[WORKING TUTORIAL] Computing integer $\log_2(n)$

Rui-Jie Fang

Synopsis We discuss several different ways to efficiently compute the integer-log2 (or, the most significant bit) of an integer. We start from a few obvious ways into ways that are harder to think of but are more efficient.

Note Citations come as footnotes to links. Sections 3+ describe the work of others and others only. The author is too lazy to configure bibtex :(

Note We assume that the reader is familiar with conversion between binary and hexadecimals. A review of hexadecimal-binary conversion can be found at the footnote[1].

Caution: Draft - Sections Are Subject To Change And Prone To Errors

1 Standard way

Integer-$\log_2(n)$ is equivalent to the 0-indexed position of the MSB of n(assuming always round down). Hence the question is equivalent to asking the position of $\text{msb}(n)$. It's helpful to think about why it is the case.

The ith bit corresponds to 2^i in decimal. Integer $\log_2(n)$ (rounding down) has to be an i such that $2^i \leq n \leq 2^{i+1}$.

That trivially leads to the following algorithm running in $O(\log n)$ time with the number of branches being $O(\log n)$:

```
function ilog2(unsigned int n) {
    unsigned int count;
    while (n>>=1) {
        ++count;
    }
    return count;
}
```

This is apparently not creative enough.

[1] https://nx.magikpns.net/resources/hexbin.html.

2 Binary search

We can first think of using binary search to cut the number of iterations to $O(\log \log n)$ (However, this does not deal with the overwhelming amount of branches):

```
#include <stdio.h>
#include <math.h>
unsigned int table[32] =
{ 0x1, 0x2, 0x4, 0x8, 0x10,
  0x20, 0x40, 0x80, 0x100, 0x200,
  0x400, 0x800, 0x1000, 0x2000, 0x4000,
  0x8000, 0x10000, 0x20000, 0x40000, 0x80000,
  0x100000, 0x200000, 0x400000, 0x800000, 0x1000000,
  0x2000000, 0x4000000, 0x8000000, 0x10000000, 0x20000000,
  0x40000000, 0x80000000 }; /* 1<<i, 0<=i<size */
const unsigned int size = 32; /* size of table */
#define mid(x,y)((x)+((y)-(x))/2)
static inline unsigned int diff(unsigned int L, unsigned int R)
{ /* can be branchless */
    return (unsigned int)abs((int)((int)R-(int)L));
}
unsigned int ilog2(int s)
{
    if (s < 0) return  0;
    unsigned int x = (int) s;
    if(x==0||x==1)return 0;
    unsigned int L = 0, R = size, M = mid(L,R), invs = 0;
    while (diff(L,R) > 1)  {
        if (x > table[M]) {
            L = M;
            M = mid(L,R);
            continue;
        } else if (x < table[M]) {
            R = M;
            M = mid(L,R);
            continue;
        } else {
            return M;
        }
    }
    return x>table[L] && x<table[R] ? L : R;
```

```
}
```

(We don't have to use a table but we use one here for convenience)

The only culprit for this is the number of branches, which is 5 for each iteration, plus 1 for the return statement, which makes the entire cost become

$$O(T(n)) = O(C_{instruction} \cdot \log^2 n + C_{branch} \cdot (3 \log^2 n + 1))$$

We can further cut down the branching amount by 1 or 2 per iteration by eliminating the branching at absolute value computation, but that is still a lot.

3 Idea similar to `glibc`'s vectorized `strlen`, but in a byte per cycle[2]

glibc's strlen() function processes length of strings in chunks of words by casting the entire string to a `size_t`. We may do something similar by casting to a `char` and process in chunks of 8 bits (for us it's a bit simpler, as our in-loop check is just for checking if there's 1s in the current byte). Code below:

```
unsigned int ilog2_v(int s) /* vectorized ilog2 */
{
    if (s < 0) return 0;
    unsigned int x = (unsigned int) s, count = 0, r = 0;
    if (x == 0 || x == 1) return 0;
    char *p = &(x); /* vectorize */
    while (r < 4 && p[r]) count += 4, ++r;
    if (x >> count) return count;
    if (x >> (count-1)) return count-1;
    if (x >> (count-2)) return count-2;
    if (x >> (count-3)) return count-3;
    if (x >> (count-4)) return count-4;
}
```

This gives us time $n/4$, but still with a crazy amount of branches. This is similar to carry-lookahead adders. For 64bit integers, we can add another level going in 16bits at a time (using shorts) which then uses our byte-level ilog2 to reach the solution (analogous to a two-layer carry-lookahead adder scheme). Of course, we can also add a level of int32 for the ultimate tradeoff on 64bit integers, which first cuts us down by half of the input (but again, the number of functions would grow exponentially for bigints if we always want to cut down by a half). Most conventional bithack algorithms work as a combination of word-level parallelism and unrolled binary search, which are the ideas that this section and section 2 stemmed from.

