REVISIONS

These pages are revisions, mostly new material, for Hacker's Delight. This material is relative to the sixth printing.

- - -

Page 4, insert this new paragraph just before Section 1–2:

The ISO C standard does not specify whether right shifts ("$>>$" operator) of signed quantities are 0-propagating or sign-propagating. In the C code herein, it is assumed that if the left operand is signed, then a sign-propagating shift results (and if it is unsigned then a 0-propagating shift results, following ISO). Most modern C compilers work this way.

Another potential problem with shifts is that the ISO C standard specifies that if the shift amount is negative or is greater than or equal to the width of the left operand, the result is undefined. But, nearly all 32-bit machines treat shift amounts modulo 32 or 64. The code herein relies on one of these behaviors; an explanation is given when the distinction is important.

- - -

Page 12, replace the second paragraph with (this is adding a second formula):

Use either of the following formulas to turn off the rightmost contiguous string of 1-bits (e.g., 01011000 ==> 01000000) [Wood]:

$$(((\boldsymbol{x} \mid (\boldsymbol{x} - 1)) + 1) \mathbin{\&} \boldsymbol{x}), \quad \text{or}$$
$$((\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}) + \boldsymbol{x}) \mathbin{\&} \boldsymbol{x}$$

These may be used ….

- - -

Page 12, replace the third paragraph with:

These formulas all have duals in the following sense. Read what the formula does, interchanging 1's and 0's in the description. Then, in the formula, for all variables and subexpressions $\boldsymbol{x}$, replace $\boldsymbol{x} - \mathbf{1}$ with $\boldsymbol{x} + \mathbf{1}$, $\boldsymbol{x} + \mathbf{1}$ with $\boldsymbol{x} - \mathbf{1}$, $-\boldsymbol{x}$

with $\neg(\boldsymbol{x}+\mathbf{1})$, & with |, and | with &. Leave $\boldsymbol{x}$ and $\neg\boldsymbol{x}$ alone. If the formula has subexpressions other than those listed, then it may not have a dual. The result of these substitutions is a valid description and formula. For example, the dual of the first formula in this section reads as follows:

- - -

Page 18, insert this new section just after Section 2–4:

2–5 Average of Two Integers

The following formula may be used to compute the average of two unsigned integers, $\lfloor (x+y)/2 \rfloor$, without causing overflow [Dietz]:

$$(\boldsymbol{x} \,\&\, \boldsymbol{y}) + ((\boldsymbol{x} \oplus \boldsymbol{y}) \overset{u}{\gg} 1) \tag{3}$$

The formula below computes $\lceil (x+y)/2 \rceil$ for unsigned integers:

$$(\boldsymbol{x} \mid \boldsymbol{y}) - ((\boldsymbol{x} \oplus \boldsymbol{y}) \overset{u}{\gg} 1)$$

To compute the same quantities (“floor and ceiling averages”) for signed integers, use the same formulas but with the unsigned shift replaced with a signed shift.

For signed integers, one might also want the average with the division by 2 rounded toward 0. Computing this “truncated average” (without causing overflow) is a little more difficult. It may be done by computing the floor average and then correcting it. The correction is to add 1 if $x+y$ is negative and exactly one of $\boldsymbol{x}$ and $\boldsymbol{y}$ is odd. But $x+y$ is negative if and only if the result of (3) with the unsigned shift replaced with a signed shift, is negative. This leads to the following method (seven instructions on the basic RISC, after commoning the subexpression $\boldsymbol{x} \oplus \boldsymbol{y}$):

$$\boldsymbol{t} \leftarrow (\boldsymbol{x} \,\&\, \boldsymbol{y}) + ((\boldsymbol{x} \oplus \boldsymbol{y}) \overset{s}{\gg} 1);$$
$$\boldsymbol{t} + ((\boldsymbol{t} \overset{u}{\gg} 31) \,\&\, (\boldsymbol{x} \oplus \boldsymbol{y}))$$

Some common special cases can be done more efficiently. If $\boldsymbol{x}$ and $\boldsymbol{y}$ are signed integers and known to be nonnegative, then the average can be computed as simply $(\boldsymbol{x}+\boldsymbol{y}) \overset{u}{\gg} 1$. The sum can overflow, but the overflow bit is retained in the register that holds the sum, so that the unsigned shift moves the overflow bit to the proper position and supplies a zero sign bit.

If $\boldsymbol{x}$ and $\boldsymbol{y}$ are unsigned integers and $\boldsymbol{x} \overset{u}{\leq} \boldsymbol{y}$, or if $\boldsymbol{x}$ and $\boldsymbol{y}$ are signed integers and $\boldsymbol{x} \leq \boldsymbol{y}$ (signed comparison), then the average is given by $\boldsymbol{x} + ((\boldsymbol{y} - \boldsymbol{x}) \overset{u}{\gg} 1)$. These are “floor averages,” e.g., the average of –1 and 0 is –1.

- - -

Page 24, section "**Comparison Predicates from the Carry Bit**," add some material to the formulas on the fourth and fifth lines, so they are:

$$\boldsymbol{x} < \boldsymbol{y}: \qquad \neg\text{carry}((\boldsymbol{x} + \mathbf{2^{31}}) - (\boldsymbol{y} + \mathbf{2^{31}})), \text{ or } \neg\text{carry}(\boldsymbol{x} - \boldsymbol{y}) \oplus \boldsymbol{x}_{31} \oplus \boldsymbol{y}_{31}$$

$$\boldsymbol{x} \le \boldsymbol{y}: \qquad \text{carry}((\boldsymbol{y} + \mathbf{2^{31}}) - (\boldsymbol{x} + \mathbf{2^{31}})), \text{ or } \text{carry}(\boldsymbol{y} - \boldsymbol{x}) \oplus \boldsymbol{x}_{31} \oplus \boldsymbol{y}_{31}$$

- - -

Page 33, add the following to the end of Section 2–12:

Some machines have a "long division" instruction (see page 148), and you may want to predict, using elementary instructions, when it would overflow. We will discuss this in terms of an instruction that divides a doubleword by a fullword, producing a fullword quotient and possibly also a fullword remainder.

Such an instruction overflows if either the divisor is 0 or if the quotient cannot be represented in 32 bits. Typically, in these overflow cases both the quotient and remainder are incorrect. The remainder cannot overflow in the sense of being too large to represent in 32 bits (it is less than the divisor in magnitude), so the test that the remainder will be correct is the same as the test that the quotient will be correct.

We assume the machine either has 64-bit general registers, or it has 32-bit registers and there is no problem doing elementary operations (shifts, adds, and so forth) on 64-bit quantities. For example, the compiler might implement a doubleword integer data type.

In the unsigned case the test is trivial: for $\boldsymbol{x} \div \boldsymbol{y}$ with $\boldsymbol{x}$ a doubleword and $\boldsymbol{y}$ a fullword, the division will not overflow if (and only if) either of the following equivalent expressions is true.

$$\boldsymbol{y} \ne 0 \ \& \ \boldsymbol{x} < (\boldsymbol{y} \ll 32)$$

$$\boldsymbol{y} \ne 0 \ \& \ (\boldsymbol{x} \overset{u}{\gg} 32) < \boldsymbol{y}$$

On a 32-bit machine, the shifts need not be done; simply compare $\boldsymbol{y}$ to the register that contains the high-order half of $\boldsymbol{x}$. On a 64-bit machine, to ensure correct results it is also necessary to check that the divisor $\boldsymbol{y}$ is a 32-bit quantity (e.g., check that $(\boldsymbol{y} \overset{u}{\gg} 32) = 0$).

The signed case is more interesting. It is first necessary to check that $\boldsymbol{y} \ne 0$ and, on a 64-bit machine, that $\boldsymbol{y}$ is correctly represented in 32 bits (check that $((\boldsymbol{y} \ll 32) \overset{s}{\gg} 32) = \boldsymbol{y}$.) Assuming these tests have been done, the array below shows how the tests might be done to determine precisely whether or not the quotient is representable in 32 bits, by considering separately the four cases of the dividend and divisor being positive or negative. The expressions in the array are in ordinary arithmetic, not computer arithmetic.

In each column, each relation follows from the one above it in an if-and-only-if way. To remove the floor and ceiling functions, some relations from Theorem D1 on page 139 are used.

$x \ge 0, y > 0$	$x \ge 0, y < 0$	$x < 0, y > 0$	$x < 0, y < 0$
$\lfloor x/y \rfloor < 2^{31}$	$\lceil x/y \rceil \ge -2^{31}$	$\lceil x/y \rceil \ge -2^{31}$	$\lfloor x/y \rfloor < 2^{31}$
$x/y < 2^{31}$	$\lceil x/y \rceil > -2^{31} - 1$	$\lceil x/y \rceil > -2^{31} - 1$	$x/y < 2^{31}$
$x < 2^{31}y$	$x/y > -2^{31} - 1$	$x/y > -2^{31} - 1$	$x > 2^{31}y$
	$x < -2^{31}y - y$	$x > -2^{31}y - y$	$-x < 2^{31}(-y)$
	$x < 2^{31}(-y) + (-y)$	$-x < 2^{31}y + y$	

As an example of interpreting this array, consider the leftmost column. It applies to the case in which $x \ge 0$ and $y > 0$. In this case the quotient is $\lfloor x/y \rfloor$, and this must be strictly less than 2^{31} to be representable as a 32-bit quantity. From this it follows that the real number x/y must be less than 2^{31}, or x must be less than $2^{31}y$. This test can be implemented by shifting y left 31 positions and comparing the result to x.

When the signs of x and y differ, the quotient of conventional division is $\lceil x/y \rceil$. Because the quotient is negative, it can be as small as -2^{31}.

In the bottom row of each column, the comparisons are all of the same type (less than). Because of the possibility that x is the maximum negative number, in the third and fourth columns an unsigned comparison must be used. In the first two columns the quantities being compared begin with a leading 0-bit, so an unsigned comparison can be used there too.

These tests can of course be implemented by using conditional branches to separate out the four cases, doing the indicated arithmetic, and then doing a final compare and branch to the code for the overflow or non-overflow case. However, branching can be reduced by taking advantage of the fact that when y is negative, $-y$ is used, and similarly for x. Hence the tests may be made more uniform by using the absolute values of x and y. Using also a standard device for optionally doing the additions in the second and third columns results in the following scheme.

$$\begin{aligned}
\boldsymbol{x'} &= |\boldsymbol{x}| \\
\boldsymbol{y'} &= |\boldsymbol{y}| \\
\delta &= ((\boldsymbol{x} \oplus \boldsymbol{y}) \overset{s}{\gg} 63)\ \&\ \boldsymbol{y'}
\end{aligned}$$

if $\boldsymbol{x'} < (\boldsymbol{y'} \ll 31) + \delta$ then {will not overflow}

Using the three-instruction method of computing the absolute value (see page 17), on a 64-bit version of the basic RISC this amounts to 12 instructions plus a conditional branch.

