

Count the number of set bits in a 32-bit integer

Asked 16 years, 3 months ago Modified 6 months ago

Viewed 663k times



8 bits representing the number 7 look like this:

1024

```
00000111
```



Three bits are set.



What are the algorithms to determine the number of set bits in a 32-bit integer?



algorithm

binary

bit-manipulation

hammingweight

iec10967

Share

edited Aug 26, 2022 at 20:11

Improve this question

Follow

community wiki

15 revs, 9 users 40%

Matt Howells

-
- 125 This is the Hamming weight BTW. – [Purfideas](#) Sep 20, 2008 at 19:17
-
- 14 What's a real-world application for this? (This isn't to be taken as a criticism--I'm just curious.) – [jonmorgan](#) Dec 10, 2010 at 20:59
-
- 11 Calculation of parity bit (look it up), which was used as simple error detection in communication. – [Dialecticus](#) Dec 11, 2010 at 0:28
-
- 9 @Dialecticus, calculating a parity bit is [cheaper](#) than calculating the Hamming weight – [finnw](#) May 12, 2011 at 12:14
-
- 18 @spookyjon Let's say you have a graph represented as an adjacency matrix, which is essentially a bit set. If you want to calculate the number of edges of a vertex, it boils down to calculating the Hamming weight of one row in the bit set. – [fuz](#) Oct 10, 2011 at 16:02
-

66 Answers

Sorted by:

Highest score (default)



1

2

3

Next



This is known as the '[Hamming Weight](#)', 'popcount' or 'sideways addition'.

998



Some CPUs have a single built-in instruction to do it and others have parallel instructions which act on bit vectors. Instructions like x86's [popcnt](#) (on CPUs where it's supported) will almost certainly be fastest for a single integer. Some other architectures may have a slow instruction implemented with a microcoded loop that tests





a bit per cycle (*citation needed* - hardware popcount is normally fast if it exists at all.).

The 'best' algorithm really depends on which CPU you are on and what your usage pattern is.

Your compiler may know how to do something that's good for the specific CPU you're compiling for, e.g. [C++20](#) `std::popcount(.)`, or C++ `std::bitset<32>::count(.)`, as a portable way to access builtin / intrinsic functions (see [another answer](#) on this question). But your compiler's choice of fallback for target CPUs that don't have hardware popcnt might not be optimal for your use-case. Or your language (e.g. C) might not expose any portable function that could use a CPU-specific popcount when there is one.

Portable algorithms that don't need (or benefit from) any HW support

A pre-populated table lookup method can be very fast if your CPU has a large cache and you are doing lots of these operations in a tight loop. However it can suffer because of the expense of a 'cache miss', where the CPU has to fetch some of the table from main memory. (Look up each byte separately to keep the table small.) If you want popcount for a contiguous range of numbers, only the low byte is changing for groups of 256 numbers, [making this very good](#).

If you know that your bytes will be mostly 0's or mostly 1's then there are efficient algorithms for these scenarios, e.g. clearing the lowest set with a bithack in a loop until it becomes zero.

I believe a very good general purpose algorithm is the following, known as 'parallel' or 'variable-precision SWAR algorithm'. I have expressed this in a C-like pseudo language, you may need to adjust it to work for a particular language (e.g. using `uint32_t` for C++ and `>>>` in Java):

GCC10 and clang 10.0 can recognize this pattern / idiom and compile it to a hardware `popcnt` or equivalent instruction when available, giving you the best of both worlds. ([Godbolt](#))

```
int numberOfSetBits(uint32_t i)
{
    // Java: use int, and use >>> instead of >>. Or u
    // C or C++: use uint32_t
    i = i - ((i >> 1) & 0x55555555);           // add pa
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
    i = (i + (i >> 4)) & 0x0F0F0F0F;           // groups
    i *= 0x01010101;                           // horizo
    return i >> 24;                             // return just tha
    truncating to 32-bit even when int is wider than uint3
}
```

For JavaScript: coerce to integer with `|0` for

performance: change the first line to `i = (i|0) - ((i >>`
`1) & 0x55555555);`

This has the best worst-case behaviour of any of the algorithms discussed, so will efficiently deal with any usage pattern or values you throw at it. (Its performance is not data-dependent on normal CPUs where all integer operations including multiply are constant-time. It doesn't get any faster with "simple" inputs, but it's still pretty decent.)

References:

- [Bit Twiddling Hacks](#)
- [Introduction to Low Level Bit Hacks](#) for bithack basics, like how subtracting 1 flips contiguous zeros.
- [Hamming weight \(Wikipedia\)](#).
- [Fast Bit Counting](#)
- [The Aggregate Magic Algorithms](#)

How this SWAR bithack works:

```
i = i - ((i >> 1) & 0x55555555);
```

The first step is an optimized version of masking to isolate the odd / even bits, shifting to line them up, and adding. This effectively does 16 separate additions in 2-bit accumulators ([SWAR = SIMD Within A Register](#)). Like

```
(i & 0x55555555) + ((i>>1) & 0x55555555) .
```

The next step takes the odd/even eight of those 16x 2-bit accumulators and adds again, producing 8x 4-bit sums.

The `i - ...` optimization isn't possible this time so it does just mask before / after shifting. Using the same `0x33...` constant both times instead of `0xccc...` before shifting is a good thing when compiling for ISAs that need to construct 32-bit constants in registers separately.

The final shift-and-add step of `(i + (i >> 4)) & 0x0F0F0F0F` widens to 4x 8-bit accumulators. It masks *after* adding instead of before, because the maximum value in any 4-bit accumulator is `4`, if all 4 bits of the corresponding input bits were set. $4+4 = 8$ which still fits in 4 bits, so carry between nibble elements is impossible in `i + (i >> 4)`.

So far this is just fairly normal SIMD using SWAR techniques with a few clever optimizations. Continuing on with the same pattern for 2 more steps can widen to 2x 16-bit then 1x 32-bit counts. But there is a more efficient way on machines with fast hardware multiply:

Once we have few enough "elements", **a multiply with a magic constant can sum all the elements into the top element**. In this case byte elements. Multiply is done by left-shifting and adding, so **a multiply of** `x * 0x01010101` **results in** `x + (x<<8) + (x<<16) + (x<<24)`. Our 8-bit elements are wide enough (and holding small enough counts) that this doesn't produce carry *into* that top 8 bits.

A 64-bit version of this can do 8x 8-bit elements in a 64-bit integer with a `0x0101010101010101` multiplier, and extract the high byte with `>>56`. So it doesn't take any extra steps, just wider constants. This is what GCC uses

for `__builtin_popcountll` on x86 systems when the hardware `popcnt` instruction isn't enabled. If you can use builtins or intrinsics for this, do so to give the compiler a chance to do target-specific optimizations.

With full SIMD for wider vectors (e.g. counting a whole array)

This bitwise-SWAR algorithm could parallelize to be done in multiple vector elements at once, instead of in a single integer register, for a speedup on CPUs with SIMD but no usable popcount instruction. (e.g. x86-64 code that has to run on any CPU, not just Nehalem or later.)

However, the best way to use vector instructions for popcount is usually by using a variable-shuffle to do a table-lookup for 4 bits at a time of each byte in parallel. (The 4 bits index a 16 entry table held in a vector register).