4 `gcc` 's built-in function

This is probably the fastest one (and the easiest one)... For users of gcc, the following function

[2]For the glibc implementation, see `https://stackoverflow.com/questions/20021066/how-the-glibc-strlen-implementation-works`

```
int __builtin_clz (unsigned int x)
```

Returns the leading zeroes of an (unsigned) integer. Given the known size of integer, it gets us the msb. We also note obligatorily that gcc has a few other relevant builtin functions:

```
int __builtin_ctz (unsigned int x) // computes trailing zeroes
int __builtin_popcount (unsigned int x) // determines number of ones
```

5 Inventions by other people

There are in general two ways to get ilog2. One way is to "bite the bullet" and compute msb(n), the other way being computing the number of leading zeroes, which when given with the size of the integer, gives us ilog2(n).

5.1 Computing ilog2 through msb

5.1.1 The SWAR algorithm

The bit twiddling ilog2 from Henry Gordon Diet's aggregate.org link[3] suggests the use of SWAR algorithms[4], packing everything into a bit vector; it also provides a way to get the ceiling of base 2 log. The floored version is as follows:

```
unsigned int floor_log2(register unsigned int x) {
    x |= (x >> 1);
    x |= (x >> 2);
    x |= (x >> 4);
    x |= (x >> 8);
    x |= (x >> 16);
#ifdef LOGOUNDEFINED
    return(ones32(x) - 1);
#else
    return(ones32(x >> 1));
#endif
}
```

The first few bitwise-OR operations are actually a divide-and-conquer scheme, which is a lot of fun to learn[5]. The first OR fills neighboring holes (i.e. if the nth position is turned on, also turn on the $n-1$th position). The second OR fills two adjacent 2-bit positions The third OR fills two adjacent 4-bit positions and so on. To see this, the best way is to do this procedure by hand (or in a good REPL) on a number like 0x8888:

[3] `http://aggregate.org/MAGIC/#Log2\%20of\%20an\%20Integer`.
[4] SIMD within a register, `https://en.wikipedia.org/wiki/SWAR`.
[5] Actually, their website contains more of this stuff. Highly recommended.

```
x = 0x8888; // bit pattern: 0b1000100010001000
x |= x >> 1; // bit pattern: 0b1100110011001100
x |= x >> 2; // bit pattern: 0b1111111111111111
```

Or to better illustrate, we may choose 0x8000 (a maximum 16-bit integer), which creates a bit-vector with LSB at the 31st position:

```
x = 0x8000; // bit pattern: 0b1000000000000000
x |= x >> 1; // bit pattern: 0b1100000000000000
x |= x >> 2; // bit pattern: 0b1111000000000000
x |= x >> 4; // bit pattern: 0b1111111100000000
x |= x >> 8; // bit pattern: 0b1111111111111111
```

For dummy-dummies, we can use 0x8001 to illustrate how the rightmost bits are preserved:

```
x = 0x8001; // bit pattern: 0b1000000000000001
x |= x >> 1; // bit pattern: 0b1100000000000001
x |= x >> 2; // bit pattern: 0b1111000000000001
x |= x >> 4; // bit pattern: 0b1111111100000001
x |= x >> 8; // bit pattern: 0b1111111111111111
```

Hence the folding goes strictly from right-to-left and does not change the position of the msb.

The `ones32` function is a population count. Again, gcc users can use the built in `__builtin_popcount`. The same page at aggregate.org also contains a SWAR algorithm for population count:

```
unsigned int ones32(register unsigned int x) {

            /* 32-bit recursive reduction using SWAR...
       but first step is mapping 2-bit values
       into sum of 2 1-bit values in sneaky way
        */
            x -= ((x >> 1) & 0x55555555); /* 8x (0101) bit pattern */
            x = (((x >> 2) & 0x33333333) + (x & 0x33333333)); /* 8x (0011) bit pattern */
            x = (((x >> 4) + x) & 0x0f0f0f0f);
            x += (x >> 8);
            x += (x >> 16);
            return(x & 0x0000003f);
    }
```

The above function may not seem obvious at first. Joerg Arndt provides a more comprehendable way for population count in his book Matters Computational:

```
static inline ulong bit_count(ulong x) {


    x = (0x5555555555555555UL & x) + (0x5555555555555555UL & (x>> 1)); // 0-2 in 2 bits
    x = (0x3333333333333333UL & x) + (0x3333333333333333UL & (x>> 2)); // 0-4 in 4 bits
    x = (0x0f0f0f0f0f0f0f0fUL & x) + (0x0f0f0f0f0f0f0f0fUL & (x>> 4)); // 0-8 in 8 bits
```

```
    x = (0x00ff00ff00ff00ffUL & x) + (0x00ff00ff00ff00ffUL & (x>> 8)); // 0-16 in 16 bits
    x = (0x0000ffff0000ffffUL & x) + (0x0000ffff0000ffffUL & (x>>16)); // 0-32 in 32 bits
    x = (0x00000000ffffffffUL & x) + (0x00000000ffffffffUL & (x>>32)); // 0-64 in 64 bits
    return x;
}
```

Each bitmask are made so that adjacent bits are counted in a tree-like fashion. The right shifts make adjacent bits countable. As he mentions,

> The underlying idea is to do a search via bit masks.