- - -

Page 34, add the following paragraphs to Section 2–14:

If your machine has double-length shifts, they may be used to do rotate shifts. These instructions might be written

```
shldi RT,RA,RB,I
shrdi RT,RA,RB,I
```

They treat the concatenation of RA and RB as a single double-length quantity, and shift it left or right by the amount given by the immediate field I. (If the shift amount is in a register, the instructions are awkward to implement on most RISCs because they require reading three registers.) The result of the left shift is the high-order word of the shifted double-length quantity, and the result of the right shift is the low-order word.

Using shldi, a rotate left of Rx may be accomplished by

```
shldi RT,Rx,Rx,I
```

and similarly a rotate right shift can be accomplished with `shrdi`.

A rotate left shift of one position may be accomplished by adding the contents of a register to itself with “end-around carry” (adding the carry that results from the addition to the sum in the low-order position). Most machines do not have that instruction, but on many machines it may be accomplished with two instructions: (1) add the contents of the register to itself, generating a carry (into a status register), and (2) add the carry to the sum.

- - -

Pages 37-38, replace Section 2–18 with the following:

2–18 Doz, Max, Min

The “doz” function is “difference or zero,” defined as follows:

Signed Unsigned

$$\text{doz}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y. \end{cases} \qquad \text{dozu}(x, y) = \begin{cases} x - y, & x \overset{u}{\geq} y, \\ 0, & x \overset{u}{<} y. \end{cases}$$

It has been called “first grade subtraction” because the result is 0 if you try to take away too much. If implemented as a computer instruction, perhaps its most important use is to implement the max($\boldsymbol{x}, \boldsymbol{y}$) and min($\boldsymbol{x}, \boldsymbol{y}$) functions (in both signed and unsigned forms) in just two simple instructions, as will be seen. Implementing max($\boldsymbol{x}, \boldsymbol{y}$) and min($\boldsymbol{x}, \boldsymbol{y}$) in hardware is difficult because the machine would need paths from the output ports of the register file back to an input port, bypassing the adder. These paths are not normally present. If supplied, they would be in a region that’s often crowded with wiring for register bypasses. The situation is illustrated in Figure 2–1. The adder is used (by the instruction) to do the sub-

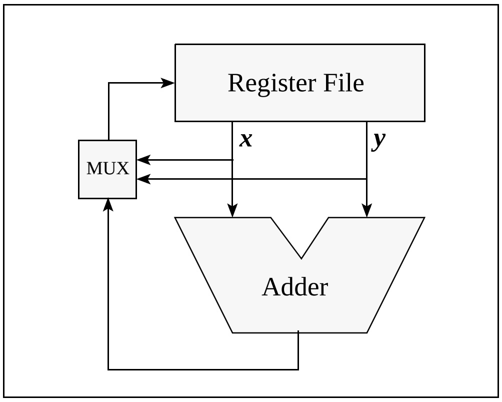


FIGURE 2–1. Implementing max($\boldsymbol{x}, \boldsymbol{y}$) and min($\boldsymbol{x}, \boldsymbol{y}$).

traction $\boldsymbol{x} - \boldsymbol{y}$. The high-order bits of the result of the subtraction (sign bit and carries, as described on page 25) define whether $\boldsymbol{x} \geq \boldsymbol{y}$ or $\boldsymbol{x} < \boldsymbol{y}$. The comparison result is fed to a multiplexor (MUX) which selects either $\boldsymbol{x}$ or $\boldsymbol{y}$ as the result to write into the target register. These paths, from register file outputs $\boldsymbol{x}$ and $\boldsymbol{y}$ to the multiplexor, are not normally present and would have little use. The *difference or zero* instructions can be implemented without these paths because it is the output of the adder (or 0) that is fed back to the register file.

Using *difference or zero,* max($\boldsymbol{x}, \boldsymbol{y}$) and min($\boldsymbol{x}, \boldsymbol{y}$) can be implemented in two instructions as follows.

Signed	Unsigned
$\max(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{y} + \mathrm{doz}(\boldsymbol{x}, \boldsymbol{y})$	$\mathrm{maxu}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{y} + \mathrm{dozu}(\boldsymbol{x}, \boldsymbol{y})$
$\min(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} - \mathrm{doz}(\boldsymbol{x}, \boldsymbol{y})$	$\mathrm{minu}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} - \mathrm{dozu}(\boldsymbol{x}, \boldsymbol{y})$

In the signed case, the result of the *difference or zero* instruction can be negative. This happens if overflow occurs in the subtraction. Overflow should be ignored; the addition of $\boldsymbol{y}$ or subtraction from $\boldsymbol{x}$ will overflow again, and the result will be correct. When doz($\boldsymbol{x}, \boldsymbol{y}$) is negative, it is actually the correct difference if it is interpreted as an unsigned integer.

Suppose your computer does not have the *difference or zero* instructions, but you want to code doz($\boldsymbol{x}, \boldsymbol{y}$), max($\boldsymbol{x}, \boldsymbol{y}$), and so forth, in an efficient branch-free way. In the next few paragraphs we show how these functions might be coded if your machine has the *conditional move* instructions, comparison predicates, efficient access to the carry bit, or none of these.

If your machine has the *conditional move* instructions, it can get doz($\boldsymbol{x}, \boldsymbol{y}$) in three instructions, and destructive[1] max($\boldsymbol{x}, \boldsymbol{y}$) and min($\boldsymbol{x}, \boldsymbol{y}$) in two instructions. For example, on the full RISC, $\boldsymbol{z} \leftarrow \text{doz}(\boldsymbol{x}, \boldsymbol{y})$ can be calculated as follows (r0 is a permanent zero register):

```
sub     z,x,y        Set z = x - y.
cmplt   t,x,y        Set t = 1 if x < y, else 0.
movne   z,t,r0       Set z = 0 if x < y.
```

Also on the full RISC, $\boldsymbol{x} \leftarrow \max(\boldsymbol{x}, \boldsymbol{y})$ can be calculated as follows:

```
cmplt   t,x,y        Set t = 1 if x < y, else 0.
movne   x,t,y        Set x = y if x < y.
```

The min function, and the unsigned counterparts, are obtained by changing the comparison conditions.

These functions can be computed in four or five instructions using comparison predicates (three or four if the comparison predicates give a result of –1 for “true”):

$$\begin{aligned}
\text{doz}(\boldsymbol{x}, \boldsymbol{y}) &= (\boldsymbol{x} - \boldsymbol{y}) \mathbin{\&} -(\boldsymbol{x} \ge \boldsymbol{y}) \\
\max(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{y} + \text{doz}(\boldsymbol{x}, \boldsymbol{y}) \\
&= ((\boldsymbol{x} \oplus \boldsymbol{y}) \mathbin{\&} -(\boldsymbol{x} \ge \boldsymbol{y})) \oplus \boldsymbol{y} \\
\min(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{x} - \text{doz}(\boldsymbol{x}, \boldsymbol{y}) \\
&= ((\boldsymbol{x} \oplus \boldsymbol{y}) \mathbin{\&} -(\boldsymbol{x} \le \boldsymbol{y})) \oplus \boldsymbol{y}
\end{aligned}$$

On some machines the carry bit may be a useful aid to computing the unsigned versions of these functions. Let $\text{carry}(\boldsymbol{x} - \boldsymbol{y})$ denote the bit that comes out of the adder for the operation $\boldsymbol{x} + \bar{\boldsymbol{y}} + \mathbf{1}$, moved to a GPR. Thus $\text{carry}(\boldsymbol{x} - \boldsymbol{y})$ = 1 iff $\boldsymbol{x} \ge \boldsymbol{y}$. Then we have

$$\begin{aligned}
\text{dozu}(\boldsymbol{x}, \boldsymbol{y}) &= ((\boldsymbol{x} - \boldsymbol{y}) \mathbin{\&} \neg(\text{carry}(\boldsymbol{x} - \boldsymbol{y}) - \mathbf{1})) \\
\text{maxu}(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{x} - ((\boldsymbol{x} - \boldsymbol{y}) \mathbin{\&} (\text{carry}(\boldsymbol{x} - \boldsymbol{y}) - \mathbf{1})) \\
\text{minu}(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{y} + ((\boldsymbol{x} - \boldsymbol{y}) \mathbin{\&} (\text{carry}(\boldsymbol{x} - \boldsymbol{y}) - \mathbf{1}))
\end{aligned}$$

On most machines that have a *subtract* that generates a carry or borrow, and another form of *subtract* that uses that carry or borrow as an input, the expression

1. A destructive operation is one that overwrites one or more of its arguments.

$\text{carry}(\boldsymbol{x} - \boldsymbol{y}) - \mathbf{1}$ can be computed in one more instruction after the subtraction of $\boldsymbol{y}$ from $\boldsymbol{x}$. For example, on the Intel x86 machines, $\text{minu}(\boldsymbol{x}, \boldsymbol{y})$ can be computed in four instructions as follows:

```
sub eax,ecx    ; Inputs x and y are in eax and ecx resp.
sbb edx,edx    ; edx = 0 if x >= y, else -1.
and eax,edx    ; 0 if x >= y, else x - y.
add eax,ecx    ; Add y, giving y if x >= y, else x.
```

In this way, all three of the functions can be computed in four instructions (three instructions for $\text{dozu}(\boldsymbol{x}, \boldsymbol{y})$ if the machine has *and with complement*).

A method that applies to nearly any RISC is to use one of the above expressions that employ a comparison predicate, and to substitute for the predicate one of the expressions given on page 22. For example:

$$\begin{aligned}
&\boldsymbol{d} \leftarrow \boldsymbol{x} - \boldsymbol{y} \\
&\text{doz}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \boldsymbol{d} \,\&\, [(\boldsymbol{d} \equiv ((\boldsymbol{x} \oplus \boldsymbol{y}) \,\&\, (\boldsymbol{d} \oplus \boldsymbol{x}))) \overset{s}{\gg} 31] \\
&\text{dozu}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \boldsymbol{d} \,\&\, \neg[((\neg \boldsymbol{x} \,\&\, \boldsymbol{y}) \mid ((\boldsymbol{x} \equiv \boldsymbol{y}) \,\&\, \boldsymbol{d})) \overset{s}{\gg} 31]
\end{aligned}$$

These require from seven to ten instructions, depending on the computer's instruction set, plus one more to get max or min.

These operations can be done in four branch free basic RISC instructions if it is known that $-2^{31} \le x - y \le 2^{31} - 1$ (that is an expression in ordinary arithmetic, not computer arithmetic). The same code works for both signed and unsigned integers, with the same restriction on x and y. A sufficient condition for these formulas to be valid is that, for signed integers, $-2^{30} \le x, y \le 2^{30} - 1$, and for unsigned integers, $0 \le x, y \le 2^{31} - 1$.

$$\begin{aligned}
\text{doz}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \text{dozu}(\boldsymbol{x}, \boldsymbol{y}) &\;=\; (\boldsymbol{x} - \boldsymbol{y}) \,\&\, \neg((\boldsymbol{x} - \boldsymbol{y}) \overset{s}{\gg} 31) \\
\text{max}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \text{maxu}(\boldsymbol{x}, \boldsymbol{y}) &\;=\; \boldsymbol{x} - ((\boldsymbol{x} - \boldsymbol{y}) \,\&\, ((\boldsymbol{x} - \boldsymbol{y}) \overset{s}{\gg} 31)) \\
\text{min}(\boldsymbol{x}, \boldsymbol{y}) \;=\; \text{minu}(\boldsymbol{x}, \boldsymbol{y}) &\;=\; \boldsymbol{y} + ((\boldsymbol{x} - \boldsymbol{y}) \,\&\, ((\boldsymbol{x} - \boldsymbol{y}) \overset{s}{\gg} 31))
\end{aligned}$$

Some uses of the *difference or zero* instruction are given below. In these, the result of $\text{doz}(\boldsymbol{x}, \boldsymbol{y})$ must be interpreted as an unsigned integer.

