; 0<=n<=128

```
#define likelyhasbetween(x,m,n) \
((((x)-~0UL/255*(n))&~(x)&((x)&~0UL/255*127)+~0UL/255*(127-(m)))&~0UL/255*128)
```

This technique would be suitable for a fast pretest. A variation that takes one more operation (8 total for constant m and n) but provides the exact answer is:

```
#define hasbetween(x,m,n) \
((~0UL/255*(127+(n))-((x)&~0UL/255*127)&~(x)&((x)&~0UL/255*127)+~0UL/255*(127-(m)))&~0UL/255*128)
```

To count the number of bytes in x that are between m and n (exclusive) in 10 operations, use:

```
#define countbetween(x,m,n) (hasbetween(x,m,n)/128%255)
```

Juha Järvi suggested `likelyhasbetween` on April 6, 2005. From there, Sean Anderson created `hasbetween` and `countbetween` on April 10, 2005.

Compute the lexicographically next bit permutation

Suppose we have a pattern of N bits set to 1 in an integer and we want the next permutation of N 1 bits in a lexicographical sense. For example, if N is 3 and the bit pattern is 00010011, the next patterns would be 00010101, 00010110, 00011001,00011010, 00011100, 00100011, and so forth. The following is a fast way to compute the next permutation.

```
unsigned int v; // current permutation of bits
unsigned int w; // next permutation of bits

unsigned int t = v | (v - 1); // t gets v's least significant 0 bits set to 1
// Next set to 1 the most significant bit to change,
// set to 0 the least significant ones, and add the necessary 1 bits.
w = (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(v) + 1));
```

The __builtin_ctz(v) GNU C compiler intrinsic for x86 CPUs returns the number of trailing zeros. If you are using Microsoft compilers for x86, the intrinsic is _BitScanForward. These both emit a bsf instruction, but equivalents may be available for other architectures. If not, then consider using one of the methods for counting the consecutive zero bits mentioned earlier.

Here is another version that tends to be slower because of its division operator, but it does not require counting the trailing zeros.

```
unsigned int t = (v | (v - 1)) + 1;
w = t | ((((t & -t) / (v & -v)) >> 1) - 1);
```

Thanks to Dario Sneidermanis of Argentina, who provided this on November 28, 2009.

A Belorussian translation (provided by Webhostingrating) is available.