CONTENTS

---

(1) From Julius Goryavsky, 7/25/2003:

Hello,

I apologize for spend your time, however I continued searches of "magic" numbers and improvements for algorithms placed in your book! I have found not only new magic constants and method's modifications for reducing the table size, but I also have found some of performance optimizations for original Robert Harley's algorithm.

If you interests, I can send you the program for the analysis of all possible constants for all modifications of algorithms known to me and search among those constants which have the minimal or maximal numerical value, giving the shortest tables, having the minimal or maximal value of population count.

I offer the following modifications and improvements for Robert Harley's and David Seal's algorithms:

1). I add new transformation step after conversion "x" to "111...111"-like mask:

```
   ...
   x = x | (x >> 8);
   x = x | (x >> 16);
/* New step: */
   x = x ^ (x >> 16);
```

To reduce performance impact, this additional step may be combined with previous one:

```
   ...
   x = x | (x >> 8);
/* x = x | (x >> 16); */
/* x = x ^ (x >> 16); */
   x = x & ~(x >> 16);
```

If the processor has a command "and with complement", then this construction uses the same number of processor cycles and instructions. Using this method we can reduce number of multipliers in the magic constant from four (in original algorithm) to three. We can use a constant:

0xFD7049FF = 511 * 2047 * 16383

This change allows us to reduce number of instructions in algorithm nlz1b and to speed up its work by two instructions (or by one instruction, if the processor do not have the "and with complement" command).

2). If we use Robert Harley's algorithm without replacement of multiplication by shifts and subtractions, then the offered modifications allows us to reduce the size of the auxiliary table from 64 to 48 elements. For this purpose it is necessary to use the magic constant = 0x3EF5D037. Reduction of the size of the table from 64 to 48 elements allows us to saving one line of cache for processors having 16-byte cache lines.

3). Also it is possible to use other modification (similar to first):

```
   ...
   x = x | (x >> 8);
   x = x | (x >> 16);
/* New step: */
   x = ~(x ^ (x >> 16));
```

This additional step may be combined with previous one and simplified by using of De-Morgan's law:

```
   ...
   x = x | (x >> 8);
/* x = x | (x >> 16); */
/* x = ~(x ^ (x >> 16))= ~(x & ~(x >> 16)) */
   x = ~x | (x >> 16);
```

This construction is like previous, but it does not demand additional time, even at one cycle of the processor, because operation "~x" can be executed in parallel with operation "(x > > 16)".

This modification of algorithm allows us to reduce number of factors in the magic constant from four to three if the appropriate constant is used:

   0xF8877FF1 = 15 * 32767 * 0x800001

Interesting feature of this modification is it never produces a table index equal to zero for the given magic constant (and the first element can be excluded from the table, see program "ntz1e").

4). This modification also allows us to reduce the size of the table from 64 to 48 elements if the magic constant = 0x353B8761 is used. Thus the operating time of algorithm is not increased in comparison with original version, even for those processors, which do not have "and with complement" instruction, because operations "~x" and " (x > > 16) " are calculated in parallel. If we use of this constant, then first three elements of the original table (with 64 elements) are not used (also they are reduced by my program used for construction of tables - to minimize the effective size of the table to 48 elements).

5). The previous algorithm has an interesting magic constant 0x02293001 which has population count = 7. Due to this it is possible to replace multiplication by a sequence of 6 shifts and additions:

   0x02293001 = 10 0010 1001 0011 0000 0000 0001 binary:

```
   x = x + (x << 12) + (x << 13) + (x << 16) +
           (x << 19) + (x << 21) + (x << 25);
```

Thus all shifts can be executed in parallel, and six additions can be made in 3 cycles (in pairs) if the processor has infinity parallelism. Other interesting constant is: 0xFAD7BFFF, it may be used in first modification of Harley's algorithm offered by me (x ^ (x > > 16)). It has maximal population count = 27, that gives the following sequence:

```
   0xFAD7BFFF = 1111 1010 1011 0111 1011 1111 1111 1111 binary:
   x = -x - (x << 26) - (x << 24) - (x << 21) -
            (x << 19) - (x << 15);
```

6). If to keep original Robert Harley's and David Seal's algorithms, magic constants minimize the size of the tables for them with 64/63 to 53 elements it 0x2E9BBECD (Harley's) and 0x0432A68B (Seal's).

7). The minimally possible constant for Harley's algorithm is 0x0218A58B which is intended for second modification of the algorithm offered by me. From constants of a kind "sum(2 ^ k/-1)" it is interesting 0x0D322585 = 5 * 127 * 2049 * 131071 * 0x100001. This constant minimizes the size of the table in Seal's algorithm (up to 53 elements), but it consists of five factors instead of three.:(Also it's possible the modification of Seal's algorithm, using expression "-x | x" for which there is smallest of constants that I know: 0x0218A393.

Further included the full code for the modification of algorithms offered by me:

1). Method: "x ^ (x >> 16)", shift/add implementation:

```
int nlz1c(unsigned x) {
   static unsigned char table [64] = {
      32, 20, 19,  u,  u, 18,  u,  7,
      10, 17,  u,  u, 14,  u,  6,  u,
       u,  9,  u, 16,  u,  u,  1, 26,
       u, 13,  u,  u, 24,  5,  u,  u,
       u, 21,  u,  8, 11,  u, 15,  u,
       u,  u,  u,  2, 27,  0, 25,  u,
      22,  u, 12,  u,  u,  3, 28,  u,
      23,  u,  4, 29,  u,  u, 30, 31
   };
   x = x | (x >> 1);    // Propagate leftmost
   x = x | (x >> 2);    // 1-bit to the right.
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x & ~(x >> 16);
/* x = x * 0xFD7049FF; */
   x = (x << 9) - x;
   x = (x << 11) - x;
   x = (x << 14) - x;
   return table[x >> 26];
}
```

2). Method: "x ^ (x >> 16)", multiply-based implementation (table length is 48 elements!):

```
int nlz1d(unsigned x) {
   static unsigned char table [48] = {
      32,  6,  5,  u,  4, 12,  u, 20,
      15,  3, 11,  0,  u, 18, 25, 31,
       8, 14,  2,  u, 10,  u,  u,  u,
       u,  u,  u, 21,  u,  u, 19, 26,
       7,  u, 13,  u, 16,  1, 22, 27,
       9,  u, 17, 23, 28, 24, 29, 30
```

```
   };
   x = x | (x >> 1);    // Propagate leftmost
   x = x | (x >> 2);    // 1-bit to the right.
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x & ~(x >> 16);
   x = x * 0x3EF5D037;
   return table[x >> 26];
}
```

3). Method: "~(x ^ (x >> 16))", shift/add implementation:

```
int nlz1e(unsigned x) {
   unsigned char table [62] = {
      32,  u, 31, 19,  u, 11, 30, 18,
       u,  u, 10,  u,  7, 29,  u, 17,
       1,  u,  u,  9,  u,  u,  u,  6,
       u,  u,  4,  u, 28, 24,  u, 16,
       0, 20,  u, 12,  u,  u,  8,  u,
       2,  u,  u,  u,  u,  5, 25,  u,
      21,  u, 13,  u,  3,  u, 26, 22,
       u, 14, 27, 23,  u, 15
   };
   x = x | (x >> 1); // Propagate leftmost
   x = x | (x >> 2); // 1-bit to the right.
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = ~x | (x >> 16);
/* x = x * 0xF8877FF1; */
   x = (x << 4) - x;
   x = (x << 15) - x;
   x = (x << 23) + x;
   return (table - 1)[x >> 26]; // Zero index is impossible!
}
```

4). Method: "~(x ^ (x >> 16))", multiply-based implementation (table length is 48 elements!):

```
int nlz1f(unsigned x) {
   unsigned char table [48] = {
       1, 22,  u,  3, 18, 21, 15, 30,
       u,  u,  6, 13, 17, 10, 20,  u,
       u,  0, 29, 27,  u, 25,  u,  u,
       2,  4,  u, 16,  7, 14, 11, 23,
      19,  u, 31,  u,  5,  u,  8, 12,
      28, 26,  u,  9,  u,  u, 24, 32
   };
   x = x | (x >> 1); // Propagate leftmost
   x = x | (x >> 2); // 1-bit to the right.
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = ~x | (x >> 16);
   x = x * 0x353B8761;
   return (table - 3)[x >> 26]; // Index in range [0...2] is impossible!
}
```

5). Method: "~(x ^ (x >> 16))", using magic constants with population count = 7 and 27:

```
int nlz1g(unsigned x) {
   unsigned char table [64] = {
       3,  u,  2,  u,  9,  1,  u,  u,
      12,  8,  u,  0,  u,  u,  u,  5,
      11, 14,  u,  7,  u,  u, 22,  u,
       u, 17,  u, 20,  u, 26,  u,  u,
```

```
        4, 10,  u, 13,  u,  u,  u,  6,
       15,  u,  u, 23, 18, 21, 27,  u,
        u,  u,  u, 16,  u, 24, 19, 28,
        u,  u, 25, 29,  u, 30, 31, 32
    };
    x = x | (x >> 1); // Propagate leftmost
    x = x | (x >> 2); // 1-bit to the right.
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = ~x | (x >> 16);
/*  x = x * 0x02293001; */
    x = x + (x << 12) + (x << 13) + (x << 16) +
            (x << 19) + (x << 21) + (x << 25);
    return table[x >> 26];
}

int nlz1h(unsigned x) {
    unsigned char table [64] = {
       32,  u,  u,  u,  u, 14,  u,  u,
        u,  8, 13,  u,  u,  u,  5,  u,
        0,  u, 10,  7, 12,  u,  u,  u,
       27, 22,  u,  u, 25,  4,  u,  u,
       20,  u, 15,  u,  9,  u,  u,  6,
        1, 11,  u,  u, 28, 23, 26,  u,
       21, 16,  u,  u,  2,  u, 29, 24,
        u, 17,  3,  u, 30, 18, 31, 19
    };
    x = x | (x >> 1);    // Propagate leftmost
    x = x | (x >> 2);    // 1-bit to the right.
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x & ~(x >> 16);
/*  x = x * 0xFAD7BFFF; */
    x = -x - (x << 26) - (x << 24) - (x << 21) -
             (x << 19) - (x << 15);
    return table[x >> 26];
}
```

