Fast Bit Counting

August 5th, 2008

Puzzle: Devise a fast algorithm for computing the number of 1-bits in an unsigned integer. If the machine supports n-bit integers, can we compute the number of 1-bits in O(log n) machine instructions? Can we compute the number of 1-bits in O(b) machine instructions where b is the number of 1-bits in the integer?

Source: Commonly asked in job interviews for Software Engineers. I first heard it in 1996 when I was a graduate student.

Solution: This article presents six solutions to this problem. ~~Source code in C~~ is available.

1. Iterated Count

```
int bitcount (unsigned int n) {
   int count = 0;
   while (n) {
      count += n &amp; 0x1u;
      n &gt;&gt;= 1;
   }
   return count;
}
```

Iterated Count runs in time proportional to the total number of bits. It simply loops through all the bits, terminating slightly earlier because of the while condition. Useful if 1's are sparse and among the least significant bits.

2. Sparse Ones

```
int bitcount (unsigned int n)  {
   int count = 0 ;
   while (n)  {
      count++ ;
      n &amp;= (n - 1) ;
   }
   return count ;
}
```

Sparse Ones runs in time proportional to the number of 1 bits. The mystical line n &= (n – 1) simply sets the rightmost 1 bit in n to 0.

3. Dense Ones

```
int bitcount (unsigned int n)   {
   int count = 8 * sizeof(int) ;
   n ^= (unsigned int) - 1 ;
   while (n)  {
      count-- ;
      n &amp;= (n - 1) ;
   }
   return count ;
}
```

Dense Ones runs in time proportional to the number of 0 bits. It is the same as Sparse Ones, except that it first toggles all bits (n ~= -1), and continually subtracts the number of 1 bits from sizeof(int).

Sparse Ones and Dense Ones were first described by Peter Wegner in “~~A Technique for Counting Ones in a Binary Computer~~“, Communications of the ACM, Volume 3 (1960) Number 5, page 322.

4a. Precompute-8bit

```
static int bits_in_char [256] ;

int bitcount (unsigned int n)  {
  // works only for 32-bit ints

  return bits_in_char [n  &amp; 0xffu]
      +  bits_in_char [(n &gt;&gt;  8 ) &amp; 0xffu]
      +  bits_in_char [(n &gt;&gt; 16) &amp; 0xffu]
      +  bits_in_char [(n &gt;&gt; 24) &amp; 0xffu] ;
}
```

Precompute_8bit assumes an array bits_in_char such that bits_in_char[i] contains the number of 1 bits in the binary representation for i. It repeatedly updates count by masking out the last eight bits in n, and indexing into bits_in_char.

4b. Precompute-16bit

```
static char bits_in_16bits [0x1u &lt;&lt; 16] ;

int bitcount (unsigned int n)  {
   // works only for 32-bit ints

   return bits_in_16bits [n          &amp; 0xffffu]
       +  bits_in_16bits [(n &gt;&gt; 16) &amp; 0xffffu] ;
}
```

Precompute_16bit is a variant of Precompute_8bit in that an array bits_in_16bits[] stores the number of 1 bits in successive 16 bit numbers (shorts).

5. Parallel Count

```
#define TWO(c)     (0x1u &lt;&lt; (c))
#define MASK(c) \
  (((unsigned int)(-1)) / (TWO(TWO(c)) + 1u))
#define COUNT(x,c) \
  ((x) &amp; MASK(c)) + (((x) &gt;&gt; (TWO(c))) &amp; MASK(c))

int bitcount (unsigned int n)  {
   n = COUNT(n, 0) ;
   n = COUNT(n, 1) ;
   n = COUNT(n, 2) ;
   n = COUNT(n, 3) ;
   n = COUNT(n, 4) ;
   /* n = COUNT(n, 5) ;    for 64-bit integers */
   return n ;
}
```

Parallel Count carries out bit counting in a parallel fashion. Consider n after the first line has finished executing. Imagine splitting n into pairs of bits. Each pair contains the *number of ones* in those two bit positions in the original n. After the second line has finished executing, each nibble contains the *number of ones* in those four bits positions in the original n. Continuing this for five iterations, the 64 bits contain the number of ones among these sixty-four bit positions in the original n. That is what we wanted to compute.