1. It directly implements the Fortran IDIM function.
2. To compute the absolute value of a difference [Knu4]:

$$\begin{aligned}
|\boldsymbol{x} - \boldsymbol{y}| &= \text{doz}(\boldsymbol{x}, \boldsymbol{y}) + \text{doz}(\boldsymbol{y}, \boldsymbol{x}), && \text{signed arguments,} \\
&= \text{dozu}(\boldsymbol{x}, \boldsymbol{y}) + \text{dozu}(\boldsymbol{y}, \boldsymbol{x}), && \text{unsigned arguments.}
\end{aligned}$$

Corollary: $|\boldsymbol{x}| = \text{doz}(\boldsymbol{x}, \mathbf{0}) + \text{doz}(\mathbf{0}, \boldsymbol{x})$ (other three-instruction solutions are given on page 17).

3. To clamp the upper limit of the true sum of unsigned integers $\boldsymbol{x}$ and $\boldsymbol{y}$ to the maximum positive number $(2^{32}-1)$ [Knu4]:

$$\neg\text{dozu}(\neg \boldsymbol{x}, \boldsymbol{y}).$$

4. Some comparison predicates (four instructions each):

$$\begin{aligned}\boldsymbol{x} > \boldsymbol{y} &= (\text{doz}(\boldsymbol{x}, \boldsymbol{y}) \mid -\text{doz}(\boldsymbol{x}, \boldsymbol{y})) \overset{u}{\gg} 31,\\ \boldsymbol{x} \overset{u}{>} \boldsymbol{y} &= (\text{dozu}(\boldsymbol{x}, \boldsymbol{y}) \mid -\text{dozu}(\boldsymbol{x}, \boldsymbol{y})) \overset{u}{\gg} 31.\end{aligned}$$

5. The carry bit from the addition $\boldsymbol{x} + \boldsymbol{y}$ (five instructions):

$$\text{carry}(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{x} \overset{u}{>} \neg\boldsymbol{y} = (\text{dozu}(\boldsymbol{x}, \neg\boldsymbol{y}) \mid -\text{dozu}(\boldsymbol{x}, \neg\boldsymbol{y})) \overset{u}{\gg} 31.$$

The expression doz($\boldsymbol{x}$, $-\boldsymbol{y}$), with the result interpreted as an unsigned integer, is in most cases the true sum $x + y$ with the lower limit clamped at 0. However, it fails if $\boldsymbol{y}$ is the maximum negative number.

The IBM RS/6000 computer, and its predecessor the 801, have the signed version of *difference or zero*. Knuth's MMIX computer [Knu4] has the unsigned version (including some varieties that operate on parts of words in parallel). This raises the question of how to get the signed version from the unsigned version, and vice versa. This can be done as follows (where the additions and subtractions simply complement the sign bit):

$$\begin{aligned}\text{doz}(\boldsymbol{x}, \boldsymbol{y}) &= \text{dozu}(\boldsymbol{x} + 2^{31}, \boldsymbol{y} + 2^{31}),\\ \text{dozu}(\boldsymbol{x}, \boldsymbol{y}) &= \text{doz}(\boldsymbol{x} - 2^{31}, \boldsymbol{y} - 2^{31}).\end{aligned}$$

Some other identities that may be useful are:

$$\begin{aligned}\text{doz}(\neg\boldsymbol{x}, \neg\boldsymbol{y}) &= \text{doz}(\boldsymbol{y}, \boldsymbol{x}),\\ \text{dozu}(\neg\boldsymbol{x}, \neg\boldsymbol{y}) &= \text{dozu}(\boldsymbol{y}, \boldsymbol{x}).\end{aligned}$$

The relation $\text{doz}(-\boldsymbol{x}, -\boldsymbol{y}) = \text{doz}(\boldsymbol{y}, \boldsymbol{x})$ fails if either $\boldsymbol{x}$ or $\boldsymbol{y}$, but not both, is the maximum negative number.

- - -

Page 44, add this new section at the end of Chapter 2:

2–21 A Boolean Decomposition Formula

In this section we have a look at the minimum number of binary Boolean operations, or instructions, that suffice to implement any Boolean function of three, four, or five variables. By a “Boolean function” we mean a Boolean-valued function of Boolean arguments.

Our notation for Boolean algebra uses “+” for *or*, juxtaposition for *and*, $\oplus$ for *exclusive or*, and either an overbar or a prefix $\neg$ for *not*. These operators can be applied to single-bit operands or “bitwise” to computer words. Our main result is the following theorem.

THEOREM. *If $f(x, y, z)$ is a Boolean function of three variables, then it can be decomposed into the form $g(x, y) \oplus zh(x, y)$, where g and h are Boolean functions of two variables.*[2]

Proof [Ditlow]. $f(x, y, z)$ can be expressed as a sum of minterms and then $\bar{z}$ and z can be factored out of their terms, giving

$$f(x, y, z) = \bar{z}f_0(x, y) + zf_1(x, y).$$

Because the operands to “+” cannot both be 1, the *or* can be replaced with *exclusive or*, giving

$$\begin{aligned} f(x, y, z) &= \bar{z}f_0(x, y) \oplus zf_1(x, y) \\ &= (1 \oplus z)f_0(x, y) \oplus zf_1(x, y) \\ &= f_0(x, y) \oplus zf_0(x, y) \oplus zf_1(x, y) \\ &= f_0(x, y) \oplus z(f_0(x, y) \oplus f_1(x, y)), \end{aligned}$$

where we have twice used the identity $(a \oplus b)c = ac \oplus bc$.

This is in the required form with $g(x, y) = f_0(x, y)$ and $h(x, y) = f_0(x, y) \oplus f_1(x, y)$. $f_0(x, y)$, incidentally, is $f(x, y, z)$ with $z = 0$, and $f_1(x, y)$ is $f(x, y, z)$ with $z = 1$.

COROLLARY. *If a computer’s instruction set includes an instruction for each of the 16 Boolean functions of two variables, then any Boolean function of three variables can be implemented with four (or fewer) instructions.*

One instruction implements $g(x, y)$, another implements $h(x, y)$, and these are combined with *and* and *exclusive or*.

As an example, consider the Boolean function which is 1 if exactly two of *x*, *y*, and *z* are 1:

$$f(x, y, z) = xy\bar{z} + x\bar{y}z + \bar{x}yz.$$

2. Logic designers will recognize this as Reed-Muller, aka positive Davio, decomposition. According to Knuth [], it was known to I. I. Zhegalkin [Matematicheskii Sbornik 35 (1928), 311-369]. It is sometimes referred to as the Russian decomposition.

Before proceeding, the interested reader might like to try to implement *f* with four instructions, without using the theorem.

From the proof of the theorem,

$$\begin{aligned} f(x, y, z) &= f_0(x, y) \oplus z(f_0(x, y) \oplus f_1(x, y)) \\ &= xy \oplus z(xy \oplus (x\bar{y} + \bar{x}y)) \\ &= xy \oplus z(x + y), \end{aligned}$$

which is four instructions.

Clearly the theorem can be extended to functions of four or more variables. That is, any Boolean function $f(x_1, x_2, \ldots, x_n)$ can be decomposed into the form $g(x_1, x_2, \ldots, x_{n-1}) \oplus x_n h(x_1, x_2, \ldots, x_{n-1})$. Thus a function of four variables can be decomposed as follows:

$$\begin{aligned} f(w, x, y, z) &= g(w, x, y) \oplus zh(w, x, y), \quad \text{where} \\ g(w, x, y) &= g_1(w, x) \oplus yh_1(w, x) \quad \text{and} \\ h(w, x, y) &= g_2(w, x) \oplus yh_2(w, x). \end{aligned}$$

This shows that a computer that has an instruction for each of the 16 binary Boolean functions can implement any function of four variables with ten instructions. Similarly, any function of five variables can be implemented with 22 instructions.

However, it is possible to do much better. For functions of four or more variables there is probably no simple plug-in equation like the theorem gives, but extensive computer searches have been done. The results are that any Boolean function of four variables can be implemented with seven binary Boolean instructions, and any such function of five variables can be implemented with 12 such instructions [Knu4].

In the case of five variables, only 1920 of the $2^{2^5} = 4{,}294{,}967{,}296$ functions require 12 instructions, and these 1920 functions are all essentially the same function. The variations are obtained by permuting the arguments, replacing some arguments with their complements, or complementing the value of the function.

Implementing Instructions for all 16 Binary Boolean Operations

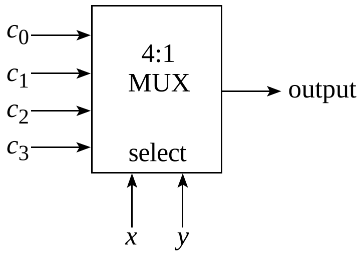
The instruction sets of some computers include all 16 binary Boolean operations. Many of the instructions are useless in that their function can be accomplished with another instruction. For example, the function $f(x, y) = 0$ simply clears a register, and most computers have a variety of ways to do that. But nevertheless, one reason a computer designer might choose to implement all 16 is that there is a simple and quite regular circuit for doing it.

Refer to Table 2-1 on page 17, which shows all 16 binary Boolean functions. To implement these functions as instructions, choose four of the opcode bits to be the same as the function values shown in the table. Denoting these opcode bits by c_0, c_1, c_2, and c_3, reading from the bottom up in the table, and the input registers by x and y, the circuit for implementing all 16 binary Boolean operations is described by the logic expression

$$c_0xy + c_1x\bar{y} + c_2\bar{x}y + c_3\bar{x}\bar{y}.$$

For example, with $c_0 = c_1 = c_2 = c_3 = 0$, the instruction computes the zero function, $f(x, y) = 0$. With $c_0 = 1$ and the other opcode bits 0 it is the *and* instruction. With $c_0 = c_3 = 0$ and $c_1 = c_2 = 1$ it is *exclusive or*, and so forth.

This can be implemented with n 4:1 MUXs, where n is the word size of the machine. The data bits of x and y are the select lines, and the four opcode bits are the data inputs to each MUX. The MUX is a standard building block in today's technology, and it is usually a very fast circuit. It is illustrated below.



The function of the circuit is to select c_0, c_1, c_2, or c_3 to be the output, depending on whether x and y are 00, 01, 10, or 11, respectively. It is like a four-position rotary switch.

Elegant as this is, it is somewhat expensive in opcode points, using 16 of them. There are a number of ways to implement all 16 Boolean operations using only eight opcode points, at the expense of less regular logic. One such scheme is illustrated in Table 2–3.

