

Population Counts

Tommy M. McGuire
mcguire@crsr.net

August 5, 2007

1 Introduction

In *Beautiful Code*, Henry S. Warren, Jr., discusses a *population count* or *sideways sum* algorithm, which “calculates the number of bits in a computer word that are 1.”^[2] The problem of finding the number of set bits in a binary value has a number of applications, apparently, although I have never seen it except in answering Google interview questions. (Representing a set of elements as a bit string, with the bit corresponding to an element set to 1 if the element is present in the set, is fairly common. However, I have never had to *count* the number of elements in the set.)

Warren describes several ways of computing the count, from a couple of simple iterations over the bits of the value (in this case, one C int of 32-bits) to a table lookup (see Figure 1), and finally to a couple non-trivial bit-manipulation schemes (see Figure 2). The fourth algorithm is the subject of this note.

The first algorithm should be clear enough, to anyone who is familiar with bit-wise Boolean operations. The second a little less so, since it relies on some C-isms (some simple profiling shows this version to be uninteresting, since it is not terribly faster than the first and is less clear). The third should be tolerably clear as an extension of the first—rather than counting one bit at a time, count n . But the final three are not obviously even related to the problem, and the analysis of why they work as presented by Warren did not help me. To see the beauty of them, I had to go through the steps presented in this note.

2 Divide and conquer: pop4

Parallel algorithms are usually associated with multiple CPUs, threads, and (at least for me) message passing. In this parallel algorithm, however, the operations can be performed on smaller bit-fields of the value simultaneously, in a single-instruction, multiple-data (SIMD) manner.

As a first step, notice that by dividing a 32-bit word into 32 individual-bit fields, each bit field is self-counting. If a bit is set, then that bit field contains 1; if the bit is not set, then that bit field contains 0.

```

unsigned
pop1(unsigned x)
{
< unsigned pop = 0;
  unsigned sz = sizeof(unsigned)*8;
  int i = 0;
  for (i = 0; i < sz; ++i) {
    if (x & 1) ++pop;
    x >>= 1;
  }
  return pop;
}

unsigned
pop2(unsigned x)
{
  unsigned pop = 0;
  while (x) {
    pop += (x & 1);
    x >>= 1;
  }
  return pop;
}

static unsigned char table[16] = {
  0, /* 0 */    1, /* 1 */
  1, /* 2 */    2, /* 3 */
  ...
  2, /* c */    3, /* d */
  3, /* e */    4, /* f */
};
unsigned
pop3(unsigned x)
{
  unsigned sz = sizeof(unsigned)*8;
  unsigned pop = 0;
  int i = 0;
  for (i = 0; i < sz; i += 4) {
    pop += table[(x >> i) & 0xf];
  }
  return pop;
}

```

Figure 1: Three simple bit counts. pop1 and pop2 iterate over each bit in the value, counting those that are set. pop3 uses a four-bit lookup table, a direct and simple optimization of the first method.

```

unsigned
pop4(unsigned 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;
}

unsigned
pop5(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 & 0x3f);
}

unsigned
pop6(unsigned x)
{
  unsigned n;
  n = (x >> 1) & 033333333333;
  x -= n;
  n = (n >> 1) & 033333333333;
  x -= n;
  x = (x + (x >> 3)) & 030707070707;
  return (x % 63);
}

```

Figure 2: Three complex bit counts. pop4 is a divide-and-conquer algorithm, and pop5 is the same algorithm with unnecessary operations removed. pop6 is a 32-bit version of a 3-bit field algorithm from Item 169 of HAKMEM[1].

The second step is to expand the size of the bit fields from one to two, producing a sequence of two-bit fields each containing the number of bits set in the corresponding two-bit field of the original value.

For this analysis, I am going to reduce the number of bits from 32 to 8, and show the original value thus:

$$x = abcdefgh_2$$

Each bit in the value is represented by a letter from a to h , and the subscript 2 indicates that the value is in binary: each letter represents one bit.

Looking at the first two bits of the value, ab , what we want to happen after the first step is for the corresponding bits of the new value to be the number of a and b that are set; 0, 1, or 2. The first statement of pop4 makes this happen:

$abcdefgh_2$	$0abcdefg_2$	x and x right shifted by 1
$\& 01010101_2$	01010101_2	55_{16}
$0b0d0f0h_2$	$+ 0a0c0e0g_2$	

Because of the way binary addition works, the final sum produces the following sequence of two-bit fields:

$$\begin{aligned} x &= {}_2\langle b+a \rangle {}_2\langle d+c \rangle {}_2\langle f+e \rangle {}_2\langle h+g \rangle \\ &= klmnopqr_2 \end{aligned}$$

The notation “ ${}_n\langle b \rangle$ ” indicates a bit field of width n containing b . In the final value, the bits kl equal ${}_2\langle b+a \rangle$, and so on.

The next step of the algorithm should add two sequential two-bit fields to produce a sequence of four-bit fields. The next statement performs this step:

$klmnopqr_2$	$00klmnop_2$	x and x right shifted by 2 bits
$\& 00110011_2$	00110011_2	33_{16}
$00mn00qr_2$	$+ 00kl00op_2$	

The final sum produces the following sequence of four-bit fields:

$$\begin{aligned} x &= {}_4\langle mn+kl \rangle {}_4\langle qr+op \rangle \\ &= {}_4\langle {}_2\langle d+c \rangle + {}_2\langle b+a \rangle \rangle {}_4\langle {}_2\langle h+g \rangle + {}_2\langle f+e \rangle \rangle \\ &= stuvwxyz_2 \end{aligned}$$

The final step of the 8-bit version of the algorithm combines the two four-bit fields into a single eight-bit field containing the total number of 1 bits in the original value.

$stuvwxyz_2$	$0000stuv_2$	x and x right shifted by 4 bits
$\& 00001111_2$	00001111_2	$0f_{16}$
$0000wxyz_2$	$+ 0000stuv_2$	


```
unsigned
revl(unsigned x)
{
  x = ((x >> 1) & 0x55555555) | ((x << 1) & 0xaaaaaaaa);
  x = ((x >> 2) & 0x33333333) | ((x << 2) & 0xcccccccc);
  x = ((x >> 4) & 0x0f0f0f0f) | ((x << 4) & 0xf0f0f0f0);
  x = ((x >> 8) & 0x00ff00ff) | ((x << 8) & 0xff00ff00);
  x = ((x >> 16) & 0x0000ffff) | ((x << 16) & 0xffff0000);
  return x;
}
```

Figure 3: 32-bit bit reversal function.

References

- [1] Michael Beeler, R. William Gosper, and Richard Schroepel. HAKMEM. Technical Report AIM 239, MIT Artificial Intelligence Laboratory, 1972.
- [2] Henry S. Warren, Jr. The quest for an accelerated population count. In *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly and Associates, 2007.