It is worth mentioning that the bitmask technique is easily generalizable and is extremely useful to other applications.

(There's also a trivial way to get a population count; make a table of size MAX_INT and fill in the corresponding numbers for indexes 0x0, 0x1, 0x3, 0x7, 0xf, 0xf1, 0xf3, 0xf7, 0xff... and so on, until 0xffffffffffffffffffffffffffffffff; but this isn't optimal because the table size is too large. This inspires the de Bruijn tables approach) Interestingly, the same webpage also provides a significantly simpler way of finding msb without using a population count:

```
unsigned int msb32(register unsigned int x) {
            x |= (x >> 1);
            x |= (x >> 2);
            x |= (x >> 4);
            x |= (x >> 8);
            x |= (x >> 16);
            return(x & ~(x >> 1));
}
```

Alternatively, the divide-and-conquer 1bit mirroring scheme can be combined with a vectorized algorithm or binary search discussed in sections 2 and 3 if the population count seems too obscure.

This section is To Be finalized.

5.1.2 The Stanford Graphics Bithack Algorithm

Sean Anderson maintained the wildly popular Stanford Graphics bithack page, which also contains various ways to get msb or integer log2.

This section is To Be written.

5.2 Computing ilog2 through leading zeroes

Hacker's Delight[6], again, provides us with a million ways of getting the leading zeroes of n.

This section is To Be written.

[6]The chapter for computing leading zeroes: `http://www.hackersdelight.org/hdcodetxt/nlz.c.txt`

6 gmplib Implmentation

gmplib's implementation uses the approach of section 5.2, i.e. counting leading zeroes[7]. However it doesn't invent its own wheels, but uses gnulib's count_leading_zeroes function[8]. The gnulib function, in turn, checks for the gcc builtin (as in section 4), and uses it if available; if not, it resorts to counting 32 bits at a time using the idea of section 3 (32bit is a cutoff), and resorts to the cutoff function count_leading_zeros_32 for dealing with 32bit numbers.

For sizes greater than 32 bits (code taken from their macro `COUNT_LEADING_ZEROS(BUILTIN, MSC_BUILTIN, TYPE)`):

```
do                                                                      \
  {                                                                     \
    int count;
                                                                        \
    unsigned int leading_32;                                            \
    if (! x)                                                            \
      return CHAR_BIT * sizeof x;                                       \
    for (count = 0;                                                     \
         (leading_32 = ((x >> (sizeof (TYPE) * CHAR_BIT - 32))          \
                        & 0xffffffffU),                                 \
          count < CHAR_BIT * sizeof x - 32 && !leading_32);             \
         count += 32)                                                   \
      x = x << 31 << 1;                                                 \
    return count + count_leading_zeros_32 (leading_32);                 \
  }                                                                     \
while (0)
```

For 32bit numbers, gnulib simply uses the Stanford Graphics de Bruijn table bithack (which is probably the fastest way) by Sean Anderson (as illustrated in section 5.1):

```
/* Compute and return the number of leading zeros in X,    where 0 < X < 2**32.  */
COUNT_LEADING_ZEROS_INLINE int count_leading_zeros_32 (unsigned int x) {
  /* http://graphics.stanford.edu/~seander/bithacks.html */

    static const char de_Bruijn_lookup[32] = {
     31, 22, 30, 21, 18, 10, 29, 2, 20, 17, 15, 13, 9, 6, 28, 1,
     23, 19, 11, 3, 16, 14, 7, 24, 12, 4, 8, 25, 5, 26, 27, 0
    };
     x |= x >> 1;
     x |= x >> 2;
     x |= x >> 4;
     x |= x >> 8;
     x |= x >> 16;
     return de_Bruijn_lookup[((x * 0x07c4acddU) & 0xffffffffU) >> 27];
```

[7] https://fossies.org/dox/gmp-6.1.2/gmp-impl_8h.html#a4e73eff31639a8c6e6150c5a708694bd
[8] https://github.com/gagern/gnulib/blob/master/lib/count-leading-zeros.h

```
}
```

So again, the gnulib implementation is pretty close to perfect (if not; 64bit CPUs now have 64bit built-in population counts).

7 Conclusion

We have detailed some methods of computing the integer log2 function. The goal is to prevent further use of

```
#include<math.h>
(int)log(x)/log(2);
```

(Just kidding :P).