On Intel CPUs, the hardware 64bit `popcnt` instruction can outperform an [SSSE3](#) `PSHUFB` [bit-parallel implementation](#) by about a factor of 2, but only [if your compiler gets it just right](#). Otherwise SSE can come out significantly ahead. Newer compiler versions are aware of the [popcnt false dependency problem on Intel](#).

- [State-of-the-art x86 SIMD popcount](#) for SSSE3, AVX2, AVX512BW, AVX512VBMI, or AVX512 VPOPCNT. Using Harley-Seal across vectors to defer popcount within an element. (Also ARM NEON)

- [Counting 1 bits \(population count\) on large data using AVX-512 or AVX-2](#)
- related: [mklargvist's Positional popcount](#) - separate counts for each bit-position of multiple 8, 16, 32, or 64-bit integers. (Again, x86 SIMD including AVX-512 which is really good at this, with `vpternlogd` making Harley-Seal very good.)

Share Improve this answer

edited Nov 19, 2023 at 6:41

Follow

community wiki

26 revs, 10 users 56%

Peter Cordes

104 ha! love the NumberOfSetBits() function, but good luck getting that through a code review. :-) – [Jason S](#) Nov 22, 2009 at 6:51

41 Maybe it should use `unsigned int` , to easily show that it is free of any sign bit complications. Also would `uint32_t` be safer, as in, you get what you expect on all platforms? – [Craig McQueen](#) Dec 15, 2009 at 2:18

39 @nonnb: Actually, as written, the code is buggy and needs maintenance. `>>` is implementation-defined for negative values. The argument needs to be changed (or cast) to `unsigned` , and since the code is 32-bit-specific, it should probably be using `uint32_t` . – [R.. GitHub STOP HELPING ICE](#) May 14, 2011 at 21:55

8 It's not really magic. It's adding sets of bits but doing so with some clever optimizations. The wikipedia link given in the answer does a good job of explaining what's going on but I'll go line by line. 1) Count up the number of bits in every

pair of bits, putting that count in that pair of bits (you'll have 00, 01, or 10); the "clever" bit here is the subtract that avoids one mask. 2) Add pairs of those sums of bitpairs into their corresponding nibbles; nothing clever here but each nibble will now have a value 0-4. (cont'd) – [dash-tom-bang](#)
Dec 5, 2012 at 0:42

- 8 Another note, this extends to 64 and 128 bit registers by simply extending the constants appropriately. Interestingly (to me), those constants are also $\sim 0 / 3, 5, 17$, and 255; the former three being 2^{n+1} . This all makes more sense the more you stare at it and think about it in the shower. :)
– [dash-tom-bang](#) Dec 5, 2012 at 0:48
-



256

Some languages portably expose the operation in a way that *can* use efficient hardware support if available, otherwise some library fallback that's hopefully decent.



For example (from [a table by language](#)):

- C++ has `std::bitset<>::count()`, or [C++20](#) `std::popcount(T x)`.
- Java has `java.lang.Integer.bitCount()` (also for Long or BigInteger)
- C# has `System.Numerics.BitOperations.PopCount()`
- Python has `int.bit_count()` (since 3.10)

Not all compilers / libraries actually manage to use HW support when it's available, though. (Notably MSVC, even with options that make `std::popcount` inline as `x86 popcnt`, its `std::bitset::count` still always uses a lookup table. This will hopefully change in future versions.)

Also consider the built-in functions of your compiler when the portable language doesn't have this basic bit operation. In GNU C for example:

```
int __builtin_popcount (unsigned int x);  
int __builtin_popcountll (unsigned long long x);
```

In the worst case (no single-instruction HW support) the compiler will generate a call to a function (which in current GCC uses a shift/and bit-hack [like this answer](#), at least for x86). In the best case the compiler will emit a cpu instruction to do the job. (Just like a `*` or `/` operator - GCC will use a hardware multiply or divide instruction if available, otherwise will call a libgcc helper function.) Or even better, if the operand is a compile-time constant after inlining, it can do constant-propagation to get a compile-time-constant popcount result.

The GCC builtins even work across multiple platforms. Popcount has almost become mainstream in the x86 architecture, so it makes sense to start using the builtin now so you can recompile to let it inline a hardware instruction when you compile with `-mpopcnt` or something that includes that ([example](#)). Other architectures have had popcount for years, but in the x86 world there are still some ancient Core 2 and similar vintage AMD CPUs in use.

On x86, you can tell the compiler that it can assume support for `popcnt` instruction with `-mpopcnt` (also

implied by `-msse4.2`). See [GCC x86 options](#). `-march=nehalem -mtune=skylake` (or `-march=` whatever CPU you want your code to assume and to tune for) could be a good choice. Running the resulting binary on an older CPU will result in an illegal-instruction fault.

To make binaries optimized for the machine you build them on, **use** `-march=native` (with gcc, clang, or ICC).

[MSVC provides an intrinsic for the x86 `popcnt` instruction](#), but unlike gcc it's really an intrinsic for the hardware instruction and requires hardware support.

Using `std::bitset<>::count()` instead of a built-in

In theory, any compiler that knows how to popcount efficiently for the target CPU should expose that functionality through ISO C++ `std::bitset<>`. In practice, you might be better off with the bit-hack AND/shift/ADD in some cases for some target CPUs.

For target architectures where hardware popcount is an optional extension (like x86), not all compilers have a `std::bitset` that takes advantage of it when available. For example, MSVC has no way to enable `popcnt` support at compile time, and it's `std::bitset<>::count` always uses [a table lookup](#), even with `/Ox /arch:AVX` (which implies SSE4.2, which in turn implies the `popcnt` feature.) (Update: see below; that *does* get MSVC's C++20 `std::popcount` to use x86 `popcnt`, but still not its

bitset<>::count. MSVC could fix that by updating their standard library headers to use std::popcount when available.)

But at least you get something portable that works everywhere, and with gcc/clang with the right target options, you get hardware popcount for architectures that support it.

```
#include <bitset>
#include <limits>
#include <type_traits>

template<typename T>
//static inline // static if you want to compile with
// compilation unit but not others
typename std::enable_if<std::is_integral<T>::value, u
popcount(T x)
{
    static_assert(std::numeric_limits<T>::radix == 2,

    // sizeof(x)*CHAR_BIT
    constexpr int bitwidth = std::numeric_limits<T>::d
std::numeric_limits<T>::is_signed;
    // std::bitset constructor was only unsigned long
    porting to C++03
    static_assert(bitwidth <= std::numeric_limits<unsi
"arg too wide for std::bitset() constructor");

    typedef typename std::make_unsigned<T>::type UT;
    needed, bitset width chops after sign-extension

    std::bitset<bitwidth> bs( static_cast<UT>(x) );
    return bs.count();
}
```

See [asm from gcc, clang, icc, and MSVC](#) on the Godbolt compiler explorer.

x86-64 `gcc -O3 -std=gnu++11 -mpopcnt` emits this:

```
unsigned test_short(short a) { return popcount(a); }
    movzx    eax, di          # note zero-extension, not si
    popcnt   rax, rax
    ret

unsigned test_int(int a) { return popcount(a); }
    mov      eax, edi
    popcnt   rax, rax         # unnecessary 64-bit opera
    ret

unsigned test_u64(unsigned long long a) { return popco
    xor      eax, eax        # gcc avoids false dependenci
    popcnt   rax, rdi
    ret
```

PowerPC64 `gcc -O3 -std=gnu++11` emits (for the `int` arg version):

```
    rldicl  3,3,0,32         # zero-extend from 32 to 64-bi
    popcntd 3,3              # popcount
    blr
```

This source isn't x86-specific or GNU-specific at all, but only compiles well with gcc/clang/icc, at least when targeting x86 (including x86-64).