TABLE 2–3. EIGHT SUFFICIENT BOOLEAN INSTRUCTIONS

Function Values	**Formula**	**Instruction Mnemonic (Name)**
0001	xy	*and*
0010	$x\bar{y}$	*andc* (*and with complement*)
0110	$x \oplus y$	*xor* (*exclusive or*)
0111	$x + y$	*or*
1110	$\overline{xy}$	*nand* (*negative and*)
1101	$\overline{x\bar{y}}$, or $\bar{x} + y$	*cor* (*complement and or*)
1001	$\overline{x \oplus y}$, or $x \equiv y$	*eqv* (*equivalence*)
1000	$\overline{x + y}$	*nor* (*negative or*)

The eight operations not shown in the table can be done with the eight instructions shown, by interchanging the inputs or by having both register fields of the instruction refer to the same register. See exercise 12.

IBM's POWER architecture uses this scheme, with the minor difference that POWER has *or with complement* rather than *complement and or.* The scheme shown in Table 2–3 allows the last four instructions to be implemented by complementing the result of the first four instructions, respectively.

Historical Notes

The algebra of logic expounded in George Boole's *An Investigation of the Laws of Thought* (1854)[3] is somewhat different from what we know today as "Boolean algebra." Boole used the *integers* 1 and 0 to represent truth and falsity, respectively, and he showed how they could be manipulated with the methods of ordinary numerical algebra to formalize natural language statements involving "and," "or," and "except." He also used ordinary algebra to formalize statements in set theory involving intersection, union of disjoint sets, and complementation, and to formalize statements in probability theory, in which the variables take on real number values from 0 to 1. The work often deals with questions of philosophy, religion, and law.

Boole is regarded as a great thinker about logic because he formalized it, allowing complex statements to be manipulated mechanically and flawlessly with the familiar methods of ordinary algebra.

Skipping ahead in history, there are a few programming languages that include all 16 Boolean operations. IBM's PL/I (ca. 1966) includes a built-in function named BOOL. In BOOL(x, y, z), z is a bit string of length four (or converted to that if necessary), and x and y are bit strings of equal length (or converted to that if necessary). Argument z specifies the Boolean operation to be performed on x and y. Binary 0000 is the zero function, 0001 is xy, 0010 is $x\bar{y}$, and so forth.

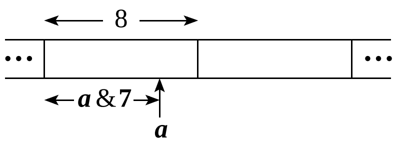
Another early implementation is in the Wang System 2200B computer (ca. 1974), which provided a version of Basic that included the BOOL function [Neum]. It operated on character strings rather than on bit strings or integers.

- - -

3. The entire 335-page work is available at www.gutenberg.org/etext/15114.

Page 50, add the following short paragraph and picture to the end of this section (3–3):

This formula can be easily understood from the figure below [Kumar], which illustrates that $\boldsymbol{a}\,\&\,\mathbf{7}$ is the offset of $\boldsymbol{a}$ in its block, and thus $\mathbf{8} - (\boldsymbol{a}\,\&\,\mathbf{7})$ is the space remaining in the block.



- - -

Page 68, insert the following paragraph after the one that ends “it does work for word lengths up to 62.”

The return statement in the code above can be replaced with the following, which runs faster on most machines but is perhaps less elegant (octal notation again).

```
return ((x * 0404040404) >> 26) +  // Add 6-bit sums.
        (x >> 30);
```

- - -

Pages 72-73, replace the entire subsection “**Counting 1-bits in an Array**,” including Figure 5–5, with the following material (three subsections):

Sum and difference of population counts of two words

To compute $\text{pop}(\boldsymbol{x}) + \text{pop}(\boldsymbol{y})$ (if your computer does not have the *population count* instruction), some time can be saved by using the first two lines of Figure 5–2 on $\boldsymbol{x}$ and $\boldsymbol{y}$ separately, and then adding $\boldsymbol{x}$ and $\boldsymbol{y}$ and executing the last three stages of the algorithm on the sum. After the first two lines of Figure 5–2 are executed, $\boldsymbol{x}$ and $\boldsymbol{y}$ consist of eight four-bit fields, each containing a maximum value of 4. Thus $\boldsymbol{x}$ and $\boldsymbol{y}$ may safely be added, because the maximum value in any four-bit field of the sum would be 8, so no overflow occurs. (In fact, three words may be combined in this way.)

This idea also applies to subtraction. To compute $\text{pop}(\boldsymbol{x}) - \text{pop}(\boldsymbol{y})$, use

$$\begin{aligned}\text{pop}(\boldsymbol{x}) - \text{pop}(\boldsymbol{y}) &= \text{pop}(\boldsymbol{x}) - (32 - \text{pop}(\bar{\boldsymbol{y}})) \\ &= \text{pop}(\boldsymbol{x}) + \text{pop}(\bar{\boldsymbol{y}}) - 32.\end{aligned}$$

Then, use the technique just described to compute $\text{pop}(\boldsymbol{x}) + \text{pop}(\bar{\boldsymbol{y}})$. The code is shown in Figure 5–5. It uses 32 instructions, vs. 43 for two applications of the code of Figure 5–2 followed by a subtraction.

```
int popDiff(unsigned x, unsigned y) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   y = ~y;
   y = y - ((y >> 1) & 0x55555555);
   y = (y & 0x33333333) + ((y >> 2) & 0x33333333);
   x = x + y;
   x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
   x = x + (x >> 8);
   x = x + (x >> 16);
   return (x & 0x0000007F) - 32;
}
```

FIGURE 5–5. Computing $\text{pop}(\boldsymbol{x}) - \text{pop}(\boldsymbol{y})$.

Comparing the Population Counts of Two Words

Sometimes one wants to know which of two words has the larger population count, without regard to the actual counts. Can this be determined without doing a population count of the two words? Computing the difference of two population counts as in Figure 5–5, and comparing the result to 0, is one way, but there is another way that is preferable if either the population counts are expected to be low, or if there is a strong correlation between the particular bits that are set in the two words.

The idea is to clear a single bit in each word until one of the words is all zero; the other word then has the larger population count. The process runs faster in its worst and average cases if the bits that are 1 at the same positions in each word are first cleared. The code is shown in Figure 5–6. The procedure returns a negative integer if $\text{pop}(\boldsymbol{x}) < \text{pop}(\boldsymbol{y})$, 0 if $\text{pop}(\boldsymbol{x}) = \text{pop}(\boldsymbol{y})$, and a positive integer (1) if $\text{pop}(\boldsymbol{x}) > \text{pop}(\boldsymbol{y})$.

```
int popCmpr(unsigned xp, unsigned yp) {
   unsigned x, y;
   x = xp & ~yp;                // Clear bits where
   y = yp & ~xp;                // both are 1.
   while (1) {
      if (x == 0) return y | -y;
      if (y == 0) return 1;
      x = x & (x - 1);          // Clear one bit
      y = y & (y - 1);          // from each.
   }
}
```

FIGURE 5–6. Comparing $\text{pop}(\boldsymbol{x})$ with $\text{pop}(\boldsymbol{y})$.

After clearing the common 1-bits in each 32-bit word, the maximum possible number of 1-bits in both words together is 32. Therefore the word with the smaller number of 1-bits can have at most 16. Thus the loop in Figure 5–6 is executed a maximum of 16 times, which gives a worst case of 119 instructions executed on the basic RISC $(16 \cdot 7 + 7)$. A simulation using uniformly distributed random 32-bit integers showed that the average population count of the word with the smaller population count is approximately 6.186, after clearing the common 1-bits. This gives an average execution time of about 50 instructions executed for random 32-bit inputs, not as good as using Figure 5–5. For this procedure to beat that of Figure 5–5, the number of 1-bits in either x or y, after clearing the common 1-bits, would have to be three or less.

Counting the 1-bits in an Array

The simplest way to count the number of 1-bits in an array (vector) of fullwords, in the absence of the population count instruction, is to use a procedure such as that of Figure 5–2 on page 66 on each word of the array, and simply add the results. We call this the “naive” method. Ignoring loop control, the generation of constants, and loads from the array, it takes 16 instructions per word: 15 for the code of Figure 5–2 plus one for the addition. We assume the procedure is expanded in line, the masks are loaded outside of the loop, and the machine has a sufficient number of registers to hold all the quantities used in the calculation.

Another way is to use the first two executable lines of Figure 5–2 on groups of three words in the array, adding the three partial results. Because each partial result has a maximum value of 4 in each four-bit field, the sum of the three has a maximum value of 12 in each four-bit field, so no overflow occurs. This idea can be applied to the 8- and 16-bit fields. Coding and compiling this method indicates that it gives about a 20% reduction over the naive method in total number of instructions executed on the basic RISC. Much of the savings are cancelled by the additional housekeeping instructions required. We will not dwell on this method because there is a *much* better way to do it.

The better way seems to have been invented by Robert Harley and David Seal in about 1996 [Seal1]. It is based on a circuit called a *carry-save adder* (CSA), or 3:2 compressor. A CSA is simply a sequence of independent full adders[4] [H&P], and it is often used in binary multiplier circuits.

In Boolean algebra notation, the logic for each full adder is

$$\begin{aligned} h &\leftarrow ab + ac + bc = ab + (a + b)c = ab + (a \oplus b)c, \\ l &\leftarrow (a \oplus b) \oplus c. \end{aligned}$$

where a, b, and c are the 1-bit inputs, l is the low-bit output (sum) and h is the high-bit output (carry). Changing $a + b$ on the first line to $a \oplus b$ is justified

4. A full adder is a circuit with three 1-bit inputs (the bits to be added) and two 1-bit outputs (the sum and carry).

because when a and b are both 1, the term ab makes the value of the whole expression 1. By first assigning $a \oplus b$ to a temporary, the full adder logic can be evaluated in five logical instructions, each operating on 32 bits in parallel (on a 32-bit machine). We will refer to these five instructions as CSA(h, l, a, b, c). This is a "macro," with h and l being outputs.

One way to use the CSA operation is to process elements of the array A in groups of three, reducing each group of three words to two, and applying the population count operation to these two words. In the loop, these two population counts are summed. After executing the loop, the total population count of the array is twice the accumulated population count of the CSA's high-bit outputs plus the accumulated population count of the low-bit outputs.

Let n_c be the number of instructions required for the CSA steps, and n_p be the number of instructions required to do the population count of one word. On a typical RISC machine $n_c = 5$ and $n_p = 15.$ Ignoring loads from the array and loop control (the code for which may vary quite a bit from one machine to another), the loop discussed above takes $(n_c + 2n_p + 2)/3 \approx 12.33$ instructions per word of the array (the "+ 2" is for the two additions in the loop). This is in contrast to the 16 instructions per word required by the naive method.

There is another way to use the CSA operation that results in a program that's more efficient and slightly more compact. This is shown in Figure 5–7. It takes

$(n_c + n_p + 1)/2 = 10.5$ instructions per word (ignoring loop control and loads).

```
#define CSA(h,l, a,b,c) \
   {unsigned u = a ^ b; unsigned v = c; \
      h = (a & b) | (u & v); l = u ^ v;}

int popArray(unsigned A[], int n) {

   int tot, i;
   unsigned ones, twos;

   tot = 0;                         // Initialize.
   ones = 0;
   for (i = 0; i <= n - 2; i = i + 2) {
      CSA(twos, ones, ones, A[i], A[i+1])
      tot = tot + pop(twos);
   }
   tot = 2*tot + pop(ones);

   if (n & 1)                       // If there's a last one,
      tot = tot + pop(A[i]);        // add it in.

   return tot;
}
```

FIGURE 5–7. Array population count, processing elements in groups of two.