6). Constants for minimization of the size of tables up to 53 elements (using the original algorithms):

Robert Harley's algorithm, magic constant 0x2E9BBECD:

```
unsigned char table [53] = {
    32,  6, 23,  u,  2,  u,  u, 25,
     u,  u,  u, 31,  u,  9,  5,  u,
    22, 29, 19, 15,  1,  u,  u,  u,
     u,  u,  7, 24,  3,  u, 26,  u,
     u, 10, 30, 20, 16,  u,  u,  8,
     4, 27, 11, 21, 17,  u, 28, 12,
    18, 13,  u, 14,  0
};
```

David Seal's algorithm, magic constant 0x0432A68B:

```
unsigned char table [53] = {
    32,  0,  1,  6,  2, 25,  7,  u,
     3,  u, 12, 26,  8, 19,  u,  u,
     4, 23,  u, 17, 15, 13, 27,  u,
    29,  9, 20,  u,  u,  u,  u,  u,
    31,  5, 24,  u,  u, 11, 18,  u,
    22, 16, 14,  u, 28,  u,  u,  u,
    30,  u, 10,  u, 21
};
```

Sincerely, Julius Goryavsky.

---

(2) From Doug Moore, 5/6/2003:

If you're interested, I have a few bit twiddling items at a web site
that persists at my former employer:
http://www.caam.rice.edu/~dougm/twiddle/
Of particular interest, I think:

Two concern integers with a fixed number of bits set:
1. given an integer with k bits set, find the next larger one
2. given an integer with k bits set, find another one that differs from
this one in only two bit positions, and has k bits set; by repeating the
operation, you enumerate the k-bits-set integers.

One concerns Hilbert curves - code in n-dimensions, and for comparison
and range queries, not just conversions from int to point and back. I'd
have to do some work to make it understandable though.

Related to that is some work on transposing rectangular bit matrices.
It's an important part of n-dimensional Hilbert encoding.

One not on the web page that has interested me is the representation of
rational numbers by their position in the Sturm-Brochot tree - it's not
a different base for numbers, but a different binary representation that
includes rationals. Very briefly:

if x is a number, then
1x = 1+x, and
0x = x/(1+x)

so 0101 = 2/5.

There's a HAKMEM item about this representation, though not focusing on
a bit-representation.

I'm looking forward to reading the book more carefully.

Doug Moore

---

(3) From Dean Gaudet, 10/23/2003:

here's yet another way of counting trailing zeros which i don't think i
saw on your pages / in your book.

```
int ntz(unsigned src)
{
                unsigned x;
                unsigned bit4, bit3, bit2, bit1, bit0;

                if (src == 0) return 32;
                x = src & -src;
                bit4 = (src & 0xffff) ? 0 : 16;
                bit3 = (x & 0xff00ff00) ? 8 : 0;
                bit2 = (x & 0xf0f0f0f0) ? 4 : 0;
                bit1 = (x & 0xcccccccc) ? 2 : 0;
                bit0 = (x & 0xaaaaaaaa) ? 1 : 0;

                return bit4 + bit3 + bit2 + bit1 + bit0;
}
```

```
the bit4 test is deliberately different because it can go in parallel with
the (src & -src) calculation.

notice how all those ?: tests are independent of each other.  if the
target machine is really wide they could go entirely in parallel.

it looks like it'd work well on your "full risc" model using movne for the
?:.

or cmpne could set bitN to 0/1 and ins could be used in place of all the
additions.

those 32-bit immediates could be replaced with 16-bit immediates if x>>16
is used when bit4 != 0.

take care
-dean

------------------------------------------------------------------------

(4) From Falk Hueffner, 1/23/2004:

Okay. Here's what I have [on "multibyte," or SIMD arithmetic
operations]. It seems this method is worthwhile for add/sub, I already
have a patch for gcc to do this for the new autovectorization framework
(http://gcc.gnu.org/projects/tree-ssa/vectorization.html), but
probably not for saturated sub/add, unless somebody can come up with a
better method. "v8qi" is gcc speak for a vector of 4 "quarter ints",
i.e., bytes.

typedef unsigned long long uint64_t;

// 6 insns, 3 cycles (without constant construction)
inline uint64_t addv8qi(uint64_t x, uint64_t y) {
    uint64_t signmask = 0x8080808080808080ULL;
    uint64_t signs = (x ^ y) & signmask;
    x &= ~signmask;
    y &= ~signmask;
    x += y;
    x ^= signs;
    return x;
}

// 6 insns, 3 cycles (without constant construction)
inline uint64_t subv8qi(uint64_t x, uint64_t y) {
    uint64_t signmask = 0x8080808080808080ULL;
    uint64_t signs = (x ^ ~y) & signmask;
    x |=  signmask;
    y &= ~signmask;
    x -= y;
    return x ^ signs;
}

// 4 insns, 3 cycles
inline uint64_t negv8qi(uint64_t y) {
    return subv8qi(0, y);
}

// unsigned saturated add, saturate to 0xff
inline uint64_t usaddv8qi(uint64_t x, uint64_t y) {
    uint64_t t0, t1;
    uint64_t signmask = 0x8080808080808080ULL;

    t0 = (y ^ x) & signmask;
    t1 = (y & x) & signmask;
```

```
    x &= ~signmask;
    y &= ~signmask;
    x += y;

    t1 |= t0 & x;
    t1 = (t1 << 1) - (t1 >> 7);
    return (x ^ t0) | t1;
}

// unsigned saturated sub, saturate to 0x00 or 0xff
// 14 insns, 7 cycles (without constant construction)
inline uint64_t ussubv8qi(uint64_t x, uint64_t y) {
    uint64_t t0, t1;
    uint64_t signmask = 0x8080808080808080ULL;

    t0 = (y ^ ~x) & signmask;
    t1 = (y & ~x) & signmask;

    x |=  signmask;
    y &= ~signmask;
    x -= y;

    t1 |= t0 & ~x;
    t1 = (t1 << 1) - (t1 >> 7);
    return (x ^ t0) & ~t1;
}

// 1 insn, 1 cycle
inline uint64_t usnegv8qi(uint64_t x) {
    return 0;
}

// signed saturated add, saturate to 0x80 or 0x7f
// 16 insns, 8 cycles
uint64_t ssaddv8qi(uint64_t x, uint64_t y)
{
    uint64_t eq, xv, yv, satmask, satbits, satadd, t0, t1;
    uint64_t signmask = 0x8080808080808080ULL;

    eq = (x ^ ~y) & signmask;
    xv = x & ~signmask;
    yv = y & ~signmask;
    xv += yv;
    satbits = (xv ^ y) & eq;
    satadd = satbits >> 7;
    satmask = (satbits << 1) - satadd;
    xv ^= eq;
    t0 = (xv & ~satmask) ^ signmask;
    t1 = satadd & ~(xv >> 7);
    return t0 - t1;
}

// Average. This is valuable for MPEG codecs.
inline uint64_t avgv8qi(uint64_t a, uint64_t b) {
    return (a & b) + (((a ^ b) & 0xfefefefefefefefeULL) >> 1);
}

// Rounded average.
inline uint64_t avg_roundv8qi(uint64_t a, uint64_t b)
{
    return (a | b) - (((a ^ b) & 0xfefefefefefefefeULL) >> 1);
}

void dumpsb(uint64_t v) {
```

```
    signed char c[8];
    memcpy(c, &v, 8);
    printf("%016lx %4d %4d %4d %4d %4d %4d %4d %4d\n", v,
                    c[7], c[6], c[5], c[4], c[3], c[2], c[1], c[0]);
}

// some example
int main() {
    uint64_t x =  0x01ff807fff897908;
    uint64_t y =  0x08010712fe8870ff;
    dumpsb(x);
    dumpsb(y);
    dumpsb(ssaddv8qi(x, y));
}
```

As I mentioned, the ideas are not from me but mostly stolen somewhere.
I've also not thoroughly tested them...

--
Falk

---

(5) From Paul Curtis, 12/15/2004:

Hi,

P. 92 of Hacker's Delight indicates that after the second assignment to y, you can do a jump-if-zero to determine if there's any zero byte in a word.

But... If that's what you want to do, then why not use:

(fz == find-if-zero)

```
int fz(int x)
{
  return ((x-0x01010101) ^ x) & ~x & 0x80808080;
}
```

(X-0x01010101) ^ x indicates whether there was a change of sign in bit 7 of each byte of the word after subtraction. Anding with ~x separates the 80-1=0x7f and 00-=0xff case. Anding with 80 gives us each bit we're after.