Also note that gcc's fallback for architectures without single-instruction popcount is a byte-at-a-time table lookup. This isn't wonderful [for ARM, for example](#).

C++20 has `std::popcount(T)`.

Current libstdc++ headers unfortunately define it with a special case `if(x==0) return 0;` at the start, which clang doesn't optimize away when compiling for x86:

```
#include <bit>
int bar(unsigned x) {
    return std::popcount(x);
}
```

clang 11.0.1 `-O3 -std=gnu++20 -march=nehalem` [Live demo](#)

```
# clang 11
    bar(unsigned int):
        popcnt    eax, edi
        cmov     eax, edi           # redundant: if popcn
original 0 instead of the popcnt-generated 0...
        ret
```

But GCC compiles nicely:

```
# gcc 10
        xor      eax, eax           # break false depende
before Ice Lake.
        popcnt   eax, edi
        ret
```

Even MSVC does well with it, as long as you use `-arch:AVX` or later (and enable C++20 with `-std:c++latest`). [Live demo](#)

```
int bar(unsigned int) PROC
    popcnt    eax, ecx
```

```
ret 0
int bar(unsigned int) ENDP
```

Share Improve this answer

edited Nov 19, 2023 at 6:38

Follow

community wiki

7 revs, 4 users 84%

Peter Cordes

-
- 5 I agree that this is good practice in general, but on XCode/OSX/Intel I found it to generate slower code than most of the suggestions posted here. See my answer for details. – Mike F Sep 25, 2008 at 3:29
-
- 5 The Intel i5/i7 has the SSE4 instruction POPCNT which does it, using general purpose registers. GCC on my system does not emit that instruction using this intrinsic, i guess because of no -march=nehalem option yet. – matja Nov 24, 2009 at 10:31
-
- 3 @matja, my GCC 4.4.1 emits the popcnt instruction if I compile with -msse4.2 – Nils Pipenbrinck Nov 24, 2009 at 13:29
-
- 76 use c++'s `std::bitset::count` . after inlining this compiles to a single `__builtin_popcount` call. – deft_code Sep 4, 2010 at 18:18
-
- 1 @nlucaroni Well, yes. Times are changing. I've wrote this answer in 2008. Nowadays we have native popcount and the intrinsic will compile down to a single assembler statement if the platform allows that. – Nils Pipenbrinck Jul 24, 2013 at 19:45
-

**213**

In my opinion, the "best" solution is the one that can be read by another programmer (or the original programmer two years later) without copious comments. You may well want the fastest or cleverest solution which some have already provided but I prefer readability over cleverness any time.

```
unsigned int bitCount (unsigned int value) {
    unsigned int count = 0;
    while (value > 0) {           // until all
bits are zero
        if ((value & 1) == 1)     // check lower
bit
            count++;
        value >>= 1;             // shift bits,
removing lower bit
    }
    return count;
}
```

If you want more speed (and assuming you document it well to help out your successors), you could use a table lookup:

```
// Lookup table for fast calculation of bits set
in 8-bit unsigned char.

static unsigned char oneBitsInUChar[] = {
//  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
(<- n)
//
=====
    0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3,
4, // 0n
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4,
5, // 1n
    : : :
```



```

        4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7,
8, // Fn
};

// Function for fast calculation of bits set in
16-bit unsigned short.

unsigned char oneBitsInUShort (unsigned short x) {
    return oneBitsInUChar [x >> 8]
        + oneBitsInUChar [x & 0xff];
}

// Function for fast calculation of bits set in
32-bit unsigned int.

unsigned char oneBitsInUInt (unsigned int x) {
    return oneBitsInUShort (x >> 16)
        + oneBitsInUShort (x & 0xffff);
}

```

These rely on specific data type sizes so they're not that portable. But, since many performance optimisations aren't portable anyway, that may not be an issue. If you want portability, I'd stick to the readable solution.

Share Improve this answer

edited Sep 10, 2022 at 12:26


Follow

community wiki

6 revs, 3 users 89%

paxdiablo

23 Instead of dividing by 2 and commenting it as "shift bits...", you should just use the shift operator (>>) and leave out the comment. – [indiv](#) Sep 25, 2008 at 3:42

- 12 wouldn't it make more sense to replace `if ((value & 1) == 1) { count++; }` with `count += value & 1 ?`
– [Ponkadoodle](#) Apr 25, 2010 at 19:04
-
- 29 No, the best solution isn't the one most readable in this case. Here the best algorithm is the fastest one. – [NikiC](#) Sep 23, 2010 at 7:55
-
- 26 That's entirely your opinion, @nikic, although you're free to downvote me, obviously. There was no mention in the question as to how to quantify "best", the words "performance" or "fast" can be seen nowhere. That's why I opted for readable. – [paxdiablo](#) Sep 23, 2010 at 8:57
-
- 4 I made it quite clear that the answer was heavily based on my opinions and preferences, and I saw little need to rehash the performance-based answers already in play. Earlier comments have already covered this ground and, apparently, 172 people agreed though, admittedly, that's four fifths of bugger-all of the SO community so may not carry *that* much weight :-) Bottom line, I have no issue with the way you vote, I just wanted to make sure you et al at least understood *why* I gave the answer I gave. You may have the last word if you wish, I think I've explained as best I can. – [paxdiablo](#) Mar 2, 2018 at 1:45 
-



107



[From Hacker's Delight, p. 66, Figure 5-2](#)

```
int pop(unsigned x)
{
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) &
0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    x = x + (x >> 16);
}
```

```
    return x & 0x00000003F;
}
```

Executes in ~20-ish instructions (arch dependent), no branching.

[Hacker's Delight](#) is delightful! Highly recommended.


Share Improve this answer edited Aug 9, 2012 at 21:38


Follow

community wiki
2 revs, 2 users 94%
Kevin Little

8 The Java method `Integer.bitCount(int)` uses this same exact implementation. – [Marco Bolis](#) Jan 5, 2015 at 16:33

Having a little trouble following this - how would it change if we only cared about 16-bit values, instead of 32-bit?
– [Jeremy Blum](#) Feb 24, 2015 at 7:23

1 Maybe hackers delight is delightful, but I would give a good kicking to anybody calling this `pop` instead of `population_count` (or `pop_cnt` if you must have an abbreviation). @MarcoBolis I presume that will be true of all versions of Java, but officially that would be implementation dependent :) – [Maarten Bodewes](#) Mar 18, 2015 at 18:51 

And, this requires no multiplications, like the code in the accepted answer. – [Alex](#) Jul 7, 2017 at 13:23 

Note that in generalizing to 64-bit there is a problem. The result cannot be 64, because of the mask.