In this code, the CSA operation expands into:

```
   u = ones ^ A[i];
   v = A[i+1];
   twos = (ones & A[i]) | (u & v);
   ones = u ^ v;
```

The code relies on the compiler to common the loads.

There are ways to use the CSA operation to further reduce the number of instructions required to compute the population count of an array. They are most easily understood by means of a circuit diagram. For example, Figure 5–8 illustrates a way to code a loop that takes array elements eight at a time and compresses them into four quantities, labelled *eights*, *fours*, *twos*, and *ones*. The *fours*, *twos*, and *ones* are fed back into the CSAs on the next loop iteration, and the 1-bits in *eights* are counted by an execution of the word level population count function, and this count is accumulated. When all of the array has been processed, the total population count is

$$8\text{pop}(eights) + 4\text{pop}(fours) + 2\text{pop}(twos) + \text{pop}(ones).$$

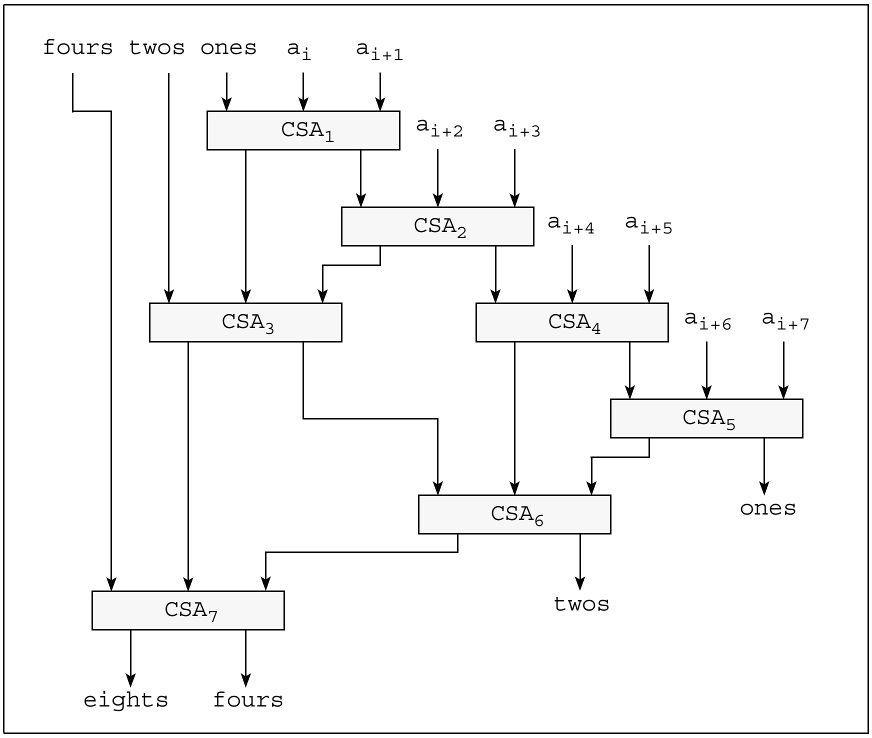


FIGURE 5–8. A circuit for the array population count.

The code is shown in Figure 5–9, which uses the CSA macro defined in Figure 5–7. The numbering of the CSA blocks in Figure 5–8 corresponds to the order of the CSA macro calls in Figure 5–9. The execution time of the loop, exclusive of array loads and loop control, is $(7n_c + n_p + 1)/8 = 6.375$ instructions per word of the array.

The CSAs may be connected in many arrangements other than that shown in Figure 5–8. For example, increased parallelism might result from feeding the first three array elements into one CSA, and the next three into a second CSA, which allows the instructions of these two CSAs to execute in parallel. One might also be able to permute the three input operands of the CSA macros for increased parallelism. With the plan shown in Figure 5–8, one can easily see how to use only the first three CSAs to construct a program that processes array elements in groups of four, and also how to expand it to construct programs that process array elements in groups of 16 or more. The plan shown also spreads out the loads somewhat, which would be advantageous for a machine that has a relatively low limit on the number of loads that can be outstanding at any one time.

The plan of Figure 5–8 can be generalized so that very few word population counts are done. To sketch how this program might be constructed, it needs an array of m×2 words to hold two of each of the variables we have called *ones*, *twos*, *fours*, and so forth. For an array of size *n*, choosing $m \geq \lceil \log_2(n+1) \rceil + 1$ is suf-

```
int popArray(unsigned A[], int n) {

   int tot, i;
   unsigned ones, twos, twosA, twosB,
      fours, foursA, foursB, eights;

   tot = 0;                        // Initialize.
   fours = twos = ones = 0;

   for (i = 0; i <= n - 8; i = i + 8) {
      CSA(twosA, ones, ones, A[i], A[i+1])
      CSA(twosB, ones, ones, A[i+2], A[i+3])
      CSA(foursA, twos, twos, twosA, twosB)
      CSA(twosA, ones, ones, A[i+4], A[i+5])
      CSA(twosB, ones, ones, A[i+6], A[i+7])
      CSA(foursB, twos, twos, twosA, twosB)
      CSA(eights, fours, fours, foursA, foursB)
      tot = tot + pop(eights);
   }
   tot = 8*tot + 4*pop(fours) + 2*pop(twos) + pop(ones);

   for (i = i; i < n; i++)         // Simply add in the last
      tot = tot + pop(A[i]);       // 0 to 7 elements.
   return tot;
}
```

FIGURE 5–9. Array population count, processing elements in groups of eight.

ficient ($m = 31$ is sufficient for any size array that can be held in a machine with a 32-bit byte-addressed space). A byte array of size m is also needed to keep track of how many (0, 1, or 2) values are currently in each row of the $m\times 2$ array. The program processes array elements in groups of two. For each group, the CSA is invoked to compress those two array elements with a saved value of *ones*, which is most conveniently kept in the [0,0] position of the $m\times 2$ array. In an inner loop, the resulting *twos* is saved in the array, by scanning down (usually not far at all) to find a row with fewer than two items. If the *twos* row is full, its two values are combined with *twos* (using the CSA). The *twos* output is put in the array, resetting its row count to 1. The scan continues with the *fours* output to find a place to put it, and so forth.

After completing the pass over the input array, the program next makes a pass over the (much shorter) $m\times 2$ array, compressing all full rows, so that all rows contain only one significant value. Lastly, the program invokes the word level population count operation on the first element of each row until a row with a zero count is encountered, computing the total array population count as

$$\text{pop(row 0)} + 2\text{pop(row 1)} + 4\text{pop(row 2)} + \ldots.$$

The value suggested above for m ensures that the last row will have a zero count, which may be used to terminate the scans.

The resulting program executes exactly $\lceil \log_2(n + 3) \rceil$ word population counts. Unfortunately it is not practical because the housekeeping steps for loading from and storing into the intermediate result arrays outweigh the computational instructions that are saved. An experimental program (without trying too hard to optimize it) ran in about 29 instructions per array word (counting all instructions in the loop). This is significantly worse than the naive method.

Table 5–4 summarizes the number of instructions executed by this plan for various group sizes. The values in the middle two columns ignore loads and loop control. The third column gives the total loop instruction execution count, per word of the input array, produced by a compiler for the basic RISC machine (which does not have indexed loads).

TABLE 5–4. INSTRUCTIONS PER WORD FOR THE ARRAY POPULATION COUNT

	Instructions exclusive of loads and loop control		**All instructions in loop (compiler output)**
Program	**Formula**	**For $n_c = 5$, $n_p = 15$**	
Naive method	$n_p + 1$	16	21
Groups of 2	$(n_c + n_p + 1)/2$	10.5	14
Groups of 4	$(3n_c + n_p + 1)/4$	7.75	10
Groups of 8	$(7n_c + n_p + 1)/8$	6.38	8
Groups of 16	$(15n_c + n_p + 1)/16$	5.69	7
Groups of 32	$(31n_c + n_p + 1)/32$	5.34	6.5
Groups of 2^n	$n_c + \dfrac{n_p - n_c + 1}{2^n}$	$5 + \dfrac{11}{2^n}$	–

For small arrays, there are better plans than that of Figure 5–8. For example, for an array of seven words, the plan of Figure 5–10 is quite efficient [Seal1]. It executes in $4n_c + 3n_p + 4 = 69$ instructions, or 9.86 instructions per word. Similar plans exist that apply to arrays of size $2^k - 1$ words for any positive integer k.

The plan for 15 words executes in $11n_c + 4n_p + 6 = 121$ instructions, or 8.07 instructions per word.

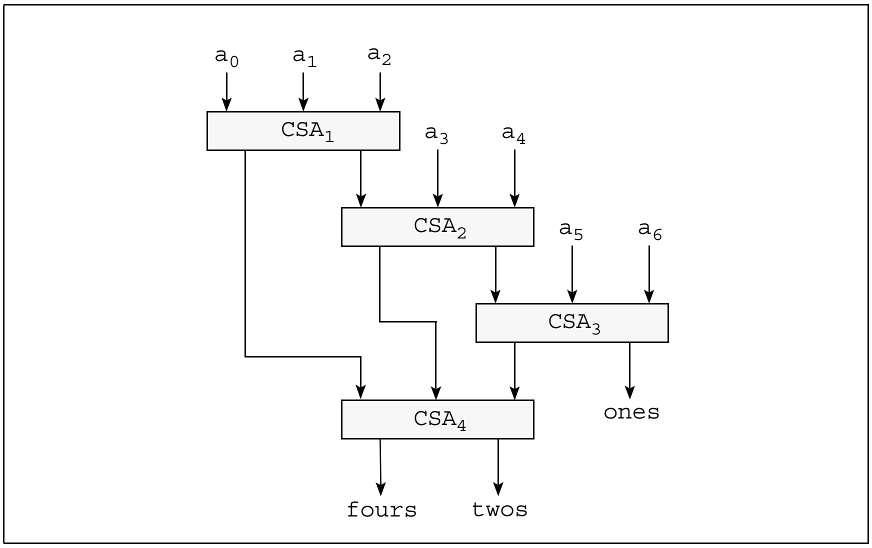


FIGURE 5–10. A circuit for the total population count of seven words.

- - -

Page 74, middle, after the line “The cost of this representation …”, insert the following paragraph:

The population function can be used to generate binomially distributed random integers. To generate an integer drawn from a population given by BINOMIAL(t, p) where t is the number of trials and $p = 1/2$, generate t random bits and count the number of 1’s in the t bits. This can be generalized to probabilities p other than 1/2; see for example [Knu2, sec. 3.4.1, prob. 27].

- - -

Page 74, middle, after the paragraph “Still another application …”, insert the following paragraph:

According to computer folklore, the population count function is important to the National Security Agency. No one (outside of NSA) seems to know just what they use it for, but it may be in cryptography work or in searching huge amounts of material.