But with De Morgan, bit twiddling can be reduced somewhat.

Assume v = x-0x01010101, then:

```
(v ^ x) & ~x =>
  (v & ~x | ~v & x) & ~x =>
    (v & ~x & ~x | ~v & x & ~x) =>
      (v & ~x | 0) =>
         (v & ~x)
```

```
int fz(int x)
{
  return (x-0x01010101) & ~x & 0x80808080;
}
```

If this returns a non-zero result, then there's a zero in the word. It looks like it delivers along the same as the first two lines of Figure 6-3.

This is a significant result. It can accelerate many zero-terminated

string functions if the inputs are word aligned...

Regards,

--
Paul Curtis, Rowley Associates Ltd http://www.rowley.co.uk
CrossWorks for MSP430, ARM, and now AVR processors

---

(6) From Oleg Khovayko, 3/29/05:

Dear Henry!

Let I introduce myself. My name Oleg Khovayko. Detailed info about myself you can found at: http://olegh.spedia.net (sorry, not updated for some years)

I want to share to you some my old tricks, not reflected in the your book now.

[Below is an efficient way to unroll a loop by a factor of 2, when it is a counting-loop (not a while-loop), and the machine has a way to shift right 1 and then branch on the bit shifted out. First is Intel x86 code, followed by PDP-11 code. Part of the wisdom here is that if the loop is to be executed an odd number of times, then it is best to take care of the odd iteration at the start, rather than at the end, to avoid a compare and branch in the middle of the unrolled loop. The presence of the label $2 will, however, inhibit some optimizations, such as commoning and some scheduling. Here CMD is the body of the loop being unrolled.]

1 ------- unloop code:

```
        mov     count, %cx
        inc     %cx
        shr     %cx             ; Shift right 1, setting carry (CF)
        jnc     $2              ; Jump if no carry.
$1:     CMD
$2:     CMD
        loop    $1              ; Decrement count in %cx & jump if nonzero.
```

[Another loop transformation that might be considered is to change:

```
          do i = 1 to n;
             CMD
          end;
```

to:

```
          do i = 1 to (n & -2) by 2;
             CMD
             i = i + 1;
             CMD
          end;
          if n > 0 && (n & 1) != 0 then do;
             i = n;
             CMD
          end;
```

This puts the two occurrences of CMD in the loop with no labels or branches between them, giving more opportunities for optimizations, particularly scheduling. Of course, there's a code blow-up.]

Same code on the DEC PDP-11 Assembly language (more familiar to me):

```
        mov     #count, r0
        inc     r0
        asr     r0
        bcc     2$
1$:     CMD
2$:     CMD
        sob     r0, 1$
```

[Below is a method for counting the 1-bits in a word by shifting left 1, adding the bit shifted out to an accumulator (r0 below), and continuing until the word being shifted becomes 0. This is PDP-11 code. The bne branches if the register result of the asl (arithmetic shift left) instruction is nonzero.]

2 --- compact code for bit counting in the word:

```
        clr     r0              ; accumulator
        mov     #WORD, r1
1$:     adc     r0              ; Add carry to accumulator
        asl     r1              ; shift left, and put bit 15 to "C"
        bne     1$              ; not zero
        adc     r0
        ; r0 - contains result
```

3 --- Bit counting in a sparse array

For bit counting in the array, approx 4 years ago I've written following procedure (not copied from code, restored from memory):

```
int ar_bit(int *ar, int len) {
  int ac = *ar++, rc = 0;
  while(--len > 0) {
    register x = *ar++;
    register n = x & ac;        // conflict bits
    ac |= x;                    // Or next value to accumulator
    if(n) {                     // was conflict
      rc += wbc(ac); ac = n;
    }
  } // while
  return rc + wbc(ac);
} // ar_bit
```

It is very fast for sparsely populated arrays. We use an "accumulator," where we accumulate (by ORs) bits from the array. We call the wbc (WordBitCount) function only when the accumulator "overflows" (two bits want to go into the same bit position).

In this way, we do not count bits in each word; we count bits only in the "rich" accumulator from time to time.

This code is good for superscalar CPUs, since during regular control flow, the result of the current command uses in "overnext" command only.

---

(7) From a reader, 7/12/05:

[Ed. note: SHA = Secure Hash Algorithm. See http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf]

Looking at section 2-19, I was reminded of some tricks that arise in implementing SHA.

One operation required in SHA (and some monochrome graphics)
is a bit-wise x ? y : z. If your machine has a 1-cycle
and-not, that can be simply implemented as

```
ch(x,y,z) = (x & y) | (~x & z)
```

However, another useful implementation is

```
ch(x,y,z) = z ^ (x & (y ^ z))
```

This is particularly good on 2-operand machines when you want to
leave the original operands around.

The same trick can be used to generate a 2n-bit rotate on a machine
with only n-bit rotates. Using a suitable mask M, you can do
an r-bit rotate using:

```
hi = ROTL(hi, r%n)
lo = ROTL(lo, r%n)
t = (hi ^ lo) & M
hi ^= t
lo ^= t
```

Particularly useful for the 64-bit fixed rotates in SHA-512.
In fact, similar tricks can be used to rotate an n*32-bit word by
up to +/-32 bits. Just rotate each word by the appropriate amount, and
then shuffle the corresponding portions of each word. For example,
a 17-bit left rotate of a 128-bit word, stored little-endian on
a 32-bit machine:

```
                x[0] = ROTL(x[0],17)
                x[1] = ROTL(x[1],17)
                x[2] = ROTL(x[2],17)
                x[3] = ROTL(x[3],17)
```

Followed by either three swap operations:

```
                t = (x[3] ^ x[2]) & 0x1ffff;
                x[3] ^= t;
                x[2] ^= t;
                t = (x[2] ^ x[1]) & 0x1ffff;
                x[2] ^= t;
                x[1] ^= t;
                t = (x[1] ^ x[0]) & 0x1ffff;
                x[1] ^= t;
                x[0] ^= t;
```

Or, more efficiently:

```
                x3 = x[3];
                x[3] ^=   (x3 ^ x[2]) & 0x1ffff;
                x[2] ^= (x[2] ^ x[1]) & 0x1ffff;
                x[1] ^= (x[1] ^ x[0]) & 0x1ffff;
                x[0] ^= (x[0] ^ x3)   & 0x1ffff;
```

Another trick that is useful in SHA is to replace an OR with
an XOR or + when the arguments can be proven to have no bits
in common. With the simple ch() function,

```
ch(x,y,z) = (x & y) | (~x & z)
          = (x & y) ^ (~x & z)
          = (x & y) + (~x & z)
```

is useful, because some rounds of SHA-1 compute:

```
                ROTL(a,5) + ch(b,c,d) + k + w[i]
```

Having ch() broken into two, summed sub-functions gives you
more scheduling flexibility.

The same trick can be applied to the bit-wise majority function,
also used in the SHA family:

```
maj(x,y,z) = (x&y) | (y&z) | (z&x)
           = x&(y|z) | (y&z)
           = x&(y^z) | (y&z)
           = x&(y^z) ^ (y&z)
           = x&(y^z) + (y&z)
```

Again, the latter form helps find parallelism when summing the result.

---

(8) From Norbert Juffa, 10/01/05:

Just a quick note to pass on a couple observations I made with
regard to the population count and bit reversal sections of
"Hacker's Delight".

The first observation is in regard to the HAKMEM version of
population count. The original version is quite slow since
it uses a modulo operation (% 63) to sum the 6-bit fields.
However, the summing can be accelerated significantly by
using an integer multiply. In fact, on my Athlon (which has
a very fast integer multiplier) this puts this version among
the top performers.

```
int popcnt32 (unsigned x)
{
  unsigned s = 033333333333;
  unsigned t = 030707070707;
  unsigned n;
  n = (x >> 1) & s;
  x = x - n;
  n = (n >> 1) & s;
  x = x - n;
  n = x;
  x = x >> 3;
  n = (n + x) & t;
  x = n;
  t = t & (t << 2);  /* t = 0x04104104 */
  x = (n >> 30) + ((x * t) >> 26);
  return x;
}
```

Note that the magic multiplier needed for summing the 6-bit
fields can be derived easily from one of the existing masks.
This is advantageous on processors with poor support for
constructing constants, such as ARM (the ARM can execute
t = t & (t << 2) in a single instruction, on the other hand).

The second observation is with regard to the many masks needed
in the 32-bit bit-reversal. In fact, except for one mask which
needs to be loaded, all these masks can be easily generated on
the fly. To enable this, one has to reverse the order of the
processing steps such that the bigger groups are exchanged
prior to smaller groups. On-the-fly construction of large masks
is very advantageous on the ARM, for example. Having a CPU with
an ANDN instruction certainly helps here, since half the masks
are just inverted versions of the other half.

```
unsigned bitreverse (unsigned x)
{
  unsigned m = 0xffff0000;

  x = ((x & m) >> 16) | ((x & ~m) << 16);   m = m ^ (m >> 8);
  x = ((x & m) >>  8) | ((x & ~m) <<  8);   m = m ^ (m >> 4);
  x = ((x & m) >>  4) | ((x & ~m) <<  4);   m = m ^ (m >> 2);
  x = ((x & m) >>  2) | ((x & ~m) <<  2);   m = m ^ (m >> 1);
  x = ((x & m) >>  1) | ((x & ~m) <<  1);

  return x;
}
```

One could also start with m = 0x0000ffff, if that is more
efficient on some platform. Also, on a 32-bit machine there
is no need to mask in the first step when bit reversing 32-bit
operands. It is sufficient to do:

```
x = (x >> 16) | (x << 16);
```

-- Norbert

---

(9) From Peter Luschny, 6/17/06:

Dear Mr. Warren,

I read with much delight the sample sections of your Hacker's Delight. After hearing
about the much discussed bug in the Java implementation of the binary search

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
http://lambda-the-ultimate.org/node/1549

I considered the question of finding the average of two signed integers (a+b)/2 and
tried to solve the problem with 6 rounding-modes: Up, Down, Ceiling, Floor, Even, Odd.