– [Albert van der Horst](#) Mar 5, 2019 at 10:56



87

I think the fastest way—without using lookup tables and *popcount*—is the following. It counts the set bits with just 12 operations.



```
int popcount(int v) {
    v = v - ((v >> 1) & 0x55555555);
    // put count of each 2 bits into those 2 bits
    v = (v & 0x33333333) + ((v >> 2) &
0x33333333); // put count of each 4 bits into
those 4 bits
    return ((v + (v >> 4) & 0xF0F0F0F) *
0x1010101) >> 24;
}
```

It works because you can count the total number of set bits by dividing in two halves, counting the number of set bits in both halves and then adding them up. Also known as **Divide and Conquer** paradigm. Let's get into detail..

```
v = v - ((v >> 1) & 0x55555555);
```

The number of bits in two bits can be **0b00**, **0b01** or **0b10**. Let's try to work this out on 2 bits..

v	(v >> 1) & 0b0101	v - x

0b00	0b00	0b00
0b01	0b00	0b01

0b10	0b01	0b01
0b11	0b01	0b10

This is what was required: the last column shows the count of set bits in every two bit pair. If the two bit number is `>= 2` (`0b10`) then `and` produces `0b01`, else it produces `0b00`.

```
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
```

This statement should be easy to understand. After the first operation we have the count of set bits in every two bits, now we sum up that count in every 4 bits.

```
v & 0b00110011          //masks out even two bits
(v >> 2) & 0b00110011    // masks out odd two bits
```

We then sum up the above result, giving us the total count of set bits in 4 bits. The last statement is the most tricky.

```
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >>
24;
```

Let's break it down further...

```
v + (v >> 4)
```

It's similar to the second statement; we are counting the set bits in groups of 4 instead. We know—because of our

previous operations—that every nibble has the count of set bits in it. Let's look an example. Suppose we have the byte `0b01000010`. It means the first nibble has its 4bits set and the second one has its 2bits set. Now we add those nibbles together.

```
v = 0b01000010
(v >> 4) = 0b00000100
v + (v >> 4) = 0b01000010 + 0b00000100
```

It gives us the count of set bits in a byte, in the second nibble `0b01000110` and therefore we mask the first four bytes of all the bytes in the number (discarding them).

```
0b01000110 & 0x0F = 0b00000110
```

Now every byte has the count of set bits in it. We need to add them up all together. The trick is to multiply the result by `0b10101010` which has an interesting property. If our number has four bytes, `A B C D`, it will result in a new number with these bytes `A+B+C+D B+C+D C+D D`. A 4 byte number can have maximum of 32 bits set, which can be represented as `0b00100000`.

All we need now is the first byte which has the sum of all set bits in all the bytes, and we get it by `>> 24`. This algorithm was designed for `32 bit` words but can be easily modified for `64 bit` words.

Share Improve this answer

edited Sep 26, 2022 at 16:02

Follow

community wiki
5 revs, 4 users 83%
vidit

1 What is the `c =` about? Looks like it should be eliminated. Further, suggest an extra paren set `A"(((v + (v >> 4)) & 0xF0F0F0F) * 0x1010101) >> 24"` to avoid some classic warnings. – [chux](#) Oct 15, 2013 at 15:40

4 An important feature is that this 32-bit routine works for both `popcount(int v)` and `popcount(unsigned v)`. For portability, consider `popcount(uint32_t v)`, etc. Really like the `*0x1010101` part. – [chux](#) Oct 15, 2013 at 15:49

sauce ? (book, link, inventors's names etc) would be VERY welcomed. Because then we can paste that in our codebases with a comment to where it comes from. – [v.oddou](#) Mar 31, 2015 at 1:34

2 I think for better clarity the last line should be written as:
`return (((i + (i >> 4)) & 0xF0F0F0F) * 0x01010101) >> 24;` so we don't need to count letters to see what you are actually doing (since you discarded the first `0`, I accidentally thought you used the wrong (flipped) bit pattern as mask - that is until I noted there are only 7 letters and not 8). – [emem](#) Feb 6, 2016 at 9:02 ✎

That **multiplication** by `0x01010101` might be slow, depending on processor. For example, in my old PowerBook G4, 1 multiplication was about as slow as 4 additions (not as bad as division, where 1 division was about as slow as 23 additions). – [George Koehler](#) Jan 3, 2018 at 3:05 ✎



If you happen to be using Java, the built-in method `Integer.bitCount` will do that.

63

Share Improve this answer

edited Jan 5, 2015 at 16:50

Follow



community wiki

2 revs, 2 users 67%

Noether

When sun provided different APIs, it must be using some logic on background, right? – [Vallabh Patade](#) Apr 12, 2013 at 10:19

2 As a side note, Java's implementation uses the **same** algorithm pointed out by [Kevin Little](#). – [Marco Bolis](#) Jan 5, 2015 at 16:37

4 Implementation aside, this is probably the clearest message of intent for developers maintaining your code after you (or when you come back to it 6 months later) – [divillysausages](#) May 4, 2017 at 10:08

**59**

I got bored, and timed a billion iterations of three approaches. Compiler is gcc -O3. CPU is whatever they put in the 1st gen Macbook Pro.



Fastest is the following, at 3.7 seconds:



```
static unsigned char wordbits[65536] = { bitcounts
of ints between 0 and 65535 };
static int popcount( unsigned int i )
{
    return( wordbits[i&0xFFFF] + wordbits[i>>16]
);
}
```


Second place goes to the same code but looking up 4 bytes instead of 2 halfwords. That took around 5.5 seconds.

Third place goes to the bit-twiddling 'sideways addition' approach, which took 8.6 seconds.

Fourth place goes to GCC's `__builtin_popcount()`, at a shameful 11 seconds.

The counting one-bit-at-a-time approach was waaaay slower, and I got bored of waiting for it to complete.

So if you care about performance above all else then use the first approach. If you care, but not enough to spend 64Kb of RAM on it, use the second approach. Otherwise use the readable (but slow) one-bit-at-a-time approach.

It's hard to think of a situation where you'd want to use the bit-twiddling approach.

Edit: Similar results [here](#).

Share Improve this answer

edited Sep 25, 2008 at 3:27

Follow

community wiki

3 revs

Mike F

54 @Mike, The table based approach is unbeatable if the table is in the cache. This happens in micro-benchmarks (e.g. do millions of tests in a tight loop). However, a cache miss takes


around 200 cycles, and even the most naive popcount will be faster here. It always depends on the application.

– [Nils Pipenbrinck](#) Sep 25, 2008 at 4:42

10 If you're not calling this routine a few million times in a tight loop then you have no reason to care about it's performance at all, and might as well use the naive-but-readable approach since the performance loss will be negligible. And FWIW, the 8bit LUT gets cache-hot within 10-20 calls. – Mike F Sep 25, 2008 at 11:02

8 I don't think it's all that hard to imagine a situation where this is a leaf call made from the method -actually doing the heavy lifting- in your app. Depending on what else is going on (and threading) the smaller version could win. Lots of algorithms have been written that beat their peers due to better locality of reference. Why not this too? – [Jason](#) May 6, 2010 at 8:50

Try this with clang, it's *significantly* smarter at implementing builtins. – [Matt Joiner](#) Oct 5, 2010 at 5:23

3 GCC won't emit popcont instruction unless called with -msse4.2, case which is faster than 'sideways addition'.
– [lvella](#) Jul 7, 2012 at 6:02 



41



```
unsigned int count_bit(unsigned int x)
{
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
    x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);
    return x;
}
```

Let me explain this algorithm.

