Computing Trailing Zeros HOWTO

Philip Busch <philip@0xe3.com>

Revision History

Revision v0.9	21 February 2009	pb
Initial release.		
Revision v1.0	1 March 2009	pb
added chapters 2.3 (Bit shifts), 3.2 (An Example) and 4 (Further Reading); added minor improvements		

Abstract

Given the binary representation of some integer, a typical problem is to determine the number of trailing zeros. This HOWTO will show you how to accomplish the task in constant time. The reader is assumed to have experience with the C programming language, though the methods presented throughout this document can easily be applied to other languages as well.

Table of Contents

1. Introduction

You may safely skip this chapter if you are already familiar with HOWTOs or just hate to read all this programming-unrelated stuff.

1.1. Copyright and License

1.2. Disclaimer

Citing from the README of a mathematical subroutine package by R. Freund:

*For all intent and purpose, any description of what the codes are doing should be construed as being a note of what we thought the codes did on our machine on a particular Tuesday of last year. If you're really lucky, they might do the same for you someday. Then again, do you really feel *that* lucky?*

1.3. Credits / Contributors

I would like to thank the following people for their valuable input:

- Malcolm Phillips (New Zealand)

1.4. Feedback

Feedback is most certainly welcome for this document. Send your additions, comments and criticisms to `<philip@0xe3.com>`.

2. Background Knowledge

Computing the number of trailing zeros in constant time is sweet and simple, but in order to understand the trick, we will have to look at some math first. This may sound scary, but all we have to do is understand the notion of a so called de Bruijn sequence. Once we know about de Bruijn sequences, we'll have a short reminder of Two's Complement, and off we go...

2.1. de Bruijn sequences

As we are used to do from math classes, let's start with a definition:

Definition: A de Bruijn sequence *of length $n = 2^k$ is a bitstring consisting of n digits such that every 0-1-sequence of length k occurs exactly once.*

What? Let's have an example: a binary number of length $8 = 2^3$ is a de Bruijn sequence, if it contains every possible bitstring of length 3, i.e. 000, 001, 010, 011, and so on. If you try to find such a de Bruijn sequence, you'll notice that our 8 digit binary has only room for 6 bitstrings of length 3, even if the bitstrings are allowed to overlap. So if you reach the end of a de Bruijn sequence, simply restart from the beginning. That's why they are called *sequences* and not just *numbers*.

Have you already found some de Bruijn sequence of length 8 (i.e., 2^3)? Of course you have. Here's another one: 00010111. Don't believe me? I'll show you:

```
00010111
000           0
 001          1
  010         2
   101        5
    011       3
     111      7
      110     6
       100    4
```

Look at how the bitstrings overlap and wrap around at the end. If you match them together, you'll get our original 00010111, or as we're used to call it: 23.

Now, in order to compute the number of trailing zeros of a binary, we need a de Bruijn sequence of exactly the size of our binary, i.e. CHAR_BITS*sizeof(int). The value of this expression may vary from system to system, and who says that we shall restrict ourselves to `int`s anyway, but for the purpose of this HOWTO, let's simply assume that this is 32. So how do we find a de Bruijn sequence of length 32? After some scribbling on the cover of "Math in 21 days", I ended up with 0x077cb531, which is 00000111 01111100 10110101 00110001 in binary representation. As specified above, its length is exactly $32=2^5$, so this time it even contains all possible 5-tuples.

Just for fun, let's write down the offset for each 5-tuple in the sequence:

Table 1. Sample Table

5-tuple	decimal value	offset
`00000`	00	0
`00001`	01	1
`00010`	02	28
`00011`	03	2
`00100`	04	29
`00101`	05	14
`00110`	06	24
`00111`	07	3
`01000`	08	30
`01001`	09	22
`01010`	10	20
`01011`	11	15
`01100`	12	25
`01101`	13	17
`01110`	14	4
`01111`	15	8
`10000`	16	31
`10001`	17	27
`10010`	18	13
`10011`	19	23
`10100`	20	21
`10101`	21	19
`10110`	22	16
`10111`	23	7
`11000`	24	26
`11001`	25	12
`11010`	26	18
`11011`	27	6
`11100`	28	11
`11101`	29	5
`11110`	30	10
`11111`	31	9

The table tells us that e.g. the 5-tuple `11010` (decimal 26) appears at the 18th position in the binary representation of our de Bruijn sequence. We will need this table later, but let's forget about math for now and learn some fundamental facts about our computers.

2.2. Two's Complement

As we're programmers, I assume that you are already familiar with Two's Complement (if not, see http://en.wikipedia.org/wiki/Two%27s_complement). Actually, I just want to show you a nice trick. This would probably have been a better name for this section, but frankly speaking, it doesn't sound as important.