Maybe my solutions, which I include below, are of interest to you.

With many regards
Peter Luschny

```
// ------------------------------------------------------------------
// Written in C#, code is in the public domain.

// Specifies a rounding behavior for averaging two signed integers.
public enum Rounding { Up, Down, Ceiling, Floor, Even, Odd };

public class AveragesOfTwoSignedIntegers {

// Computes the average of two signed integers according to
// the rounding mode requested. I deliberately used a tiny
// framework for simple comparison.
public static int Average(int a, int b, Rounding mode)
{
     int v = (a >> 1) + (b >> 1);
     bool cond;

     switch (mode)
     {
         // Round the average towards negative infinity.
         case Rounding.Floor: cond = true; break;

         // Round the average towards positive infinity.
```

```
          case Rounding.Ceiling: cond = false; break;

          // Round the average away from zero.
          case Rounding.Up: cond = (a ^ b) < 0; break;

          // Round the average towards zero.
          case Rounding.Down: cond = (a ^ b) > 0; break;

          // Round the average towards the even neighbor.
          case Rounding.Even: cond = ((v & 1) + ((a & b) & 1)) == 0; break;

          // Round the average towards the odd neighbor.
          case Rounding.Odd: cond = ((v & 1) + ((a & b) & 1)) != 0; break;
      }

      return v + ((cond ? a & b : a | b) & 1);
  }

  // --- I run some tests against the following function.

  public static int dAverage(int a, int b, Rounding mode)
  {
      if (a == b) return a;

      double h = a / 2.0 + b / 2.0;
      int r = 0;

      switch (mode)
      {
          case Rounding.Floor: r = (int) Math.Floor(h); break;
          case Rounding.Ceiling: r = (int) Math.Ceiling(h); break;
          case Rounding.Down:  r = ((a < 0 && b < 0) || (a >= 0 && b >= 0)) ?
                        (int) Math.Floor(h) : (int) Math.Ceiling(h); break;
          case Rounding.Up: r = ((a < 0 && b < 0) || (a >= 0 && b >= 0)) ?
                        (int) Math.Ceiling(h) : (int)Math.Floor(h); break;
          case Rounding.Even: r = (int) Math.Floor(h);
                        if ((r & 1) != 0) r = (int) Math.Ceiling(h); break;
          case Rounding.Odd: r = (int) Math.Floor(h);
                        if ((r & 1) != 1) r = (int) Math.Ceiling(h); break;
      }
      return r;
  }
  } // ------------------------------------------------------------------
```

---

(10) From Tim Shaporev, 10/30/06:

Hello!
I just wrote [bought] the "Hacker's Delight" book
(well, Russian edition to be exact)
and I think I have couple of additions
(I found nothing similar on the site from the first glance).

for #2.18 Doz, Min and Max

There is quite effective implementation of min and max functions for
Intel x86 processor (unsigned integers only).
I wrote the sample for 16-bits mode though it works for 32 bits as well.

Arguments are in AX, BX

```
sub ax,bx
sbb dx,dx ; either 0 or all 1's
and ax,dx ; either 0 or a-b
add ax,bx ; min
```

```
The max function is one command longer:
sub ax,bx
sbb dx,dx ; either 0 or all 1's
not dx
and ax,dx ; either 0 or a-b
add ax,bx ; max

These were found in some old disassembled code so I do not know who is
the real author of the trick.

for #5.3 nlz() function.

The function in the book is quite interesting for it demonstrates three
different tricks, but using just two of them gives bit more effective
implementation:

int nlz32(unsigned x)
{
     int m, y;
     int n = 32;

     y = -(int)(x >> 16);        /* test left half of x */
     m = (y >> 16) & 16;         /* either 0 or 16 */
     n -= m;                     /* remember the count */
     x >>= m;                    /* significant 16 bits to lower */

     y = -(int)(x >> 8);
     m = (y >> 8) & 8;                  /* either 0 or 8 */
     n -= m;                     /* remeber the result */
     x >>= m;                    /* significant byte to lower */

     y = -(int)(x >> 4);
     m = (y >> 4) & 4;           /* either 0 or 4 */
     n -= m;                     /* remember the result */
     x >>= m;

     y = -(int)(x >> 2);
     m = (y >> 2) & 2;
     n -= m;
     x >>= m;                    /* x = 0, 1, 2 or 3 */

     m = (int)(x & ~(x >> 1));   /* m = 0, 1 or 2 */
     return n - m;
}

Depending on the processor and/or compiler it could be advantageous to
minimize the number of constants:

int nlz32(unsigned x)
{
     int m, s, y;
     int n = 32;

     s = n >> 1;
     y = -(int)(x >> s);         /* test left half of x */
     m = (y >> s) & s;           /* either 0 or 16 */
     n -= m;                     /* remember the count */
     x >>= m;                    /* significant 16 bits to lower */
     s >>= 1;

     y = -(int)(x >> s);
     m = (y >> s) & s;                  /* either 0 or 8 */
     n -= m;                     /* remeber the result */
     x >>= m;                    /* significant byte to lower */
```

```
    s >>= 1;

    y = -(int)(x >> s);
    m = (y >> s) & s;           /* either 0 or 4 */
    n -= m;                     /* remember the result */
    x >>= m;
    s >>= 1;

    y = -(int)(x >> s);
    m = (y >> s) & s;
    n -= m;
    x >>= m;                    /* x = 0, 1, 2 or 3 */

    m = (int)(x & ~(x >> 1));   /* m = 0, 1 or 2 */
    return n - m;
}
```

Hope this could be helpful.
My apologies for your time if this is known.

Tim

---

(11) From Jasper Neumann, 7/21/2012:

Conversion of BCD coded decimals to binary SIMD-like by Estrin's scheme:

```
uint32 bcd2binary(uint32 x) {
   x=(x & 0x0F0F0F0F) + 10 * ((x >> 4) & 0x0F0F0F0F) ;
   x=(x & 0x00FF00FF) + 100 * ((x >> 8) & 0x00FF00FF) ;
   x=(x & 0x0000FFFF) + 10000 * (x >> 16);
   return x;
}
```

I found this in "Architectural Support for SWAR Text Processing with Parallel Bit Streams: The Inductive Doubling Principle" by Robert D. Cameron and Dan Lin. You can easily find this document via Google.

Furthermore as described on my site programming.sirrida.de there are very simple bit index operations capable of performing any BPC permutation in at most log_2(word_size) operations. IMHO these operations are really missing in your first edition of Hacker's Delight.
[And in the second edition. - HSW]

Best regards
Jasper Neumann

---

(12) From Ulrich Kunitz, 8/13/2012:

Hello,

I observed a way to compute nlz from ntz that I hope you find interesting.

nlz(x) = 32 - ntz(rprop(x) + 1) = 32 - ntz(~rprop(x))

The function rprop propagates the leftmost bit to the right as in the nlz variant using pop (figure 5-11 in your excellent book). In the case of Reiser's algorithm for ntz the value rprop(x) + 1 has already the right form for a direct lookup of the remainder.

Here is C code using Reiser's algorithm applied for 64 bit.

```
#include <stdint.h>

/* reiser_table[(UINT64_C(1) << k) % 67] = k for 0 <= k <= 64 */
static const int8_t reiser_table[67] = {
        64, 0, 1, 39, 2, 15, 40, 23, 3, 12, 16, 59, 41, 19, 24, 54, 4, -1, 13,
        10, 17, 62, 60, 28, 42, 30, 20, 51, 25, 44, 55, 47, 5, 32, -1, 38, 14,
        22, 11, 58, 18, 53, 63, 9, 61, 27, 29, 50, 43, 46, 31, 37, 21, 57, 52,
        8, 26, 49, 45, 36, 56, 7, 48, 35, 6, 34, 33};

int ntz64(uint64_t x) {
        x = x & -x;
        return reiser_table[x % 67];
}

uint64_t rprop64(uint64_t x) {
        x |= x >> 1;
        x |= x >> 2;
        x |= x >> 4;
        x |= x >> 8;
        x |= x >> 16;
        x |= x >> 32;
        return x;
}

int nlz64(uint64_t x) {
        x = rprop64(x) + 1;
        return 64 - reiser_table[x % 67];
}
```

The subtraction in nlz64 can be saved if two tables are used. I decided to share the table to have a higher probability of hot cache lines.

Kind regards,

Ulrich Kunitz

---

(13) From Christian Plesner Hansen, 10/5/2012:

Hi,
I recently wrote a blog post[1] about the approximate reciprocal square root function and someone pointed out to me that this is covered in the revisions for Hacker's Delight. (By the way I'm very much looking forward to a new edition that incorporates all the revisions!) Reading it over I came across a tiny inaccuracy: the original author of the approximate reciprocal square root function is now known, it was apparently created by Greg Walsh[2].