- - -

Page 75, at the bottom of the page, insert the following paragraph:

The five assignments in (3) can be done in any order (provided variable x is used in the first one). If they are done in reverse order, and if you are interested only in getting the parity in the low-order bit of y, then the last two lines:

```
y = y ^ (y >> 2);
y = y ^ (y >> 1);
```

can be replaced with [Huef]

```
y = 0x6996 >> (y & 0xF);
```

This is an “in-register table lookup” operation. On the basic RISC it saves one instruction, or two if the load of the constant is not counted. The low-order bit of y has the original word’s parity, but the other bits of y do not contain anything useful.

- - -

Page 76, after line 13 (“the machine has *remainder immediate*”), insert the following paragraph:

On a 64-bit machine, the above code employing multiplication gives the correct result after making the obvious changes (expand the hex constants to 16 nibbles each with value 1, and change the final shift amount from 28 to 60). In this case the maximum sum in any four-bit column of the partial products, other than the most significant column, is 15, so again no overflow occurs that affects the result in the most significant column. However, the variation that computes the remainder upon division by 15 does *not* work on a 64-bit machine, because the remainder result is the sum of the nibbles modulo 15, and the sum may be as high as 16.

- - -

Page 81, top (just before **Floating-Point Methods**), insert the following material:

Robert Harley [Harley] devised an algorithm for nlz(x) that is very similar to Seal’s algorithm for ntz(x) (described in Section 5–4). Harley’s method propagates the most significant 1-bit to the right using *shift*’s and *or*’s, and multiplies modulo 2^{32} by a special constant, producing a product whose high-order 6 bits uniquely identify the number of leading 0’s in x. It then does a *shift right* and a table lookup (indexed load) to translate the six-bit identifier to the actual number of leading 0’s. As shown in Figure 5–12, it consists of 14 instructions, including a *multiply*, plus an indexed load. Table entries shown as u are unused.

```
int nlz(unsigned x) {

   static char table[64] =
     {32,31, u,16, u,30, 3, u,  15, u, u, u,29,10, 2, u,
       u, u,12,14,21, u,19, u,   u,28, u,25, u, 9, 1, u,
      17, u, 4, u, u, u,11, u,  13,22,20, u,26, u, u,18,
       5, u, u,23, u,27, u, 6,   u,24, 7, u, 8, u, 0, u};

   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*0x06EB14F9;     // Multiplier is 7*255**3.
   return table[x >> 26];
}
```

FIGURE 5–12. *Number of leading zeros*, Harley's algorithm.

The multiplier is $7 \cdot 255^3$, so the multiplication may be done as shown below. In this form the function consists of 19 elementary instructions plus an indexed load.

```
   x = (x << 3) - x;     // Multiply by 7.
   x = (x << 8) - x;     // Multiply by 255.
   x = (x << 8) - x;     // Again.
   x = (x << 8) - x;     // Again.
```

There are many multipliers that have the desired uniqueness property and whose factors are all of the form $2^k \pm 1$. The smallest is 0x045BCED1 = $17 \cdot 65 \cdot 129 \cdot 513$. There are no such multipliers consisting of three factors, if the table size is 64 or 128 entries. If the table size is 256 entries, however, there are a number of such multipliers. The smallest is 0x01033CBF = $65 \cdot 255 \cdot 1025$ (using this would save two instructions at the expense of a larger table).

Julius Goryavsky [Gor] has found several variations of Harley's algorithm that reduce the table size at the expense of a few instructions, or have improved parallelism, and so on. One, shown in Figure 5–13, is a clear winner if the multiplication is done with shifts and adds. The code changes only the table and the lines that contain the *shift right* of 16 and the following *multiply* in Figure 5–12. If the machine has *and not*, this saves two instructions because the multiplier can be factored as $511 \cdot 2047 \cdot 16383 \pmod{2^{32}}$, which can be done in six elementary instructions rather than eight. If the machine does not have *and not*, it saves one instruction.

```
...
static 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 >> 16);
x = x*0xFD7049FF;
...
```

FIGURE 5–13. *Number of leading zeros,* Goryavsky's variation of Harley's algorithm.

- - -

Page 83, near the top of the page, before "**Relation to the Log Function**," insert the following subsection.

Comparing the Number of Leading Zeros of Two Words

There is a simple way to determine which of two words $\boldsymbol{x}$ and $\boldsymbol{y}$ has the larger number of leading zeros [Knu6] without actually computing nlz($\boldsymbol{x}$) or nlz($\boldsymbol{y}$). The methods are shown in the equivalences below. The three relations not shown are of course obtained by complementing the sense of the comparison on the right.

$$\begin{aligned}
\text{nlz}(\boldsymbol{x}) = \text{nlz}(\boldsymbol{y}) &\quad \text{if and only if} \quad (\boldsymbol{x} \oplus \boldsymbol{y}) \overset{u}{\leq} (\boldsymbol{x} \,\&\, \boldsymbol{y}) \\
\text{nlz}(\boldsymbol{x}) < \text{nlz}(\boldsymbol{y}) &\quad \text{if and only if} \quad (\boldsymbol{x} \,\&\, \neg\boldsymbol{y}) \overset{u}{>} \boldsymbol{y} \\
\text{nlz}(\boldsymbol{x}) \leq \text{nlz}(\boldsymbol{y}) &\quad \text{if and only if} \quad (\boldsymbol{y} \,\&\, \neg\boldsymbol{x}) \overset{u}{\leq} \boldsymbol{x}
\end{aligned}$$

- - -

Page 83, just before "**Applications**," insert the following:

An alternative, which is the same function as bitsize(x) except it gives the result 0 for $x = 0$, is

```
32 - nlz(x ^ (x << 1))
```

- - -

Page 86, just after Figure 5-16, insert the following material.

Dean Gaudet [Gaud] devised an algorithm that is interesting because with the right instructions it is branch-free, load-free (does not use table lookup), and has lots of parallelism. It is shown in Figure 5-17.

```
int ntz(unsigned x) {
   unsigned y, bz, b4, b3, b2, b1, b0;

   y = x & -x;                // Isolate rightmost 1-bit.
   bz = y ? 0 : 1;            // 1 if y = 0.
   b4 = (y & 0x0000FFFF) ? 0 : 16;
   b3 = (y & 0x00FF00FF) ? 0 : 8;
   b2 = (y & 0x0F0F0F0F) ? 0 : 4;
   b1 = (y & 0x33333333) ? 0 : 2;
   b0 = (y & 0x55555555) ? 0 : 1;
   return bz + b4 + b3 + b2 + b1 + b0;
}
```

FIGURE 5–17. *Number of trailing zeros,* Gaudet's algorithm.

As shown, the code uses the C "conditional expression" in six places. This construct has the form `a?b:c`. Its value is `b` if `a` is **true** (nonzero), and `c` if `a` is **false** (zero). Although a conditional expression must, in general, be compiled into compares and branches, for the simple cases in Figure 5–17 branching may be avoided if the machine has a *compare for equality to zero* instruction that sets a target register to 1 if the operand is 0, and to 0 if the operand is nonzero. Branching may also be avoided by using *conditional move* instructions. Using *compare*, the assignment to `b3` can be compiled into five instructions on the basic RISC: two to generate the hex constant, an *and*, the *compare*, and a *shift left* of 3. (The first, second, and last conditional expressions require one, three, and four instructions, respectively.)

The code can be compiled into a total of 30 instructions. All six lines with the conditional expressions can run in parallel. On a machine with a sufficient degree of parallelism, it executes in ten cycles. Present machines don't have that much parallelism, so as a practical matter it might help to change the first two uses of `y` in the program to `x`. This permits the first three executable statements to run in parallel.

David Seal [Seal2] devised an algorithm for computing ntz($\boldsymbol{x}$) that is based on the idea of compressing the 2^{32} possible values of $\boldsymbol{x}$ to a small dense set of integers, and doing a table lookup. He uses the expression $\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}$ to reduce the number of possible values to a small number. The value of this expression is a word that contains a single 1-bit at the position of the least significant 1-bit in $\boldsymbol{x}$, or is 0 if $\boldsymbol{x} = 0$. Thus $\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}$ has only 33 possible values. But they are not dense; they range from 0 to 2^{31}.

To produce a dense set of 33 integers which uniquely identify the 33 values of $\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}$, Seal found a certain constant which, when multiplied by $\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}$, produces the identifying value in the high-order six bits of the low-order half of the product of the constant and $\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}$. Since $\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}$ is an integral power of 2 or is 0, the multiplication amounts to a left shift of the constant, or it is a multiplication by 0. Using only the high-order five bits is not sufficient, because 33 distinct values are needed.

The code is shown in Figure 5–18, where table entries shown as u are unused.

```
int ntz(unsigned x) {

   static char table[64] =
     {32, 0, 1,12, 2, 6, u,13,   3, u, 7, u, u, u, u,14,
      10, 4, u, u, 8, u, u,25,   u, u, u, u, u,21,27,15,
      31,11, 5, u, u, u, u, u,   9, u, u,24, u, u,20,26,
      30, u, u, u, u,23, u,19,  29, u,22,18,28,17,16, u};

   x = (x & -x)*0x0450FBAF;
   return table[x >> 26];
}
```

FIGURE 5–18. *Number of trailing zeros,* Seal's algorithm.

As an example, if x is an odd multiple of 16, then x & -x = 16, so the multiplication amounts to a left shift of four positions. The high-order six bits of the low-order half of the product are then binary 010001, or 17 decimal. The table translates 17 to 4, which is the correct number of trailing 0's for an odd multiple of 16.

There are thousands of constants that have the necessary uniqueness property. The smallest is 0x0431472F, and the largest is 0xFDE75C6D. Seal chose a constant for which the multiplication can be done with a small number of shifts and adds. Since 0x0450FBAF = $17 \cdot 65 \cdot 65535,$ the multiplication can be done as follows:

```
   x = (x << 4) + x;     // x = x*17.
   x = (x << 6) + x;     // x = x*65.
   x = (x << 16) - x;    // x = x*65535.
```

With this substitution, the code of Figure 5–18 consists of nine elementary instructions plus an indexed *load*. Seal was interested in the ARM instruction set, which can do a *shift* and *add* in one instruction. On that architecture, the code is six instructions including the indexed load.

To make the multiplication even easier to do with shifts and adds, one might hope to find a constant of the form $(2^{k_1} \pm 1)(2^{k_2} \pm 1)$ that has the necessary uniqueness property. For a table size of 64, there are no such integers, and there is only one other suitable integer that is a product of three such factors: 0x08A1FBAF = $17 \cdot 65 \cdot 131071.$ Using a table size of 128 or 256 does not help. However, for a table size of 512 there are four suitable integers of the form $(2^{k_1} \pm 1)(2^{k_2} \pm 1);$ the smallest is 0x0080FF7F = $129 \cdot 65535.$ We leave it to the reader to determine the table associated with this constant.