This algorithm is based on Divide and Conquer Algorithm. Suppose there is a 8bit integer 213(11010101 in binary), the algorithm works like this(each time merge two neighbor blocks):

```
+-----+
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | <- x
| 1 0 | 0 1 | 0 1 | 0 1 | <- first time
merge
| 0 0 1 1 | 0 0 1 0 | <- second time
merge
| 0 0 0 0 0 1 0 1 | <- third time (
answer = 00000101 = 5)
+-----+
```

Share Improve this answer

answered [Aug 5, 2012 at 12:47](#)

Follow

community wiki
[abcdabcd987](#)

8 This algorithm is the version Matt Howells posted, before being optimized to the fact that it became unreadable.

– [Lefteris E](#) Jun 13, 2013 at 10:06

@LefterisE: Except using shift/add all the way to the end, instead of using a multiply to sum 8-bit chunks into the top 8 bits, replacing the last 2 `x=...` lines here. My edits demangled it some (keeping the optimized logic, improving readability and adding comments), and I added a section that explains the SWAR bithacks involved. This answer is still useful, though, at least for the diagram. Or for a hypothetical 32-bit machine with a slow multiply instruction.

– [Peter Cordes](#) Jun 18, 2022 at 2:49



30



This is one of those questions where it helps to know your micro-architecture. I just timed two variants under gcc 4.3.3 compiled with -O3 using C++ inlines to eliminate function call overhead, one billion iterations, keeping the running sum of all counts to ensure the compiler doesn't remove anything important, using rdtsc for timing (clock cycle precise).

```
inline int pop2(unsigned x, unsigned y)
{
    x = x - ((x >> 1) & 0x55555555);
    y = y - ((y >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) &
0x33333333);
    y = (y & 0x33333333) + ((y >> 2) &
0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    y = (y + (y >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    y = y + (y >> 8);
    x = x + (x >> 16);
    y = y + (y >> 16);
    return (x+y) & 0x000000FF;
}
```

The unmodified Hacker's Delight took 12.2 gigacycles. My parallel version (counting twice as many bits) runs in 13.0 gigacycles. 10.5s total elapsed for both together on a 2.4GHz Core Duo. 25 gigacycles = just over 10 seconds at this clock frequency, so I'm confident my timings are right.

This has to do with instruction dependency chains, which are very bad for this algorithm. I could nearly double the

speed again by using a pair of 64-bit registers. In fact, if I was clever and added $x+y$ a little sooner I could shave off some shifts. The 64-bit version with some small tweaks would come out about even, but count twice as many bits again.

With 128 bit SIMD registers, yet another factor of two, and the SSE instruction sets often have clever short-cuts, too.

There's no reason for the code to be especially transparent. The interface is simple, the algorithm can be referenced on-line in many places, and it's amenable to comprehensive unit test. The programmer who stumbles upon it might even learn something. These bit operations are extremely natural at the machine level.

OK, I decided to bench the tweaked 64-bit version. For this one `sizeof(unsigned long) == 8`

```
inline int pop2(unsigned long x, unsigned long y)
{
    x = x - ((x >> 1) & 0x5555555555555555);
    y = y - ((y >> 1) & 0x5555555555555555);
    x = (x & 0x3333333333333333) + ((x >> 2) &
0x3333333333333333);
    y = (y & 0x3333333333333333) + ((y >> 2) &
0x3333333333333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F0F0F0F0F;
    y = (y + (y >> 4)) & 0x0F0F0F0F0F0F0F0F;
    x = x + y;
    x = x + (x >> 8);
    x = x + (x >> 16);
    x = x + (x >> 32);
}
```

```
    return x & 0xFF;
}
```

That looks about right (I'm not testing carefully, though). Now the timings come out at 10.70 gigacycles / 14.1 gigacycles. That later number summed 128 billion bits and corresponds to 5.9s elapsed on this machine. The non-parallel version speeds up a tiny bit because I'm running in 64-bit mode and it likes 64-bit registers slightly better than 32-bit registers.

Let's see if there's a bit more OOO pipelining to be had here. This was a bit more involved, so I actually tested a bit. Each term alone sums to 64, all combined sum to 256.

```
inline int pop4(unsigned long x, unsigned long y,
               unsigned long u, unsigned long v)
{
    enum { m1 = 0x5555555555555555,
          m2 = 0x3333333333333333,
          m3 = 0x0F0F0F0F0F0F0F0F,
          m4 = 0x000000FF000000FF };

    x = x - ((x >> 1) & m1);
    y = y - ((y >> 1) & m1);
    u = u - ((u >> 1) & m1);
    v = v - ((v >> 1) & m1);
    x = (x & m2) + ((x >> 2) & m2);
    y = (y & m2) + ((y >> 2) & m2);
    u = (u & m2) + ((u >> 2) & m2);
    v = (v & m2) + ((v >> 2) & m2);
    x = x + y;
    u = u + v;
    x = (x & m3) + ((x >> 4) & m3);
    u = (u & m3) + ((u >> 4) & m3);
    x = x + u;
    x = x + (x >> 8);
```

```

    x = x + (x >> 16);
    x = x & m4;
    x = x + (x >> 32);
    return x & 0x000001FF;
}

```

I was excited for a moment, but it turns out gcc is playing inline tricks with -O3 even though I'm not using the inline keyword in some tests. When I let gcc play tricks, a billion calls to pop4() takes 12.56 gigacycles, but I determined it was folding arguments as constant expressions. A more realistic number appears to be 19.6gc for another 30% speed-up. My test loop now looks like this, making sure each argument is different enough to stop gcc from playing tricks.

```

    hitime b4 = rdtsc();
    for (unsigned long i = 10L * 1000*1000*1000; i
< 11L * 1000*1000*1000; ++i)
        sum += pop4 (i, i^1, ~i, i|1);
    hitime e4 = rdtsc();

```

256 billion bits summed in 8.17s elapsed. Works out to 1.02s for 32 million bits as benchmarked in the 16-bit table lookup. Can't compare directly, because the other bench doesn't give a clock speed, but looks like I've slapped the snot out of the 64KB table edition, which is a tragic use of L1 cache in the first place.

Update: decided to do the obvious and create pop6() by adding four more duplicated lines. Came out to 22.8gc, 384 billion bits summed in 9.5s elapsed. So there's another 20% Now at 800ms for 32 billion bits.

Share Improve this answer

edited Oct 3, 2009 at 0:17

Follow

community wiki

6 revs

user183351

-
- 2 The best non-assembler form like this I've seen unrolled 24 32bit words at a time.

dalkescientific.com/writings/diary/popcnt.c,

stackoverflow.com/questions/3693981/...,

dalkescientific.com/writings/diary/archive/2008/07/05/...

– Matt Joiner Oct 5, 2010 at 5:25



Why not iteratively divide by 2?

30



```
count = 0
while n > 0
    if (n % 2) == 1
        count += 1
    n /= 2
```



I agree that this isn't the fastest, but "best" is somewhat ambiguous. I'd argue though that "best" should have an element of clarity

Share Improve this answer

edited Dec 2, 2015 at 13:30

Follow

community wiki

3 revs, 2 users 95%

daniel

-
- 2 Unless you do this a *LOT*, the performance impact would be negligible. So all things being equal, I agree with daniel that 'best' implies "doesn't read like gibberish". – Mike F Sep 20, 2008 at 21:50
-
- 2 I deliberately didn't define 'best', to get a variety of methods. Lets face it if we have got down to the level of this sort of bit-twiddling we are probably looking for something uber-fast that looks like a chimp has typed it. – [Matt Howells](#) Sep 21, 2008 at 17:47
-
- 6 Bad code. A compiler might make good one out of it, but in my tests GCC did not. Replace (n%2) with (n&1); AND being much faster than MODULO. Replace (n/=2) with (n>>=1); bitshifting much faster than division. – [Mecki](#) Sep 25, 2008 at 10:35
-
- 6 @Mecki: In my tests, gcc (4.0, -O3) *did* do the obvious optimisations. – Mike F Sep 25, 2008 at 13:32
-
- 1 Negative values of `n` always return 0. – [chux](#) Oct 15, 2013 at 15:22
-



The Hacker's Delight bit-twiddling becomes so much clearer when you write out the bit patterns.