Secondly, you may be interested to know that the approach can be generalized to any power between -1 and 1. The general expression is

```
i = 0x3f7a3bea + p * (i - 0x3f7a3bea);
```

where p is the power. (Obviously this works best if you have a fixed p that allows you to replace the multiplication with a shift, like the square root and reciprocal square root).

[1]: http://blog.quenta.org/2012/09/0x5f3759df.html
[2]: http://www.beyond3d.com/content/articles/15/

---

(14) From Harold Aptroot, 6/17/2013:

Dear Dr Warren,

I believe I may have discovered a more efficient algorithm for propagating bounds through bitwise AND and OR (but not XOR). I'll explain the algorithm I discovered for minOR here, the other three are all different sides of the same coin as it were.

It takes the optimization of setting m = 0x80000000 >> nlz(a ^ c) to the extreme by immediately setting m to the value where it passes the test and breaks out of the loop.

It relies on the observation that the test "temp <= b" can be expressed as a bitmask as well, namely
(a ^ b) == 0 ? 0 : 0xFFFFFFFF >> nlz(a ^ b)
Since a <= b, the highest bit k at which a and b differ must be 0 in a and 1 in b.
(a | (1 << k)) & -(1 << k) must then be less than or equal to b, because it has the same prefix and then continues with only zeroes.

So now both tests can be expressed as bit masks, and another nlz (or bsr) is sufficient to find the highest bit in the intersection of the conditions - the exact m that would have ended the loop.

Here's the code

```
unsigned minOR(unsigned a, unsigned b, unsigned c, unsigned d)
{
   unsigned settablea = (a ^ b) == 0 ? 0 : 0xFFFFFFFF >> nlz(a ^ b);
   unsigned settablec = (c ^ d) == 0 ? 0 : 0xFFFFFFFF >> nlz(c ^ d);
   unsigned candidatebitsa = (~a & c) & settablea;
   unsigned candidatebitsc = (a & ~c) & settablec;
   unsigned candidatebits = candidatebitsa | candidatebitsc;

   unsigned target = candidatebits == 0 ? 0 : 0x80000000 >> nlz(candidatebits);
   // small optimization compared to ~a & target
   unsigned targeta = c & target;
   unsigned targetc = a & target;

   unsigned newa = a & (targeta == 0 ? -1 : -targeta);
   unsigned newc = c & (targetc == 0 ? -1 : -targetc);
   // no need to actually set the target bit, it will be 1 in the other bound by construction
   return newa | newc;
}
```

---

(15) From Paolo Bonzini, 10/18/2013:

When working with addition and subtraction a particularly useful boolean function is the "majority" or "median of three" function. Given the inputs and carry-in, this function gives the carry-out for each position. It can be expressed in many ways. For example, using AND/OR/NOT logic:

```
    majority(a, b, c) = (a & (b | c)) | (b & c)
                      = (a | (b & c)) & (b | c)
```

Where any permutation of a, b and c can be used on the right. However, there are other somewhat surprising formulas using XOR. These are often less apt to simplification, but they have one advantage: they can be used to compute the function with 4 basic RISC instructions even when one of the arguments is complemented, i.e. for majority(NOT a, b, c):

```
    majority(a, b, c) = (a ^ b) ? c : a = ((a ^ c) & (a ^ b)) ^ a
```

```
                    = (a ^ b) ? c : b = ((b ^ c) & (a ^ b)) ^ b

    majority(NOT a, b, c) = (a ^ b) ? b : c = ((b ^ c) & (a ^ b)) ^ c
```

Computing variants with two or three negated arguments requires five instructions if none of EQV, ANDC, ORC, NAND are available, using the following identities:

```
    majority(a, NOT b, NOT c) = NOT majority (NOT a, b, c)
    majority(NOT a, NOT b, NOT c) = NOT majority (a, b, c)
```

As mentioned earlier, the majority function is related to the carry-out bits in an addition or subtraction. Using the formulas for carry-in (2-13, bottom of page 30):

```
    add-carry-out(x, y) = majority(x, y, (x + y) ^ x ^ y)
```

A similar formula holds for borrow. The second operand of the majority function must be inverted, since we are treating it as the input to an adder. Since borrow is the negation of carry, we have to invert the third operand (otherwise given by the formula at the bottom of page 30) and the output:

```
    sub-borrow-out(x, y) = NOT majority(x, NOT y, NOT (x - y) ^ x ^ y)
                         = majority(NOT x, y, (x - y) ^ x ^ y)
```

More simplifications are possible. Whenever a = b the third argument c does not contribute to the result of majority(a, b, c). We can thus change the third argument arbitrarily as long as it remains the same for a != b. For add-carry-out we can thus assume x != y (x ^ y = 1); for sub-borrow-out we can instead assume x = y (x ^ y = 0):

```
    add-carry-out(x, y)  = majority(x, y, NOT (x + y))
    sub-borrow-out(x, y) = majority(NOT x, y, x - y)
```

These give the carry/borrow out of each position in the word in 5 basic RISC instructions (4 if the result of the addition or subtraction is already available). The most significant bit is the carry/borrow out of x + y and x - y respectively. Thus, the following identities can be used instead of the formulas of section 2-16:

```
    c <- (((x ^ y) & ((x + y) ^ x)) ^ y) >>u 31
    b <- (((x ^ y) & ((x - y) ^ y)) ^ (x - y)) >>u 31
```

The second is not only useful for multiprecision add/subtract. sub-borrow-out(x, y)'s most significant bit is 1 if and only if x <u y, so it can be used to evaluate predicates on unsigned values:

```
    [x <u y] = majority(NOT x, y, x - y)
```

Nothing new here---all of the "magic" formulas in 2-12 are expansions of this formula, and similarly for other relational operators; there are many ways to realize the majority function and just as many ways to write down the predicates.

Still, three particularly useful formulas are missing in 2-12. Here are two for unsigned comparison:

```
    [x <u y] = (((x ^ y) & ((x - y) ^ y)) ^ (x - y))

    [x >u y] = (x & ((y - x) | NOT y) | ((y - x) & NOT y)
```

The third is for signed less-than. To derive it we add 2^31 to both arguments (or equivalently flip the sign bit), which converts signed comparison to unsigned comparison. Since we only care about the

most significant bit, we can invert the first two arguments of the majority function entirely---not just the sign bit. Bits below the most significant are not meaningful anymore, but the sign bit gives the correct answer. Thus:

```
    [x <s y] = majority(x, NOT y, x - y)
             = (((x ^ y) & ((x - y) ^ x)) ^ (x - y))        [2-12]
             = (x & ((x - y) | NOT y) | ((x - y) & NOT y)
```

Efficient computation of carry-in and carry-out is also useful for arithmetic on BCD values, in order to convert the outcome of binary addition to valid BCD. For example:

```
          0x12 + 0x78 = 0x8A
```

is not valid BCD; the desired output is 0x90. The objective is to detect nybbles that, when summed, give a result above 9, and add 6 to these nybbles. To do this, we sum 6 to _all_ nybbles of one addend. A base-10 carry will then become a base-16 carry, that can be detected using binary techniques.

There are two ways to do this, one using carry-in and using carry-out.

```
     carry-in method for addition
     ------------------------------------------------
     s = a + b
     t = s + 0x66666666
     c = ((t ^ a ^ b) & 0x11111110) | (t < a)
     c = c rotr 1
     c -= c >> 2
     return s + c
```

In the carry-in method, the desired carry is in the lowest bit of the _immediately higher_ nybble, hence the mask used in line 3. The algorithm exploits an instructions such as SLTU to place the highest nybble's carry in bit 0, and then rotates all carries to their desired place. Then "8"s are converted to "6" and summed to the binary sum.

Strictly speaking, t is the sum of "a" and "b + 0x66666666", thus the first three lines should have been:

```
     v = b + 0x66666666
     t = a + v
     c = ((t ^ a ^ v) & 0x11111110) | (t < a)
```

However, the result of the XORs does not change if "b" is used instead. This trick in the computation of the carry lets us reuse "s" in the last instruction. The cost of this method is 11 instructions, plus 3 to handle the large constants if the algorithm is not placed in a loop.

If you can assume that there is no overflow (i.e. no carry out of the highest nybble), "t < a" can be assumed 0. This saves two instructions in the above code. Overflow can be checked by comparing the result with one of the two BCD inputs, as in binary addition. If you omit the "t < a", you should also check that the result is not higher than 0x9FFFFFFF.

If overflow must be handled, however, a better algorithm exists. It doesn't need SLTU, handles overflow nicely, and only needs 10 instructions (plus 3 for the large constants) if the machine has an EQV or ANDC instruction. The third line computes the carry-out of a + v using the above formula for add-carry-out, negates the result, and masks away uninteresting bits. The remaining two lines _subtract_ unwanted carries from "a + b + 0x66666666":

```
     carry-out method for addition (EQV version)
     -------------------------------------------
     v = b + 0x66666666
     t = a + v
     c = (((a ^ v) & (t ^ a)) ^ ~v) & 0x88888888
     c -= c >> 2
     return t - c
```

Here, "c" has the opposite meaning compared to the carry-in method, namely it includes *undesired* carries that had been included in the first two lines. [Knuth, vol. 4A, 7.1.3 exercise 100]

Quite surprisingly, subtraction is more efficient than addition, especially on a RISC machine where the range of immediates is limited. Here the "borrow" doesn't need any correction term because it happens (in every base) when a digit of the minuend is less than the corresponding digit of the subtrahend. We have to identify nybbles that caused a "borrow", and subtract 6 more units from each of them.