So. Have you ever wondered how to set all bits of an `int` to zero, except for the last 1 that appears in its binary representation? Well, neither did I, but we need it in order to compute the number of trailing zeros. But before I tell you how to do it, let's first play around a bit. Remember how to change the sign in Two's Complement? We have to do bitwise negation and add 1. Let's do a simple example: changing the sign of 12.

```
00001100       | that's a binary 12
11110011       | bitwise negation
11110100       | add 1, get -12
```

Start with -12 and repeat this procedure, and you'll end up with 12 again. Astonishing, isn't it? Now look at the binary representations of 12 and -12 respectively. You'll notice that they have the same number of trailing zeros. Aside from that, they differ in every digit, except for the last 1. And what's more important, this is also true for every other Two's Complement number [1].

Now it's pretty clear how to isolate the last 1. Simply use bitwise AND on an `int x` and its bitwise negation:

```
x & -x
```

Too simple[2]. The result is a number whose binary representation consists of all zeros, except for the last 1 in both `x` and `-x`, so for `x=12`, the result is a binary `00000100`.

2.3. Bit shifts

In school, one of the most delightful arithmetic operations is multiplication by ten, because there's no real work to do: simply append a zero and that's it. Same goes for multiplication by 100, 1000, and every other power of ten with a natural exponent. But astonishingly, this doesn't have to do anything with 10^n ending with zeros, but with the fact that 10 is the base of our numeral system. Multiplying by 7 in base 7 works exactly the same way: append a zero. More generally, multiplying by `b` in base `b` works by appending a zero. If you don't already believe it, have a short look at how numeral systems are used to represent numbers. For the purpose of this HOWTO, it completely suffices to know that for binaries, multiplication by 2^n is really the same as left-shifting by `n`. Actually, this is how left-shifting is formally defined:

$$x \ll n \;:=\; x * 2^n$$

Of course, no sane chip designer will use multiplication in order to implement bit shifts on a new CPU, but this trick will help us determine the number of trailing zeros in a binary. Now, we are finally ready to have a look at the actual algorithm.

[1]Except `INT_MIN`, because `abs(INT_MIN) == INT_MIN`.
[2]Quiz question: how do you isolate the *first* 1 in constant time?

3. Ready, Steady, Go!

We know some simple facts about Two's Complement arithmetic, and we are mere experts on de Bruijn sequences, so let's see what we can do with it.

3.1. The algorithm

The algorithm for determining the number of trailing zeros of a binary number `x` works as follows. First, we create an array which tells us for each possible bitstring of length 5 its offset in our de Bruijn sequence `DEBRUIJN`. That is, for a given 5-tuple t, how many times do we need to shift `DEBRUIJN` to the left in order to have t appear at the front. Then we set all bits of `x` to zero, except for the last 1. Let's call this number `n`. We multiply `n` with `DEBRUIJN`, but remember: the digit 1 appears exactly once in the binary representation of `n` (i.e. `n` is a power of two), so multiplying `DEBRUIJN` with `n` is really the same as shifting `DEBRUIJN` to the left as many times as there are trailing zeros in `n`. Now we take the first 5 digits of the shifted de Bruijn sequence and look it up in our array. This will tell us how many times we actually shifted `DEBRUIJN` to the left, i.e. the number of trailing zeros of `n`, which is the same as the number of trailing zeros of `x`, which is the solution to our original problem.

Here's the algorithm again:

1. Create array `table` for `DEBRUIJN`
2. `n = x & -x`
3. `shifted = DEBRUIJN * n`
4. `offset = table[shifted]`
5. return `offset`

3.2. An Example

Looks easy enough, but let's have an example anyway: suppose we want to determine the number of trailing zeros in the binary representation of `x = 26,784`, which is `00000000 00000000 01101000 10100000`. Let's have a look at the de Bruijn sequence:

DEBRUIJN = 00000*111 01*111100 10110101 00110001

Note that I emphasized the substring `11101`. Now, let's first first isolate the last 1: `n = x & -x`. The binary representation of `n` is `00000000 00000000 00000000 00100000`. We shift the de Bruijn sequence by the number of trailing zeros in `n`, i.e. multiply `DEBRUIJN` with `n`: `shifted = DEBRUIJN * n`.

shifted = *11101*111 10010110 10100110 00100000

We immediately right-shift the result by 27:

shifted = 00000000 00000000 00000000 000*11101*

Remember our table? It tells us for each bitstring of length 5 its offset in the de Bruijn sequence. For `11101`, it correctly reports 5, which is indeed the number of trailing zeros in `x`.

I'd love to claim that you don't need to understand this, but you need to. If it isn't already clear to you, read this chapter again. Maybe it helps to have a look at the previous chapter, too. Use a sheet of paper and try to reproduce the examples yourself. It's easy, go ahead! In the meantime, I'll continue with the actual source code.