27



```
unsigned int bitCount(unsigned int x)
{
    x = ((x >> 1) &
0b01010101010101010101010101010101)
        + (x &
0b01010101010101010101010101010101);
    x = ((x >> 2) &
0b00110011001100110011001100110011)
        + (x &
0b00110011001100110011001100110011);
```

```

    x = ((x >> 4) &
0b000001111000001111000001111000001111)
        + (x
&
0b000001111000001111000001111000001111);
    x = ((x >> 8) &
0b0000000001111111100000000011111111)
        + (x
&
0b0000000001111111100000000011111111);
    x = ((x >> 16)&
0b0000000000000000000000001111111111111111)
        + (x
&
0b0000000000000000000000001111111111111111);
    return x;
}

```

The first step adds the even bits to the odd bits, producing a sum of bits in each two. The other steps add high-order chunks to low-order chunks, doubling the chunk size all the way up, until we have the final count taking up the entire int.

Share Improve this answer


edited Mar 11, 2019 at 10:55

Follow


community wiki

2 revs, 2 users 77%

John Dimm

3 This solution seem to have minor problem, related to operator precedence. For each term it should say: $x = (((x \gg 1) \& 0b01010101010101010101010101010101) + (x \& 0b01010101010101010101010101010101));$ (i.e. extra parens added). – [Nopik](#) Aug 22, 2014 at 7:38 

1 In case you're confused, the error in the original article that @Nopik pointed out has since been fixed (by someone else),

and without newly introducing *extraneous* parentheses as the comment suggests. – [Glenn Slayden](#) Jan 18, 2022 at 3:33 



For a happy medium between a 2^{32} lookup table and iterating through each bit individually:

23



```
int bitcount(unsigned int num){
    int count = 0;
    static int nibblebits[] =
        {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3,
3, 4};
    for(; num != 0; num >>= 4)
        count += nibblebits[num & 0x0f];
    return count;
}
```

From <http://ctips.pbwiki.com/CountBits>

Share Improve this answer

answered [Sep 22, 2008 at 3:55](#)

Follow

community wiki
[PhirePhly](#)

Not portable. What if the CPU has 9 bit bytes? Yes, there are real CPU's like that out there... – [Robert S. Barnes](#) Mar 29, 2011 at 8:06

18 @Robert S. Barnes, this function will still work. It makes no assumption about native word size, and no reference to "bytes" at all. – [finnw](#) May 8, 2011 at 11:26

Is the complexity of this code `O(floor(log2(num))/4)`, assuming `num` can be as arbitrarily large as possible?

Because the `while` loop runs as long as there's a nibble to process? There are `floor(log2(num))` bits and `floor(log2(num)) / 4` nibbles. Is the reasoning correct?
– [Robur_131](#) Jun 12, 2020 at 19:28

@Robur_131 I don't see anything wrong with your reasoning, except that big-O doesn't care about constant factors so you could simplify to just $O(\log n)$. The nice thing about this algorithm is it doesn't always take worst case, if the upper bits are zero it exits early. In fact for an input of zero the loop doesn't run at all. – [Mark Ransom](#) May 16, 2021 at 15:55

@MarkRansom: So unless you're optimizing for inputs of zero, you should probably change it to a `do{}while` loop that doesn't compare/branch until after checking the low 4 bits. Or unroll to check 2x 4 bits on the first iteration. That means the branch-prediction pattern is the same for all inputs from 0..255, and making the branching more predictable is often a Good Thing, when the work saved is much cheaper than a branch mispredict. (Of course that depends on the CPU, and you wouldn't typically use this on a high-end CPU where branch misses are most expensive.)
– [Peter Cordes](#) Jun 18, 2022 at 2:55

This can be done in `o(k)`, where `k` is the number of bits set.

19

```
int NumberOfSetBits(int n)
{
    int count = 0;

    while (n){
        ++ count;
        n = (n - 1) & n;
    }
}
```

```
    return count;
}
```

Share Improve this answer

edited Aug 3, 2015 at 15:34

Follow

community wiki

2 revs

[herohuyongtao](#)

- 1 This is essentially **Brian Kernighan's** (remember him?) algorithm, with the minor change that he used the more succinct `n &= (n-1)` form. – [Adrian Mole](#) Aug 23, 2019 at 13:05



17



It's not the fastest or best solution, but I found the same question in my way, and I started to think and think. finally I realized that it can be done like this if you get the problem from mathematical side, and draw a graph, then you find that it's a function which has some periodic part, and then you realize the difference between the periods... so here you go:

```
unsigned int f(unsigned int x)
{
    switch (x) {
        case 0:
            return 0;
        case 1:
            return 1;
        case 2:
            return 1;
        case 3:
```

```

        return 2;
    default:
        return f(x/4) + f(x%4);
    }
}

```

Share Improve this answer

edited Oct 19, 2012 at 12:34

Follow

community wiki

2 revs, 2 users 97%

Peter

4 oh i like that. how bout the python version: `def f(i, d={0:lambda:0, 1:lambda:1, 2:lambda:1, 3:lambda:2}): return d.get(i, lambda: f(i//4) + f(i%4))()` – [underrun](#) Feb 1, 2013 at 19:04



12



I think the [Brian Kernighan's](#) method will be useful too... It 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.

```

/** count the number of bits set in n */
int countSetBits(unsigned int n) {
    unsigned int c; // c accumulates the total bits set
    for (c=0; n>0; n=n&(n-1)) c++;
    return c;
}

```

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.)"

Share Improve this answer

edited Jun 5 at 8:27

Follow

community wiki

4 revs, 4 users 48%

Error



11



```
private int get_bits_set(int v)
{
    int c; // 'c' accumulates the total bits set in 'v'
    for (c = 0; v>0; c++)
    {
        v &= v - 1; // Clear the least significant bit
    }
    return c;
}
```

Share Improve this answer

edited Aug 26, 2022 at 21:07

Follow

community wiki
3 revs, 3 users 53%
stacktay



10



The function you are looking for is often called the "sideways sum" or "population count" of a binary number. Knuth discusses it in pre-Fascicle 1A, pp11-12 (although there was a brief reference in Volume 2, 4.6.3-(7).)

The *locus classicus* is Peter Wegner's article "A Technique for Counting Ones in a Binary Computer", from the [Communications of the ACM, Volume 3 \(1960\) Number 5, page 322](#). He gives two different algorithms there, one optimized for numbers expected to be "sparse" (i.e., have a small number of ones) and one for the opposite case.