Both the carry-in and the carry-out methods are easily adjusted to perform subtraction. The "borrow-in" method uses 10 instructions, plus 2 to load the large constant:

```
     borrow-in method for subtraction
     ------------------------------------------------
     d = a - b
     c = ((d ^ a ^ b) & 0x11111110) | (a < b)
     c = c rotr 1
     c -= c >> 2
     return d - c
```

As before, this method is mostly useful if you can ignore overflow. This is actually relatively common for BCD subtraction because numbers are often stored in sign-magnitude form. If the magnitudes fit in a single-word, their subtraction never overflows; "a < b" will never be true and 8 instructions will suffice (plus 2 to load the large constant).

If the top nybble has to be correct even if it has a borrow, the borrow-out method can be used. It only requires 9 instructions, plus 2 to load the large constant [Knuth, ibid.], and there is no need for the EQV or ANDC instructions, even:

```
     borrow-out method for subtraction
     ---------------------------------------------
     d = a - b
     c = (((a ^ b) & (d ^ b)) ^ d) & 0x88888888
     c -= c >> 2
     return d - c
```

However, borrow-in has a redeeming grace in another important case, namely multiprecision BCD operations. These can use a similar technique to 2-16, just with BCD addition and subtraction instead of binary. When computing each word's carry out "c", the result of the sum must have had all corrections applied. However, "b" is computed from the operands and is exactly the same as the "a < b" term in the borrow-in method; thus "a < b" comes essentially for free and might even squeeze out a little extra parallelism. You should evaluate the two methods carefully as there is hardly an overall winner.

The carry-out and borrow-out methods also produce heavily streamlined code for incrementing or decrementing a BCD number. Here is the code for increment (6 instructions on full RISC, 7 without ORC, plus 3 to handle the large constants):

```
      t = a + 0x66666667
      c = (t | ~a) & 0x88888888
      c -= c >> 2
      return t - c
```

Here, massive simplification and cancellation arises because none of the bits in 0x66666667 will survive the AND with 0x88888888. Decrement is even more efficient, since it only needs one large constant... and just as mysterious:

```
      d = a - 1
      c = (d & ~a) & 0x88888888
      c -= c >> 2
      return d - c
```

BCD could use ten's complement as an equivalent to two's complement. Since subtraction is so efficient, and ten's complement is itself quite expensive to compute, it is not very useful. Ten's complement only needs an extra instruction once you have the above routines for increment or decrement. The following formulas can be used:

```
      tens(x) = (0x99999999 - x) bcd+ 1
              = 0x99999999 - (x bcd- 1)
```

Subtraction from the large constant does not need a correction, so it can be done directly in binary. But another pretty cool technique modifies the constant so as to require no correction:

```
      (0x9999999A << (ntz(x) & ~3)) - x
```

ASCII-encoded numbers are basically the same as binary-coded decimal, with hexadecimal replaced by base-256. You could apply the same algorithms; for addition, the magic number 0xF6F6F6F6 will give the right carry (or 0x96969696 if you deduct the ASCII value of '0' from the magic number; ASCII digits are in the range 0x30-0x39). But the extra bits between one digit and the next make carry-in/borrow-in methods particularly simple and efficient:

```
    t = a + b + 0x96969696;
    c = (t & 0x30303030) >> 3;
    s = (t - c) & 0x0F0F0F0F;
    return s | 0x30303030;

    d = a - b;
    c = (d & 0x30303030) >> 3;
    d = (d - c) & 0x0F0F0F0F;
    return d | 0x30303030;
```

They clock at 7 and 6 instructions respectively, or 13 and 10 respectively including loads of the large constants.

The first routine comes from http://homepage.cs.uiowa.edu/~jones/bcd/bcd.html but there is no routine for subtraction there, so I wrote the second myself. That page doesn't have carry-out/borrow-out methods, and its versions of the carry-in method for BCD is less efficient. On the other hand, it hosts subroutines to check for the validity of BCD words.

On the same page there is an addition routine for 6-bit BCD, but it is buggy. Here are two 6-bit BCD routines that work, respectively for addition and subtraction, using the same simplified carry-in/borrow-in method:

```
    t = a + b + 06666666666;
```

```
    c = (t & 06060606060) >> 3;
    return (t - c) & 01717171717;

    d = a - b;
    c = (d & 006060606060) >> 3;
    return (d - c) & 01717171717;
```

Theoretically, 6-bit BCD is faster than packed (4-bit) BCD. On the full RISC machine, not counting the cost of loading large constants, the cost is as follows (in instructions per decimal digit):

	add	subtract
32-bit		
packed (4-bit)	10/8=1.25	9/8=1.125
ASCII	7/4=1.75	6/4=1.5
6-bit	6/5=1.2	5/5=1
64-bit		
packed (4-bit)	10/16=0.625	9/16=0.562
ASCII	7/8=0.875	6/8=0.75
6-bit	6/11=0.545	5/11=0.454

When using multiprecision arithmetic, however, the less efficient encoding makes loop overhead 60% higher for 6-bit BCD. In fact, there are plausible situations where packed BCD could use single-precision arithmetic while 6-bit BCD would have to resort to multi-precision.

I'll now turn to conversion from binary to (packed) binary-coded decimal and back. For ASCII, the inefficiency of the encoding makes optimized algorithms useless in practice. I didn't produce routines for 6-bit BCD either---because octal drives me mad and because 6-bit BCD is more of a curiosity than anything else---but the same principles apply.

Conversion to binary is an almost trivial application of divide and conquer, with ceil(log2 N - 2) steps sufficing to do the conversion. The only notable thing in the following routine is how a subtraction is used to merge the low and. This saves an AND per line:

```
    h = (x >> 4)  & 0x0F0F0F0F; x -= h * (0x10 - 10);
    h = (x >> 8)  & 0x00FF00FF; x -= h * (0x100 - 100);
    h = (x >> 16);              x -= h * (0x10000 - 10000);
```

The cost is 11 instructions, 3 of which multiplications, plus 4 to load the large constants.

Conversion to BCD, instead, is the really interesting routine. A trivial implementation requires 7 multiply-high instructions, 7 multiplications by 10, and 21 other arithmetic/logical instructions:

```
    h = mulhu(x, 429496732);
    l = x - h * 10;
    x = mulhu(h, 429496732);
    l |= (h - x * 10) << 4;
    h = mulhu(x, 429496732);
    l |= (x - h * 10) << 8;
    x = mulhu(h, 429496732);
    l |= (h - x * 10) << 12;
    h = mulhu(x, 429496732);
    l |= (x - h * 10) << 16;
    x = mulhu(h, 429496732);
    l |= (h - x * 10) << 20;
    h = mulhu(x, 429496732);
```

```
    l |= (x - h * 10) << 24;
    return l|(h << 28);
```

The magic number generated by magicgu was multiplied by 4 to place the result directly in bits 32-63 of the product. Alternatively, divisions after the first could avoid multiply high at the cost of an extra shift. This may help on machines where multiply-high poses constraints on register allocation and scheduling.

```
    h = mulhu(x, 429496732);
    l = x - h * 10;
    x = (h * 6710887) >> 26;
    l |= (h - x * 10) << 4;
    h = (x * 6710887) >> 26; /* Or (838861, 23) */
    l |= (x - h * 10) << 8;
    x = (h * 6710887) >> 26; /* Or (52429, 19) */
    l |= (h - x * 10) << 12;
    h = (x * 6710887) >> 26; /* Or (3277, 15) */
    l |= (x - h * 10) << 16;
    x = (h * 6710887) >> 26; /* Or (205, 11) */
    l |= (h - x * 10) << 20;
    h = (x * 6710887) >> 26; /* Or (103, 10) */
    l |= (x - h * 10) << 24;
    return l|(h << 28);
```

As the range of the dividend becomes more limited, smaller magic numbers can be used them. The smallest possible magic numbers are included in comments above. None of them seems particularly suited to "shift and add", except perhaps x*103 = x*2*3*17+x, and 52429 = x*2*2*3*17*257+x. However, smaller immediates might also use a more compact opcode representation on CISC machines.

This method however does not scale too well to 64-bit machines, which can hold 16 BCD digits in a register. Of course it is possible to split the 16 digits in two groups of 8, so that only one 64-bit multiply high is used.

But then, one wonders if a divide-by-conquer algorithm exists, and whether it is already efficient enough on a 32-bit machine.

It turns out that the algorithm exists, and it showcases a large number of techniques from the book. Of course, a large number of divisions by constants are there; most of them can be optimized to avoid a multiply-high instruction. Some steps even treat a register as multiple subfields and perform divisions in parallel on all subfields.

The magic numbers are (103,10) for division by 10, and (5243,19) for division by 100.