3.3. Populating the table

As mentioned in the previous section, we need an array which tells us for every possible binary 5-tuple the corresponding offset in a given de Bruijn sequence. Of course we might simply hardcode the array, and if you're happy with this solution, you can safely skip the current section and proceed to the next one without any further ado. For aesthetical completeness, this is what we have to do in order to populate the array `table` on the basis of a given de Bruijn sequence `debruijn`: we iterate over `i = 0..31` and for each step, we left-shift `debruijn` by `i` and immediately right-shift back by 27. This way, we retrieve all 5-tuples from our de Bruijn number, i.e. all numbers between 0 and 31, and use them as array subscripts to store the current value of `i` in `table`. When finished, `table` tells us the offset of each 5-tuple in `debruijn`.

Here's the code:

```
void init(int table[32], unsigned int debruijn)
{
        int i;

        for(i = 0; i < 32; i++) {
                table[debruijn >> 27] = i;
                debruijn = debruijn << 1;
        }
}
```

3.4. Looking up the number of trailing zeros

Computing the number of trailing zeros for a given `int x` is now a mere child's play. Set all bits of `x` to zero, except for the last 1. Multiply the result by `debruijn` and right-shift by 27. Then use this value to look up the number of trailing zeros in the `table`.

```
int ntz(int x)
{
 size_t i = ((x & -x) * debruijn) >> 27;
 return table[i];
}
```

Here, `debruijn` is a global variable, of course. If you don't like global variables, consider the function above to be pseudocode.

3.5. Putting it all together

The following program is a complete implementation of the algorithm described above.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define DEBRUIJN 0x077CB531UL

static int table[32];
static unsigned int debruijn;

static void init(int table[32], unsigned int db);
int ntz(int x);

int main(int argc, char *argv[])
{
        int x;

        if(argc != 2) {
                fprintf(stderr, "USAGE: %s <num>\n", argv[0]);
                return EXIT_FAILURE;
        }

        x = atoi(argv[1]);

        /* initialize de Bruijn table */
        init(table, DEBRUIJN);

        printf("number of trailing zeros: %d\n", ntz(x));

        return EXIT_SUCCESS;
}


/* compute number of trailing zeros */
int ntz(int x)
{
        size_t i = ((x & -x) * debruijn) >> 27;
        return table[i];
}


/* initialize table with de Bruijn sequence db */
static void init(int table[32], unsigned int db)
{
        int i;

        debruijn = db;

        for(i = 0; i < 32; i++) {
                table[db >> 27] = i;
                db = db << 1;
        }
}
```

In a real program, you probably wouldn't want to initialize the table at runtime, and you don't even need the `ntz()` function call. So, for the sake of completeness, here's another version which I have shamelessly taken over from Sean Eron Anderson. This version is highly tuned for efficiency, but at first sight it looks like voodoo. If you are bored at work, hand it out to your colleagues and see how long it will take them to figure out what it does:

```
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[((v & -v) * 0x077CB531UL) >> 27];
```

3.6. Caveats

Now you may think: "This was soo easy... now where's the catch?". And you are right, there are some aspects to be aware of when you implement the algorithm.

Most notably, not all de Bruijn sequences are equally suitable (aside from the fact that they need to be of proper size). In the beginning, I told you that we wrap to the start of the sequence once we've reached its end. But in the implementation, we only do left shifting, i.e. fill in with zeros. But that would yield widely inaccurate results if the de Bruijn sequence of length 2^k doesn't start with k zeros, so make sure it does or simply stick to the one I supplied.

Furthermore, the implementation presented here makes possibly inaccurate assumptions about the number of bits in an `int`. Remember the 10th commandment from His prophet Henry Spencer: *Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that ``All the world's a VAX'', and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short*. I sacrificied this wisdom for the sake of readability, but in order to avoid surprises, you better replace every occurrence of the number 32 (and 31 and 27 accordingly) with `CHAR_BITS*sizeof(int)`, which fortunately happens to be a compile-time constant.

On a sidenote, it has been suggested to add something along the lines of `if(x%2) return 0;` at the beginning of `ntz()`, based on the observation that odd numbers don't have trailing zeros in their binary representation. Separate from issues concerning branch prediction, this doesn't change anything in the average case: for odd numbers, you save two machine instructions, but for even numbers, you get two more.

4. Further Reading

This HOWTO is by no means an exhaustive analysis. It serves as an informal introduction to the topic in general. A more formal discussion by Leiserson, Prokop and Randall is available in their paper Using de Bruijn Sequences to Index 1 in a Computer Word [http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.8562]. If you're generally interested in bit manipulation techniques, have a look at Sean Eron Anderson's excellent page about Bit Twiddling Hacks [http://www-graphics.stanford.edu/~seander/bithacks.html].

5. Epilogue

I provided evidence to the assumption that there occasionally exist simple solutions to problems that render impossible at first glance. I hope that you enjoyed reading this HOWTO and probably remember this insight the next time you're about to implement the first approach that comes to your mind. Maybe you now have comments, suggestions, questions or rants. Either way, I'd be glad to hear from you.