Share Improve this answer

edited Mar 4, 2014 at 6:42



Few open questions:-

9



1. If the number is negative then?
2. If the number is 1024 , then the "iteratively divide by 2" method will iterate 10 times.



we can modify the algo to support the negative number as follows:-



```
count = 0
while n != 0
if ((n % 2) == 1 || (n % 2) == -1
    count += 1
    n /= 2
return count
```

now to overcome the second problem we can write the algo like:-

```
int bit_count(int num)
{
    int count=0;
    while(num)
    {
        num=(num)&(num-1);
        count++;
    }
}
```

```
    return count;
}
```

for complete reference see :

<http://goursaha.freeoda.com/Miscellaneous/IntegerBitCount.html>

Share Improve this answer

edited Mar 3, 2011 at 22:30

Follow

community wiki
2 revs, 2 users 72%
Baban



I use the below code which is more intuitive.

8

```
int countSetBits(int n) {
    return !n ? 0 : 1 + countSetBits(n & (n-1));
}
```



Logic : $n \& (n-1)$ resets the last set bit of n .



P.S : I know this is not $O(1)$ solution, albeit an interesting solution.

Share Improve this answer

edited Sep 12, 2016 at 13:57

Follow

community wiki

this is good for "sparse" numbers with a low number of bits, as it is `O(ONE-BITS)`. It is indeed $O(1)$ since there are at most 32 one-bits. – [ealfonso](#) Jan 14, 2018 at 19:55



7



What do you means with "Best algorithm"? The shorted code or the fasted code? Your code look very elegant and it has a constant execution time. The code is also very short.

But if the speed is the major factor and not the code size then I think the follow can be faster:

```
static final int[] BIT_COUNT = { 0, 1, 1,
... 256 values with a bitsize of a byte ... };
static int bitCountOfByte( int value ){
    return BIT_COUNT[ value & 0xFF ];
}

static int bitCountOfInt( int value ){
    return bitCountOfByte( value )
        + bitCountOfByte( value >> 8 )
        + bitCountOfByte( value >> 16 )
        + bitCountOfByte( value >> 24 );
}
```

I think that this will not more faster for a 64 bit value but a 32 bit value can be faster.

Share Improve this answer

[edited Nov 24, 2009 at 10:21](#)

Follow

community wiki
2 revs, 2 users 73%
[Horcrux7](#)

My code has 10 operation. Your code has 12 operation. Your link work with smaller arrays (5). I use 256 elements. With the caching can be a problem. But if you use it very frequently then this is not a problem. – [Horcrux7](#) Sep 20, 2008 at 21:12

This approach is measurably quite a bit faster than the bit-twiddling approach, as it turns out. As for using more memory, it compiles to less code and that gain is repeated every time you inline the function. So it could easily turn out to be a net win. – Mike F Sep 25, 2008 at 3:43



7



I wrote a fast bitcount macro for RISC machines in about 1990. It does not use advanced arithmetic (multiplication, division, %), memory fetches (way too slow), branches (way too slow), but it does assume the CPU has a 32-bit barrel shifter (in other words, $\gg 1$ and $\gg 32$ take the same amount of cycles.) It assumes that small constants (such as 6, 12, 24) cost nothing to load into the registers, or are stored in temporaries and reused over and over again.

With these assumptions, it counts 32 bits in about 16 cycles/instructions on most RISC machines. Note that 15 instructions/cycles is close to a lower bound on the number of cycles or instructions, because it seems to take at least 3 instructions (mask, shift, operator) to cut

the number of addends in half, so $\log_2(32) = 5$, $5 \times 3 = 15$ instructions is a quasi-lowerbound.

```
#define BitCount(X,Y) \
    Y = X - ((X >> 1) & 033333333333) \
    - ((X >> 2) & 011111111111); \
    Y = ((Y + (Y >> 3)) & \
    030707070707); \
    Y = (Y + (Y >> 6)); \
    Y = (Y + (Y >> 12) + (Y >> 24)) & \
    077;
```

Here is a secret to the first and most complex step:

input	output	
AB	CD	Note
00	00	= AB
01	01	= AB
10	01	= AB - (A >> 1) & 0x1
11	10	= AB - (A >> 1) & 0x1

so if I take the 1st column (A) above, shift it right 1 bit, and subtract it from AB, I get the output (CD). The extension to 3 bits is similar; you can check it with an 8-row boolean table like mine above if you wish.

- Don Gillies

Share Improve this answer

edited Jun 11, 2010 at 21:49

Follow

community wiki
2 revs
systemBuilder



if you're using C++ another option is to use template metaprogramming:

7



```
// recursive template to sum bits in an int
template <int BITS>
int countBits(int val) {
    // return the least significant bit plus
    // the result of calling ourselves with
    // .. the shifted value
    return (val & 0x1) + countBits<BITS-1>(val
    >> 1);
}

// template specialisation to terminate the
// recursion when there's only one bit left
template<>
int countBits<1>(int val) {
    return val & 0x1;
}
```

usage would be:

```
// to count bits in a byte/char (this returns 8)
countBits<8>( 255 )

// another byte (this returns 7)
countBits<8>( 254 )

// counting bits in a word/short (this returns 1)
countBits<16>( 256 )
```

you could of course further expand this template to use different types (even auto-detecting bit size) but I've kept it simple for clarity.

edit: forgot to mention this is good because it *should* work in any C++ compiler and it basically just unrolls your loop for you if a constant value is used for the **bit count** (in other words, I'm pretty sure it's the fastest general method you'll find)

Share Improve this answer

edited Apr 4, 2012 at 4:31

Follow

community wiki

2 revs

pentaphobe

Unfortunately, the bit counting isn't done in parallel, so it's probably slower. Might make a nice `constexpr` though.

– [geometrian](#) Jul 3, 2015 at 15:14

Agreed - it was a fun exercise in C++ template recursion, but definitely a pretty naïve solution. – [pentaphobe](#) Sep 25, 2015 at 6:46



C++20 `std::popcount`

7

The following proposal has been merged

[http://www.open-](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0553r4.html)

[std.org/jtc1/sc22/wg21/docs/papers/2019/p0553r4.html](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0553r4.html)

and should add it to the `<bit>` header.



I expect the usage to be like:



```
#include <bit>
#include <iostream>
```

```
int main() {  
    std::cout << std::popcount(0x55) << std::endl;  
}
```

I'll give it a try when support arrives to GCC, GCC 9.1.0 with `g++-9 -std=c++2a` still doesn't support it.

The proposal says:

Header: `<bit>`

```
namespace std {  
  
    // 25.5.6, counting  
    template<class T>  
        constexpr int popcount(T x) noexcept;
```

and:

```
template<class T>  
    constexpr int popcount(T x) noexcept;
```

Constraints: T is an unsigned integer type (3.9.1 [basic.fundamental]).

Returns: The number of 1 bits in the value of x.

`std::rotl` and `std::rotr` were also added to do circular bit rotations: [Best practices for circular shift \(rotate\) operations in C++](#)

Share Improve this answer

edited Jul 31, 2019 at 8:15

Follow

community wiki

3 revs

Ciro Santilli 新疆改造中心
996ICU六四事件



You can do:

7



```
while(n){  
    n = n & (n-1);  
    count++;  
}
```



The logic behind this is the bits of $n-1$ is inverted from rightmost set bit of n .

If $n=6$, i.e., 110 then 5 is 101 the bits are inverted from rightmost set bit of n .

So if we `&` these two we will make the rightmost bit 0 in every iteration and always go to the next rightmost set bit. Hence, counting the set bit. The worst time complexity will be $O(\log n)$ when every bit is set.