Because divisions by a constant (without multiply high) need free room in the register for the result, the initial 8-digit BCD value must be split in two 4-digit groups. In each register, 16 bits hold the current stage of the conversion, and the other 16 are used for the result of division. The final steps are unrolled versions of compress-right:

```
    /* Let the compiler figure out a divmod instruction.  */
    h = x/10000; l = x%10000;

    /* Two divisions and remainders by 100 (27 bits needed).  */
    hh = (h * 5243) >> 19; h -= hh * 100; h |= hh << 16;
    ll = (l * 5243) >> 19; l -= ll * 100; l |= ll << 16;

    /* Four divisions by 10 (14 bits needed), followed by compression of the
     * nybbles instead of a remainder operation.  The high nybbles already
     * have a "weight" of 10 in h and l, we need to multiply them by 6.  The
```

```
     * next lines fuse the right shift from the division step with the first
     * shift of the multiplication by 6.  They are equivalent to this:
     *
     *     hh = ((h * 103) >> 10) & 0x000F000F; h += hh * 6;
     *     ll = ((l * 103) >> 10) & 0x000F000F; l += ll * 6;
     */
    hh = ((h * 103) >> 9) & 0x001E001E; h += hh * 3;
    ll = ((l * 103) >> 9) & 0x001E001E; l += ll * 3;

    /* Compress h and l right with mask 0x00FF00FF.  */
    h = (h & 255) | (h >> 8);
    l = (l & 255) | (l >> 8);
    return l|(h<<16);
```

Compress-left is avoided because the required masks use large constants.

On a 32-bit machine, only the final step is able to perform two divides in parallel. Still, at 8 multiplications and 27 more arithmetic/logical instructions, the efficiency of this routine is already comparable to repeated division by 10, and there is more instruction-level parallelism too because the two halves are processed more or less independently.

Depending on the machine, one could use multiply high by 42949673 to divide by 100. This places the result in bits 32-63 of the product as in the "trivial" conversion routine, saving two shifts by 19. However, depending on the instruction set it might reduce the amount of parallelism between the computation of h and the computation of l.

Because the divide-and-conquer only goes down to 4 bits in this function, all three division steps actually have different code. The recursive structure is a bit more visible in the 64-bit version. The compression step (an unrolled version of the "parallel prefix" algorithm) is more clearly identifiable in the 64-bit version, too.

The following routine processes up to 16 decimal digits in four steps. A new magic number is needed for division by 10000. The magicgu Python routine computes it as (109951163,40). Two steps are now able to use the parallel divisions trick, the final one doing four divisions with a single multiplication:

```
    /* Let the compiler figure out a divmod instruction.  */
    h = x/100000000; l = x%100000000;

    /* Two divisions and remainders by 10000 (27 bits needed).  */
    hh = (h * 109951163) >> 40; h -= hh * 10000; h |= hh << 32;
    ll = (l * 109951163) >> 40; l -= ll * 10000; l |= ll << 32;

    /* Four divisions and remainders by 100.  */
    hh = ((h * 5243) >> 19) & 0x000000FF000000FF; h -= hh * 100; h |= hh << 16;
    ll = ((l * 5243) >> 19) & 0x000000FF000000FF; l -= ll * 100; l |= ll << 16;

    /* Eight divisions by 10 (14 bits needed), followed by compression of the
     * nybbles instead of a remainder operation.
     */
    hh = ((h * 103) >> 9) & 0x001E001E001E001E; h += hh * 3;
    ll = ((l * 103) >> 9) & 0x001E001E001E001E; l += ll * 3;

    /* Compress h and l right with mask 0x00FF00FF00FF00FF.  */
    hh = (h & 0x00FF000000FF0000l); h ^= hh ^ (hh >> 8);  // 0x0000aabb0000ccdd
    ll = (l & 0x00FF000000FF0000l); l ^= ll ^ (ll >> 8);  // 0x0000aabb0000ccdd

    hh = (h & 0x0000FFFF00000000l); h ^= hh ^ (hh >> 16); // 0x00000000aabbccdd
    ll = (l & 0x0000FFFF00000000l); l ^= ll ^ (ll >> 16); // 0x00000000aabbccdd

    /* Combine the two (simpler than compressing h left, also because masks
```

```
     * are reused).  */
    return l|(h<<32);
```

At 12 multiplications and 49 arithmetic/logical instructions (not counting the cost of the large constants), the 64-bit versions has less multiplications than digits!

---

(16) From Jasper Neumann, 8/24/2013:

Dear Mr. Warren!

Please let me comment on Hacker's Delight 10-17 (Test for zero remainder after division by a constant).

When the given instruction sequence is enhanced by a leading subtraction of c, we can map any set of signed or unsigned numbers x_i = d*i+c to i where i is in the range (0..s-1) and get a free range check too.
Assume that d = a * 2**b where a is odd.
In x86 assembler (eax=x_i):

```
   sub eax,c
   ror eax,b
   imul eax,eax,inv_mul(a)
   cmp eax,s
```

Now eax contains i and the carry flag is set if the i is in the given range 0..s-1.
Alternatively the last statement can be "cmp eax,s-1" with the conditional test being "below or equal". This should be equivalent for all cases but the trivial one (d=1, empty or full range).
Needless to say, "sub eax,0", "ror eax,0" and "imul eax,eax,1" can be suppressed.
Interestingly, imul and ror can be exchanged for all cases where x_i is in range.

This is much more than only a divisibility check:
- a divisibility check (or check for any constant modulo)
- a division
- a range check
- a map into a compact range

This might find an application for "reversible hashing" which is sketched in Roger A. Sayle's "A Superoptimizer Analysis of Multiway Branch Code Generation " (Proceedings of the GCC Developers' Summit, 2008), see http://ols.fedoraproject.org/GCC/Reprints-2008/sayle-reprint.pdf. I'm currently preparing a paper concerning hashing.

Best regards
Jasper Neumann

---

(17) From Peter Polm, 8/24/2013:

Dear Mr. Warren,

I think the latest version of this algorithm is at:

http://www.hackersdelight.org/hdcodetxt/icbrt.c.txt

```
int icbrt2(unsigned x) {
   int s;
   unsigned y, b, y2;
   y2 = 0;
```

```
   y = 0;
   for (s = 30; s >= 0; s = s - 3) {
      y2 = 4*y2;
      y = 2*y;
      b = (3*(y2 + y) + 1) << s;
      if (x >= b) {
         x = x - b;
         y2 = y2 + 2*y + 1;
         y = y + 1;
      }
   }
   return y;
}
```

Slightly faster, at least on my pc with C#, one comparison (s >= 0) less,
the "for" loop is replaced by a "do while" loop:

```
int icbrt2(unsigned x) {
   int s;
   unsigned y, b, y2;
   s = 30;
   y2 = 0;
   y = 0;
   do {
      y2 = 4*y2;
      y = 2*y;
      b = (3*(y2 + y) + 1) << s;
      if (x >= b) {
         x = x - b;
         y2 = y2 + 2*y + 1;
         y = y + 1;
      }
      s -= 3;
    }
    while ( s >=0);
   }
   return y;
}
```

Another obvious trick, count the number of leading (triple) zeros, in C#:

```
private static uint cro32(uint x)
    {
        uint y = 0, z = 0, b = 0;
        int s = x < 1u << 24 ? x < 1u << 12 ? x < 1u << 06 ? x < 1u << 03 ? 00 : 03 :
                                                             x < 1u << 09 ? 06 : 09 :
                                               x < 1u << 18 ? x < 1u << 15 ? 12 : 15 :
                                                             x < 1u << 21 ? 18 : 21 :
                                  x >= 1u << 30 ? 30 : x < 1u << 27 ? 24 : 27;
        do
        {
            y *= 2;
            z *= 4;
            b = 3 * y + 3 * z + 1 << s;
            if (x >= b)
            {
                x -= b;
                z += 2 * y + 1;
                y += 1;
            }
            s -= 3;
        }
        while (s >= 0);
```

```
        return y;
    }
```

Finally: An amazing algorithm, thanks!
http://www.hackersdelight.org/hdcodetxt/acbrt.c.txt
My C# versions ("cro64, cro32, ..."):
http://bigintegers.blogspot.com/2013/08/small-cube-roots-264.html

Regards,

Peter Polm

---

(18) From Al Grant, 7/1/14:

Hi,

I've a happy owner of both editions of your book. I'd like to point out an alternative implementation of the parity function, that might be faster on some CPUs. I must admit, I thought it would be known already, but I've not seen any mention of anything like it in your book or elsewhere.

Here's the original parallel-prefix parity function (2nd ed, p. 96):

```
  unsigned int par1(unsigned int x)
  {
    unsigned int y;
    y = x ^ (x >> 1);
    y = y ^ (y >> 2);
    y = y ^ (y >> 4);
    y = y ^ (y >> 8);
    y = y ^ (y >> 16);
    return y & 1;
  }
```

This alternates shifts and XORs. Each operation depends on the result of the previous operation. So the critical path length is 10 operations, or 11 with the final AND.

Now consider this function:

```
  unsigned int par2(unsigned int x)
  {
    unsigned int y0, y1;
    y0 = x ^ (x >> 1);
    y1 = y0 ^ (x >> 2);
    y0 = y1 ^ (y0 >> 3);
    y1 = y0 ^ (y1 >> 5);
    y0 = y1 ^ (y0 >> 8);
    y1 = y0 ^ (y1 >> 13);
    y0 = y1 ^ (y0 >> 21);
    return y0 & 1;
  }
```

There are more shift and XOR operations overall, but the critical path is shorter - 8 operations, or 9 with the final AND. On superscalar CPUs (issuing multiple instructions per cycle and with multiple ALUs - found even in smartphones these days) this is likely to have a shorter latency.