There is a variation of Seal's method that is based on de Bruijn cycles [LPR]. These are cyclic sequences over a given alphabet that contain as a subsequence every sequence of the letters of the alphabet of a given length exactly once. For example, a cycle that contains as a subsequence every sequence of $\{a, b, c\}$ of length 2 is *aabacbbcc*. Notice that the sequence *ca* wraps around from the end to the beginning. If the alphabet size is k and the length is n, there are k^n sequences. For a cycle to contain all of these, it must be of length at least k^n, which would be its length if a different sequence started at each position. It can be shown that there is always a cycle of this minimum possible length that contains all k^n sequences.

For our purposes, the alphabet is {0, 1} and, for dealing with 32-bit words, we are interested in a cycle that contains all 32 sequences 00000, 00001, 00010, …, 11111. Given such a cycle that begins with at least four 0's, we can compute ntz(x) by first reducing x to a word that contains a single bit at the position of the least significant bit of x, as in Seal's algorithm. Then by multiplication we can select a five-bit field of the de Bruijn cycle, which will be a unique value for each multiplier. This can be mapped to give the number of trailing 0's by a table lookup. The algorithm follows. The de Bruijn cycle used is

```
0000 0100 1101 0111 0110 0101 0001 1111.
```

It is in effect a cycle because in use it has trailing 0's beyond the 32 bits shown above, which is effectively the same as wrapping to the beginning.

```
int ntz(unsigned x) {

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

   if (x == 0) return 32;
   x = (x & -x)*0x04D7651F;
   return table[x >> 27];
}
```

FIGURE 5–19. *Number of trailing zeros* using a de Bruijn cycle.

There are 33 possible values of ntz(x), and only 32 five-bit subsequences in the de Bruijn cycle. Therefore two words with different values of ntz(x) must map to the same number by the table lookup. The words that conflict are zero and words that end with a 1-bit. To resolve this, the code has a test for 0 and returns 32 in that case. A branch-free way to resolve it, useful if your computer has predicate comparison instructions, is to change the last statement to

```
   return table[x >> 27] + 32*(x == 0);
```

To compare the two algorithms, Seal's does not require the test for zero and it allows the alternative of doing the multiplication with six elementary instructions. The de Bruijn algorithm uses a smaller table. The de Bruijn cycle used in Figure 5–19, discovered by Danny Dubé [Dubé], is a good one because multiplication by it can be done with eight elementary instructions. The constant is

0x04D7651F = (2047 · 5 · 256 + 1) · 31, from which one can see the shifts, adds, and subtracts that do the job.

John Reiser [Reiser] observed that there is another way to map the 33 values of the factor `x & -x` in Seal's algorithm to a dense set of unique integers: divide and use the remainder. The smallest divisor that has the necessary uniqueness property is 37. The resulting code is shown in Figure 5–20, where table entries shown as `u` are unused.

```
int ntz(unsigned x) {

   static char table[37] = {32,  0,  1, 26,  2, 23, 27,
                  u,  3, 16, 24, 30, 28, 11,  u, 13,  4,
                  7, 17,  u, 25, 22, 31, 15, 29, 10, 12,
                  6,  u, 21, 14,  9,  5, 20,  8, 19, 18};

   x = (x & -x)%37;
   return table[x];
}
```

FIGURE 5–20. *Number of trailing zeros*, Reiser's algorithm.

- - -

Page 92, just before the last paragraph ("If the "nlz" instruction …"), insert the following material (thanks to Norbert Juffa for this):

A method similar to that of Figure 6–2 but for finding the *rightmost* 0-byte in a word `x` (zbyter(x)) is [Mycro]:

```
y = (x - 0x01010101) & ~x & 0x80808080;
n = ntz(y) >> 3;
```

This executes in only five instructions exclusive of loading the constants, if the machine has the *and not* and *number of trailing zeros* instructions. It cannot be used to compute zbytel(x) because of a problem with borrows. It would be most useful for finding the first 0-byte in a character string on a little-endian machine, or to simply test for a 0-byte (using only the assignment to `y`) on a machine of either endianness.

- - -

Page 99 (end of Chapter 6), insert the following two sections:

6–3 Find Longest String of 1-Bits

The nicely concise function below returns the length of the longest string of 1-bits in x [Hsieh].

```
int maxstr1(unsigned x) {
   int k;
   for (k = 0; x != 0; k++) x = x & 2*x;
   return k;
}
```

FIGURE 6–6. Find length of longest string of 1's.

It executes in $4n + 3$ instructions on the basic RISC, where n is the length of the longest string of 1's, or 131 instructions in the worst case.

To reduce the worst case execution time, a "logarithmic" version is possible. It works by propagating 0's 1, 2, 4, 8, and 16 positions to the left, stopping at the last nonzero word, and then backtracking to find the length of the longest contiguous string of 1's.

For example, suppose

```
       x = 0011 1111 1111 0011 1111 0011 1111 1000
```

Then

```
      x2 = 0011 1111 1110 0011 1110 0011 1111 0000
      x4 = 0011 1111 1000 0011 1000 0011 1100 0000
      x8 = 0011 1000 0000 0000 0000 0000 0000 0000
     x16 = all 0's
```

In this case the last nonzero word is x8. Observe that each 1-bit in x8 indicates the leftmost position of a string of eight 1's. Thus the longest string of 1's begins at the leftmost position of a 1-bit in x8, bit position 29 in the example. To test for a string of length 12, one can test the bit at position 21 (29 – 8) in x4. Since that is 0, there is no string of length 12. To test for a string of length 10, one can test the bit at position 21 in x2. Since that is 1, position 29 is the start of a string of length 10 (or more). Lastly, to test for a string of length 11, one can test the bit at position 19 (21 – 2) in x. Since that is 0, the longest string is of length 10, and it starts at position 29.

This scheme is coded in Figure 6–7, except the code uses only two variables, x and y, instead of the five variables x, x2, x4, x8, and x16. This code finds both the length and position of the longest string of 1's, with the position being measured from the left end of the string. The scheme does not work if x is 0 or all

1’s. These are special-cased, with the latter possibility being handled in a place that is not executed frequently.

```
int fmaxstr1(unsigned x, int *apos) {
   unsigned  y;
   int s;

   if (x == 0) {*apos = 32; return 0;}
   y = x & (x << 1);
   if (y == 0) {s = 1; goto L1;}
   x = y & (y << 2);
   if (x == 0) {s = 2; x = y; goto L2;}
   y = x & (x << 4);
   if (y == 0) {s = 4; goto L4;}
   x = y & (y << 8);
   if (x == 0) {s = 8; x = y; goto L8;}
   if (x == 0xFFFF8000) {*apos = 0; return 32;}
   s = 16;

L16: y = x & (x << 8);
     if (y != 0) {s = s + 8; x = y;}
L8:  y = x & (x << 4);
     if (y != 0) {s = s + 4; x = y;}
L4:  y = x & (x << 2);
     if (y != 0) {s = s + 2; x = y;}
L2:  y = x & (x << 1);
     if (y != 0) {s = s + 1; x = y;}
L1:  *apos = nlz(x);
   return s;
}
```

FIGURE 6–7. Find length and position of longest string of 1’s.

The worst case execution time on the basic RISC is 39 instructions plus those required for the nlz function. If only the length of the longest string of 1’s is wanted, there is no significant savings in execution time, except for omitting the use of the nlz function.

6–4 Find Shortest String of 1-Bits

It is more difficult to find the *shortest* string of 1-bits in a word. One way to do it is to mark the beginnings of all strings of 1’s in a word `b`, and the ends of all such strings in a word `e`. Then, if `b & e` is nonzero, the shortest string is of length 1. Otherwise, shift `e` left one position, and test again. For example, if

```
x = 0011 1111 1111 0011 1111 0011 1111 1000
```

Then

```
b = 0010 0000 0000 0010 0000 0010 0000 0000
e = 0000 0000 0001 0000 0001 0000 0000 1000
```

After shifting `e` left five places, `b & e` is nonzero. This means that the shortest string of 1-bits is of length 6.

This idea is embodied in the code shown in Figure 6–8. As in the preceding material, the position of the string is measured from the left, and if there are two or more minimal length strings of equal length, this function finds the leftmost one. For example, if `x` = 0x00FF0FF0 it returns length 8, position 8.

The function executes in $8 + 4n$ instructions on the basic RISC (without *andc*), plus the time for the nlz function, for $n \geq 2$, where n is the length of the shortest contiguous string of 1's in `x`.

```
int fminstr1(unsigned x, int *apos) {
   int k;
   unsigned b, e;       // Beginnings, ends.

   if (x == 0) {*apos = 32; return 0;}
   b = ~(x >> 1) & x;   // 0-1 transitions.
   e = x & ~(x << 1);   // 1-0 transitions.
   for (k = 1; (b & e) == 0; k++)
      e = e << 1;
   *apos = nlz(b & e);
   return k;
}
```

FIGURE 6–8. Find length and position of shortest string of 1's.

Perhaps the ultimate problem in this class is to find the length and position of the shortest string of 1's in x that is at least as long as a given integer $n > 0$. In terms of the storage allocation problem, this is a "best fit" algorithm. This may be done by first left-propagating the 0's in x by $n - 1$ positions, and then finding the shortest string of 1's in the revised x. See the exercises.

- - -

Page 101, third paragraph:

Replace the phrase "A small improvement results on most machines …" by "A small improvement may result on some machines …."

- - -

Page 102, insert the following material after Figure 7–1:

The next algorithm, by Christopher Strachey [Strach 1961], is old by computer standards, but it is instructive. It reverses the rightmost 16 bits of a word, assuming the leftmost 16 bits are clear at the start, and places the reversed halfword in the left half of the register.

Its operation is based on the number of bit positions that each bit must move. The 16 bits, taken from left-to-right, must move 1, 3, 5, …, 31 positions. The bits that must move 16 or more positions are moved first, then those that must move eight or more positions, and so forth. Operation is illustrated below, where each letter denotes a single bit, and a period denotes a “don’t care” bit.

```
0000 0000 0000 0000 abcd efgh ijkl mnop  Given
0000 0000 ijkl mnop abcd efgh .... ....  After shl 16
0000 mnop ijkl efgh abcd .... .... ....  After shl 8
00op mnkl ijgh efcd ab.. .... .... ....  After shl 4
0pon mlkj ihgf edcb a... .... .... ....  After shl 2
ponm lkji hgfe dcba .... .... .... ....  After shl 1
```

Straightforward code consists of 16 basic RISC instructions plus 12 to load the constants:

```
x = x | ((x & 0x000000FF) << 16);
x = (x & 0xF0F0F0F0) | ((x & 0x0F0F0F0F) << 8);
x = (x & 0xCCCCCCCC) | ((x & 0x33333333) << 4);
x = (x & 0xAAAAAAAA) | ((x & 0x55555555) << 2);
x = x << 1;
```

Complementation may be used to reduce the number of distinct masks. By using more irregular masks, the rightmost 16 bits can be preserved.