6. Nifty Parallel Count

```
#define MASK_01010101 (((unsigned int)(-1))/3)
#define MASK_00110011 (((unsigned int)(-1))/5)
#define MASK_00001111 (((unsigned int)(-1))/17)
 int bitcount (unsigned int n) {
   n = (n &amp; MASK_01010101) + ((n &gt;&gt; 1) &amp; MASK_01010101) ;
   n = (n &amp; MASK_00110011) + ((n &gt;&gt; 2) &amp; MASK_00110011) ;
   n = (n &amp; MASK_00001111) + ((n &gt;&gt; 4) &amp; MASK_00001111) ;
   return n % 255 ;
}
```

Nifty Parallel Count works the same way as Parallel Count for the first three iterations. At the end of the third line (just before the return), each byte of n contains the number of ones in those eight bit positions in the original n. A little thought then explains why the remainder modulo 255 works.

According to Don Knuth (The Art of Computer Programming Vol IV, p 11), in the first textbook on programming, The Preparation of Programs for an Electronic Digital Computer by Wilkes, Wheeler and Gill (1957, reprinted 1984), pages 191–193 presented Nifty Parallel Count by D B Gillies and J C P Miller.

7. MIT HAKMEM Count

```
int bitcount(unsigned int n) {
   /* works for 32-bit numbers only    */
   /* fix last line for 64-bit numbers */

   register unsigned int tmp;

   tmp = n - ((n &gt;&gt; 1) &amp; 033333333333)
           - ((n &gt;&gt; 2) &amp; 011111111111);
   return ((tmp + (tmp &gt;&gt; 3)) &amp; 030707070707) % 63;
}
```

MIT HAKMEM Count is funky. Consider a 3 bit number as being 4a+2b+c. If we shift it right 1 bit, we have 2a+b. Subtracting this from the original gives 2a+b+c. If we right-shift the original 3-bit number by two bits, we get a, and so with another subtraction we have a+b+c, which is the number of bits in the original number. How is this insight employed? The first assignment statement in the routine computes *tmp*. Consider the octal representation of *tmp*. Each digit in the octal representation is simply the number of 1's in the corresponding three bit positions in *n*. The last return statement sums these octal digits to produce the final answer. The key idea is to add adjacent pairs of octal digits together and then compute the remainder modulus 63. This is accomplished by right-shifting *tmp* by three bits, adding it to *tmp* itself and

ANDing with a suitable mask. This yields a number in which groups of six adjacent bits (starting from the LSB) contain the number of 1's among those six positions in *n*. This number modulo 63 yields the final answer. For 64-bit numbers, we would have to add triples of octal digits and use modulus 1023. This is HACKMEM 169, as used in X11 sources. Source: MIT AI Lab memo, late 1970's.

8. Builtin Instructions

GNU compiler allows for

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

which translates into a single CPU instruction if the underlying machine architecture supports it. For example, Intel machines have POPCNT (SSE4 Instruction set announced in 2006). Many GCC builtin functions exist.

Performance Measurements

```
                    No           Some          Heavy
          Optimization Optimization Optimization

Precomp_16 52.94 Mcps   76.22 Mcps   80.58 Mcps
 Precomp_8 29.74 Mcps   49.83 Mcps   51.65 Mcps
  Parallel 19.30 Mcps   36.00 Mcps   38.55 Mcps
       MIT 16.93 Mcps   17.10 Mcps   31.82 Mcps
     Nifty 12.78 Mcps   16.07 Mcps   29.71 Mcps
    Sparse  5.70 Mcps   15.01 Mcps   14.62 Mcps
     Dense  5.30 Mcps   14.11 Mcps   14.56 Mcps
  Iterated  3.60 Mcps    3.84 Mcps    9.24 Mcps

Mcps = Million counts per second
```

Which of the several bit counting routines is the fastest? Results of speed trials on an i686 are summarized in the table on left. “No Optimization” was compiled with plain *gcc*. “Some Optimizations” was *gcc -O3*. “Heavy Optimizations” corresponds to *gcc -O3 -mcpu=i686 -march=i686 -fforce-addr -funroll-loops -frerun-cse-after-loop -frerun-loop-opt -malign-functions=4*.

Thanks to Seth Robertson who suggested performing speed trials by extending ~~bitcount.c~~. Seth also pointed me to MIT_Hackmem routine. Thanks to Denny Gursky who suggested the idea of Precompute_11bit. That would require three sums (11-bit, 11-bit and 10-bit precomputed counts). I then tried Precompute_16bit which turned out to be even faster.

If you have niftier solutions up your sleeves, please send me an e-mail or write comments below!

Further Reading

1. HAKMEM (bit counting is memo number 169), MIT AI Lab, Artificial Intelligence Memo No. 239, February 29, 1972.
2. Bit Twiddling Hacks by Sean Anderson at Stanford University.
3. Bitwise Tricks and Techniques by Don Knuth (The Art of Computer Programming, Part IV).

On 15 March 2009, this article was discussed on Reddit — you might learn quite a bit by reading the comments therein.

---

share share

43 Comments Add new comment.