On my PC, I timed a billion calls to each function, with the value depending on the result of the previous call. For the original function this ran in 6.0s, for the new function it ran in 4.5s.

These functions are equivalent even without the final AND - i.e. this is

also a way to speed up the inverse Gray code function (p. 312).
And it works for OR or AND, not just XOR.

Al Grant

---

(19) From Michael Lee Finney, 3/11/2015:
Subject: Determining leading/trailing zeros

Hi,

I have a modest improvement to the non-table, non-branching version
for counting leading / trailing zeros. Ignoring x == 0 can save a few
additional cycles.

```
// Return the number of leading zeros.

   unsigned leadingZeros(
      unsigned x)
      {
      unsigned n = 31;
      unsigned m;

      m = (x > 0xffffu) << 4; x >>= m; n -= m;
      m = (x > 0x00ffu) << 3; x >>= m; n -= m;
      m = (x > 0x000fu) << 2; x >>= m; n -= m;

      return n - ((0xffffaa50u >> (x + x)) & 0x03u) + (x == 0);
      }

// Return the number of trailing zeros.

   unsigned trailingZeros(
      unsigned x)
      {
      unsigned n;
      unsigned m;

      x &= -x;
      n = (x > 0xffffu) << 4; x >>= n;
      m = (x > 0x00ffu) << 3; x >>= m; n += m;
      m = (x > 0x000fu) << 2; x >>= m; n += m;

      return n + ((0x12131210u >> (x + x)) & 0x03u) + ((x == 0) << 5);
      }
```

The magic numbers 0xffffaa50u / 0x12131210u are based on the fact
that the bottom four bits are...

```
   0000 / 0000 => 0 (this only occurs when the entire number is zero)
   0001 / xxx1 => 0
   001x / xx10 => 1
   01xx / x100 => 2
   1xxx / 1000 => 3
```

Converting to a lookup table, most to least significant (with repeats
for the "x" positions) gives...

```
   3 3 3 3 3 3 3 3 2 2 2 2 1 1 0 0

   1111 1111 1111 1111 1010 1010 0101 0000 == ffffaa50
```

and

```
   1 0 2 0 1 0 3 0 1 0 2 0 1 0 0

   0001 0010 0001 0011 0001 0010 0001 0011 == 12131210
```

then, if the bottom four bits are abcd, right shifting the above by abcd0 and masking with 0x03 recovers the bit position. That takes four operations, but eliminates two passes of the basic algorithm.

Michael Lee Finney

---

(20) From John Souvestre - New Orleans LA, 5/24/2015:
Subject: Using CRC to do single-bit error correction

I just purchased a copy of Hacker's Delight, second edition, third printing. I've only had time to leaf through it, but it looks great! Thanks for writing it.

I was attracted to the chapter on CRC's. They have been an interest of mine for a long time. At the end of Section 14-2 you talk about doing one-bit error correction. I've done this.

It was about 20 years ago in some 8031/8051 code I wrote for an embedded microcomputer in a spread-spectrum radio receiver. It was a one-way system so there was no way to get a message repeated. The messages were short enough that 2 bit error detection was guaranteed.

At first, I did as you described. I walked a one-bit change through the message recalculating the CRC each time. This took quite a while. Also, sometimes there was more than 1 error. But often it worked. The RF Engineer on the project compared receivers with and without the one-bit correction and he found that it improved RF performance by about 1.8dB, if memory serves. Very worthwhile!

It doesn't scale well for long messages, however, since the worst case correction time is O(n^2).

A few years later, after I had left the company, I began wondering if there wasn't a faster way. Indeed, there was. There are three tricks to it.

The first trick: Instead of walking the correction bit through the real message, I found that I could use a dummy message initialized to all zeroes. The result was then XOR'ed (actually just compared) with the original message's result to see if it produced the correct, all zero result.

The second trick: Instead of starting the error bit at the beginning of the message, I started it at the end. The initial CRC was then for a message with all zeroes except for the last bit. Since the leading zeroes could be ignored, the CRC calculation was trivial. I then walked the bit forward. The calculation took longer and longer, but even so the overall average time (middle of the message) was lots less than it had been.

The third trick: Say you had just a 4-bit message. As per above, the error calculation started on 0001, then 0010, then 0100, then 1000. Ignoring the leading zeroes this becomes 1, then 10, then 100, then 1000. Each successive calculation is for a message just 1 zero bit longer. So instead of calculating the CRC from scratch you can start with the result from the previous step and calculate in just one more zero. Easy and fast. Now the worst case correction time is O(n).

I wrote test code which proved that this worked. I spoke to my old company about doing this in their production code but I never got around to it.

I'm telling you this story because I thought that you might enjoy it. :)

===

An optimization I often did was to calculate 8 bits in a row, looping just once per byte. This saved most of the loop overhead.

I found that a similar optimization helps when I did table driven CRC's. If memory serves, each data byte's table lookup (8-bit register machine, 16-bit CRC) required a register swap. By coding a pair of bytes per loop I could write the code with the registers reversed for the second byte. Actually, I coded for 4 pairs of bytes, to limit the loop overhead too. Then I had a regular single byte loop to handle the leftover bytes.

I've worked with 4-bit (microcomputer with very little ROM), 8-bit and 16-bit lookup tables.

Regards,

   John Souvestre - New Orleans LA

---

(21) From Neil Viberg, 6/25/15:
Subject: Reiser's (aka Crocker's) algorithm for a 128-bit word

The divisor for 128-bit unsigned integers is 131 and, using Reiser’s conventions, the table is:

```
static char table[131] = {
       128, 0, 1, 72, 2, 46, 73, 96,
       3, 14, 47, 56, 74, 18, 97, 118,
       4, 43, 15, 35, 48, 38, 57, 23,
       75, 92, 19, 86, 98, 51, 119, 29,
       5, u, 44, 12, 16, 41, 36, 90,
       49, 126, 39, 124, 58, 60, 24, 105,
       76, 62, 93, 115, 20, 26, 87, 102,
       99, 107, 52, 82, 120, 78, 30, 110,
       6, 64, u, 71, 45, 95, 13, 55,
       17, 117, 42, 34, 37, 22, 91, 85,
       50, 28, 127, 11, 40, 89, 125, 123,
       59, 104, 61, 114, 25, 101, 106, 81,
       77, 109, 63, 70, 94, 54, 116, 33,
       21, 84, 27, 10, 88, 122, 103, 113,
       100, 80, 108, 69, 53, 32, 83, 9,
       121, 112, 79, 68, 31, 8, 111, 67,
       7, 66, 65};
```

---

(22) From Christopher Dannemiller, 10/23/14:
Subject: CRC32C "slicing by 8" algorithm

This algorithm, developed by Intel engineers, is said to be faster than the CRC32 algorithm of Hacker's Delight Second Edition Figure 14-7, as modified by Exercise 2 (page 329). It is too long to be included here, but you can find it at http://slicing-by-8.sourceforge.net/ and http://www.evanjones.ca/crc32c.html, among other web sites.

---

(23) From Geoff Bailey, 11/6/14:
Subject: Average number of trailing 0's in words with n 1-bits

In analyzing the execution time of David de Kloet's snoob4 algorithm, the question arises: For a random collection of words of length w bits, each containing exactly n 1's, what is the average number of trailing zeros?

Geoff Bailey offers the following solution. Let C(n, k) denote the binomial coefficient or "choose k from n" function. Then Geoff writes:

> There are C(w-1, n) items with at least one trailing 0, C(w-2, n) with at least two trailing zeroes, ... and C(n, n) items with at least (w - n) trailing zeroes (the maximum). Each term contributes 1 to the total, so we get the sum
>
> C(n, n) + C(n+1, n) + ... + C(w-1, n) = C(w, n+1)
>
> The average is thus C(w, n+1) / C(w, n) = (w - n) / (n + 1) as before.
>
> I suspect that the right way of thinking about things could avoid the need for summation at all, but I cannot see it offhand.

This editor does not have his intuition about this, but after seeing his solution would be inclined to compute a weighted average of the number of words with no trailing zeros, the number with exactly one trailing zero, ... , the number with w-n trailing zeros (the maximum). This would be

0*C(w-1, n-1) + 1*C(w-2, n-1) + 2*C(w-3, n-1) + ... + (w-n)*C(n-1, n-1)

The average would be this sum divided by the total number of words that have exactly n 1-bits, which is C(w, n). Both methods seem to give the same result.

---

(24) From Jan Tari, 8/30/17:
Subject: Population count with in-register indexing

... you suggested (p. 86 of 2nd ed.) one way was using a byte array to implement popcount (I think you described it as 'uninteresting' but that's probably how I'd have done it originally) but I didn't like the memory accesses as stuff has to be brought in to cache and then may be evicted, and I know how expensive this can be.

Assuming we're using a 64-bit machine, it occurred to me that if one did half-byte lookups, then the array could fit into a single register and be 'indexed' with a shift, and a mask in another register. 1/2 byte = 4 bits = 16 combinations, each combination having 0 to 4 bits so 3 bits to represent the value, but use 4 instead for easy multiplication (double shift), 16 * 4 = 64 bits. No branching needed if loop unrolled.

Advantage over the byte indexing method - no external mem acceses.
Advantage over the method in fig 5-2 on page 82 of 2nd ed - probably none actually, except perhaps fewer large constants to be loaded or tie up registers, and not sure my way actually uses fewer regs.

Just a thought.