Share Improve this answer

edited Aug 26, 2022 at 21:21

Follow

community wiki

What programming language? – [Peter Mortensen](#) Aug 26, 2022 at 21:21



I'm particularly fond of this example from the fortune file:

6



```
#define BITCOUNT(x)      (((BX_(x)+(BX_(x)>>4)) &  
0x0F0F0F0F) % 255)  
#define BX_(x)            ((x) -  
(((x)>>1)&0x77777777))  
                           -  
(((x)>>2)&0x33333333)  
                           -  
(((x)>>3)&0x11111111))
```

I like it best because it's so pretty!

Share Improve this answer

answered [Sep 23, 2008 at 1:29](#)

Follow

community wiki
[Ross](#)

-
- 1 How does it perform compared to the other suggestions?
– [asdf](#) Jul 1, 2011 at 16:08
-



Java JDK1.5

Integer.bitCount(n);

6

where n is the number whose 1's are to be counted.



check also,



```
Integer.highestOneBit(n);
Integer.lowestOneBit(n);
Integer.numberOfLeadingZeros(n);
Integer.numberOfTrailingZeros(n);

//Beginning with the value 1, rotate left 16 times
n = 1;
for (int i = 0; i < 16; i++) {
    n = Integer.rotateLeft(n, 1);
    System.out.println(n);
}
```

Share Improve this answer

edited Apr 27, 2011 at 17:26

Follow

community wiki

2 revs, 2 users 77%

Rahul

1 Not really an algorithm, this is just a library call. Useful for Java, not so much for everybody else. – [benzado](#) Dec 13, 2010 at 5:14

2 @benzado is right but +1 anyway, because some Java developers might not be aware of the method – [finnw](#) May 8, 2011 at 11:27

@finnw, i am one of those developers. :) – [neevak](#) Nov 12, 2013 at 8:37



6



I found an implementation of bit counting in an array with using of SIMD instruction (SSSE3 and AVX2). It has in 2-2.5 times better performance than if it will use `__popcnt64` intrinsic function.

SSSE3 version:

```
#include <smmintrin.h>
#include <stdint.h>

const __m128i Z = _mm_set1_epi8(0x0);
const __m128i F = _mm_set1_epi8(0xF);
//Vector with pre-calculated bit count:
const __m128i T = _mm_setr_epi8(0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4);

uint64_t BitCount(const uint8_t * src, size_t size)
{
    __m128i _sum = _mm128_setzero_si128();
    for (size_t i = 0; i < size; i += 16)
    {
        //load 16-byte vector
        __m128i _src = _mm_loadu_si128((__m128i*)(src + i));
        //get low 4 bit for every byte in vector
        __m128i lo = _mm_and_si128(_src, F);
        //sum precalculated value from T
        _sum = _mm_add_epi64(_sum, _mm_sad_epu8(Z, _mm_shuffle_epi8(T, lo)));
        //get high 4 bit for every byte in vector
        __m128i hi = _mm_and_si128(_mm_srli_epi16(_src, 4), F);
        //sum precalculated value from T
        _sum = _mm_add_epi64(_sum, _mm_sad_epu8(Z, _mm_shuffle_epi8(T, hi)));
    }
    uint64_t sum[2];
    _mm_storeu_si128((__m128i*)sum, _sum);
}
```

```

    return sum[0] + sum[1];
}

```

AVX2 version:

```

#include <immintrin.h>
#include <stdint.h>

const __m256i Z = _mm256_set1_epi8(0x0);
const __m256i F = _mm256_set1_epi8(0xF);
//Vector with pre-calculated bit count:
const __m256i T = _mm256_setr_epi8(0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
                                0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4);

uint64_t BitCount(const uint8_t * src, size_t
size)
{
    __m256i _sum = _mm256_setzero_si256();
    for (size_t i = 0; i < size; i += 32)
    {
        //load 32-byte vector
        __m256i _src =
_mm256_loadu_si256((__m256i*)(src + i));
        //get low 4 bit for every byte in vector
        __m256i lo = _mm256_and_si256(_src, F);
        //sum precalculated value from T
        _sum = _mm256_add_epi64(_sum,
_mm256_sad_epu8(Z, _mm256_shuffle_epi8(T, lo)));
        //get high 4 bit for every byte in vector
        __m256i hi =
_mm256_and_si256(_mm256_srli_epi16(_src, 4), F);
        //sum precalculated value from T
        _sum = _mm256_add_epi64(_sum,
_mm256_sad_epu8(Z, _mm256_shuffle_epi8(T, hi)));
    }
    uint64_t sum[4];
    _mm256_storeu_si256((__m256i*)sum, _sum);
    return sum[0] + sum[1] + sum[2] + sum[3];
}

```

community wiki

2 revs

Ermlg



6



A fast C# solution using a pre-calculated table of Byte bit counts with branching on the input size.

```
public static class BitCount
{
    public static uint GetSetBitsCount(uint n)
    {
        var counts = BYTE_BIT_COUNTS;
        return n <= 0xff ? counts[n]
            : n <= 0xffff ? counts[n & 0xff] + counts
            : n <= 0xffffffff ? counts[n & 0xff] + coun
counts[(n >> 16) & 0xff]
            : counts[n & 0xff] + counts[(n >> 8) & 0x
0xff] + counts[(n >> 24) & 0xff];
    }

    public static readonly uint[] BYTE_BIT_COUNTS =
    {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6
    }
```

```

        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7
        4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
    };
}

```

Share Improve this answer


edited Aug 26, 2022 at 20:44


Follow


community wiki

3 revs, 2 users 64%

[dadhi](#)

Ironically, that table could have been created by any of the algorithms posted in this thread! Nevertheless, using tables like this means constant-time performance. Going one step further and creating a 64K translation table would therefore halve the AND, SHIFT and ADD operations necessary. An interesting subject for bit manipulators! – [KRK Owner](#) Jan 10, 2016 at 23:23 

Bigger tables can be slower (and not constant-time) due to cache issues. You can 'look up' 3 bits at a time with `(0xe994 >> (k*2)) & 3`, without memory access... – [greggo](#) Mar 28, 2017 at 18:42 

Use `BitOperations.PopCount((uint)number);`
– [Wouter](#) Dec 30, 2023 at 0:45 



I always use this in [competitive programming](#), and it's easy to write and is efficient:

6

```
#include <bits/stdc++.h>
```



```
using namespace std;
```

```
int countOnes(int n) {  
    bitset<32> b(n);  
    return b.count();  
}
```

Share Improve this answer

edited Aug 26, 2022 at 20:54

Follow

community wiki

2 revs, 2 users 72%

iamdeit



5

There are many algorithm to count the set bits; but i think the best one is the faster one! You can see the detailed on this page:



[Bit Twiddling Hacks](#)



I suggest this one:



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 & 0x11111111111111ULL)  
% 0xf;
```

```
// option 2, for at most 24-bit values in v:
```



```
c = ((v & 0xffff) * 0x1001001001001ULL &
0x84210842108421ULL) % 0x1f;
c += (((v & 0xffff000) >> 12) * 0x1001001001001ULL
& 0x84210842108421ULL)
    % 0x1f;

// option 3, for at most 32-bit values in v:
c = ((v & 0xffff) * 0x1001001001001ULL &
0x84210842108421ULL) % 0x1f;
c += (((v & 0xffff000) >> 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.

Share Improve this answer

answered [Apr 13, 2011 at 7:50](#)

Follow

community wiki
[Mostafa](#)

1

2

3

Next



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.