If rotate shifts are available, Strachey’s idea can be used to reverse a 32-bit word. The idea is to consider how many bit positions each bit must move rotationally to the left to get to its final position. Taking the bits from left-to-right, the shift amounts are 1, 3, 5, …, 31, 1, 3, 5, …, 31 (no bit moves an even number of positions). The algorithm first rotate-moves those bits that must move 16 or more positions, then those that must move eight or more positions, and so forth, and finally those that must move one position (which is all of the bits, because all move amounts are odd). This scheme is shown below, for reversing a 32-bit word `x`. Function `shlr(x, y)` rotates `x` left `y` positions.

```
x = shlr(x & 0x00FF00FF, 16) | x & ~0x00FF00FF;
x = shlr(x & 0x0F0F0F0F,  8) | x & ~0x0F0F0F0F;
x = shlr(x & 0x33333333,  4) | x & ~0x33333333;
x = shlr(x & 0x55555555,  2) | x & ~0x55555555;
x = shlr(x, 1);
```

The code uses *and with complement* to avoid loading some masks. If your machine does not have that instruction, it can be avoided by rewriting the first line of code as

```
x = shlr(x, 16) & 0x00FF00FF | x & ~0x00FF00FF;
```

which is a MUX operation, and using the identity

$$\boldsymbol{x} \,\&\, \boldsymbol{m} \;|\; \boldsymbol{y} \,\&\, \neg \boldsymbol{m} \;=\; ((\boldsymbol{x} \oplus \boldsymbol{y}) \,\&\, \boldsymbol{m}) \oplus \boldsymbol{y}$$

to obtain

```
x = ((shlr(x, 16) ^ x) & 0x00FF00FF) ^ x;
```

and similarly for the other lines that have *and with complement*.

A slightly better way for many machines, in that it has a little instruction-level parallelism, is to use the identity [Karv]

$$\boldsymbol{x} \,\&\, \neg \boldsymbol{m} \;=\; (\boldsymbol{x} \,\&\, \boldsymbol{m}) \oplus \boldsymbol{x},$$

and common the *and* expression. This gives the function shown in Figure 7–2 (17 instructions plus eight to load constants, or 25 in all).

```
unsigned rev(unsigned x) {
   unsigned t;
   t = x & 0x00FF00FF; x = shlr(t, 16) | t ^ x;
   t = x & 0x0F0F0F0F; x = shlr(t,  8) | t ^ x;
   t = x & 0x33333333; x = shlr(t,  4) | t ^ x;
   t = x & 0x55555555; x = shlr(t,  2) | t ^ x;
   x = shlr(x, 1);
   return x;
}
```

FIGURE 7–2. Reversing bits with rotate shifts.

It is perhaps worth noting that the constants 0x00FF00FF, 0x0F0F0F0F, and so on, can be generated one from another as shown below. This is not useful for 32-bit machines (it may even be harmful by reducing parallelism), because 32-bit RISC machines generally can load the constants in two instructions. But it might be useful for a 64-bit machine, for which it is illustrated.

$$\begin{aligned}
&\boldsymbol{C}_0 \leftarrow \mathbf{0x0000\,0000\,FFFF\,FFFF}\\
&\boldsymbol{C}_1 \leftarrow \boldsymbol{C}_0 \oplus (\boldsymbol{C}_0 \ll 16)\\
&\boldsymbol{C}_2 \leftarrow \boldsymbol{C}_1 \oplus (\boldsymbol{C}_1 \ll 8)\\
&\ldots
\end{aligned}$$

Another way to reverse bits is to break the word up into three groups of bits, and swap the leftmost and rightmost groups, leaving the center group in place [Baum]. For a 27-bit word, this works as illustrated below.

```
012345678 9abcdefgh ijklmnopq   The given 27-bit word
ijklmnopq 9abcdefgh 012345678   First ternary swap
opqlmnijk fghcde9ab 678345012   Second ternary swap
qponmlkji hgfedcba9 876543210   Third ternary swap
```

Straightforward code for this follows. If run on a 32-bit machine, it reverses bits 0 to 26, placing the result in bit positions 0 to 26, and clearing bits 27 to 31.

```
x = (x & 0x000001FF) << 18 | (x & 0x0003FE00) |
    (x >> 18) & 0x000001FF;
x = (x & 0x001C0E07) <<  6 | (x & 0x00E07038) |
    (x >> 6) & 0x001C0E07;
x = (x & 0x01249249) <<  2 | (x & 0x02492492) |
    (x >> 2) & 0x01249249;
```

This amounts to 21 basic RISC instructions plus 10 to load the constants, or 31 in all. In comparison, the code of Figure 7–1 is 24 basic RISC instructions plus six to load constants, plus a shift right of 5 to right-justify the result, or 31 in all. Thus the ternary method is equal or superior when there are 27 or fewer bits to be reversed.

The next function, by Donald E. Knuth [Knu8], is interesting because it reverses a 32-bit word with only four stages, and the shifting and masking steps are unexpectedly irregular. It uses one rotate shift and three ternary swaps. It works as follows:

```
01234567 89abcdef ghijklmn opqrstuv   Given
fghijklm nopqrstu v0123456 789abcde   Rotate left 15
pqrstuvm nofghijk labcde56 78901234   10-swap
tuvspqrm nojklifg hebcda96 78541230   4-swap
vutsrqpo mnlkjihg fedcba98 76543210   2-swap
```

Straightforward code is shown below.

```
x = shlr(x, 15);              // Rotate left 15.
x = (x & 0x003F801F) << 10 | (x & 0x01C003E0) |
    (x >> 10) & 0x003F801F;
x = (x & 0x0E038421) <<  4 | (x & 0x11C439CE) |
    (x >>  4) & 0x0E038421;
x = (x & 0x22488842) <<  2 | (x & 0x549556B5) |
    (x >>  2) & 0x22488842;
```

An improvement in operation count, at the expense of parallelism, results from rewriting

```
x = (x & M1) << s | (x & M2) | (x >> s) & M1;
```

where `M2` is `~(M1 | (M1 << s))`, as:

```
   t = (x ^ (x >> s)) & M1; x = (t | (t << s)) ^ x;
```

This results in the code shown below (19 full RISC instructions plus six to load constants, or 25 in all).

```
unsigned rev(unsigned x) {
   unsigned t;

   x = shlr(x, 15);             // Rotate left 15.
   t = (x ^ (x>>10)) & 0x003F801F; x = (t | (t<<10)) ^ x;
   t = (x ^ (x>> 4)) & 0x0E038421; x = (t | (t<< 4)) ^ x;
   t = (x ^ (x>> 2)) & 0x22488842; x = (t | (t<< 2)) ^ x;
   return x;
}
```

FIGURE 7–3. Reversing bits, Knuth's algorithm.

Although Knuth's algorithm does not beat the algorithm given above for reversing a 32-bit quantity with rotate shifts allowed (Figure 7–2, 17 instructions plus eight to load constants), Knuth's code uses only one rotate shift instruction. If it is coded as

```
   x = (x << 15) | (x >> 17);   // Rotate left 15.
```

then Knuth's algorithm is 21 instructions plus six to load constants, which is the best found by these measures for rotating a 32-bit word using only basic RISC instructions. This makes one wonder if there is a simple way to predict the number of shifts and logical operations required to reverse a word of a given length.

Can Knuth's algorithm be extended to reversing 64 bits on a 64-bit machine? Yes, there is a simple way and a way that is more difficult to work out. The simple way is to first swap the two halves of the 64-bit register, and then apply the 32-bit version of Knuth's algorithm to both halves, in parallel. The resulting code is shown below. It is 24 operations if the swap (rotate 32) counts as one.

```
unsigned long long rev(unsigned long long x) {
   unsigned long long t;

   x = (x << 32) | (x >> 32);    // Swap register halves.
   x = (x & 0x0001FFFF0001FFFFLL) << 15 | // Rotate left
       (x & 0xFFFE0000FFFE0000LL) >> 17;  // 15.
   t = (x ^ (x >> 10)) & 0x003F801F003F801FLL;
   x = (t | (t << 10)) ^ x;
   t = (x ^ (x >> 4)) & 0x0E0384210E038421LL;
   x = (t | (t << 4)) ^ x;
   t = (x ^ (x >> 2)) & 0x2248884222488842LL;
   x = (t | (t << 2)) ^ x;
   return x;
}
```

FIGURE 7–4. Knuth's algorithm applied to 64 bits.

The other way is to find shift amounts and masks analogous to those used in Knuth's 32-bit reversal algorithm. This is shown below. It is 25 operations if the rotate left shift of 31 positions counts as one operation.

```
unsigned long long rev(unsigned long long x) {
   unsigned long long t;

   x = (x << 31) | (x >> 33);   // I.e., shlr(x, 31).
   t = (x ^ (x >> 20)) & 0x00000FFF800007FFLL;
   x = (t |(t << 20)) ^ x;
   t = (x ^ (x >> 8)) & 0x00F8000F80700807LL;
   x = (t |(t << 8)) ^ x;
   t = (x ^ (x >> 4)) & 0x0808708080807008LL;
   x = (t |(t << 4)) ^ x;
   t = (x ^ (x >> 2)) & 0x1111111111111111LL;
   x = (t |(t << 2)) ^ x;
   return x;
}
```

Bit reversal can be aided by table lookup. The code below reverses a byte at a time, using a 256-byte table, and accumulates in reverse order the four bytes selected from the table. If the loop is strung out, this amounts to 13 basic RISC instructions plus four loads, so it could be a winner on some machines.

```
unsigned rev(unsigned x) {
   static unsigned char table[256] = {0x00, 0x80, 0x40,
   0xC0, 0x20, 0xA0, 0x60, 0xE0, ..., 0xBF, 0x7F, 0xFF};
   int i;
   unsigned r;
   r = 0;
   for (i = 3; i >= 0; i--) {
      r = (r << 8) + table[x & 0xFF];
      x = x >> 8;
   }
   return r;
}
```

- - -

Pages 109–111, replace the material from the first sentence on p. 109 up to but not including the heading “Transposing a 32x32-Bit Matrix” on p. 111, with the following material. This rewrite replaces a very poor naive method with a better “straightforward” method, and employs code designed primarily for a 64-bit machine.

Figure 7–5 illustrates the transposition of a bit matrix of size 2×3 bytes. A, B, …, F are submatrices of size 8×8 bits. A^T, B^T, … denote the transpose of submatrices A, B, ….

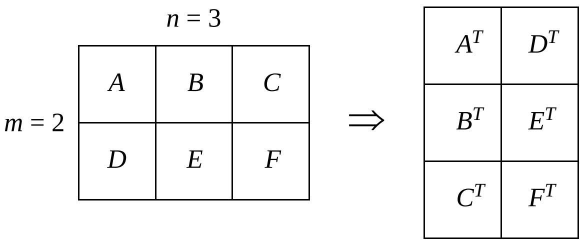


FIGURE 7–5. Transposing a 16×24-bit matrix.

For the purposes of transposing an 8×8 submatrix, it doesn’t matter whether the bit matrix is stored in row-major or column-major order; the operations are the same in either event. Assume for discussion that it’s in row-major order. Then the first byte of the matrix contains the top row of A, the next byte contains the top row of B, and so on. If L denotes the address of the first byte (top row) of a submatrix, then successive rows of the submatrix are at locations $L + n$, $L + 2n$, …, $L + 7n$.

For this problem we will depart from the usual assumption of a 32-bit machine, and assume the machine has 64-bit general registers. The algorithms are simpler and more easily understood in this way, and it is not difficult to convert them for execution on a 32-bit machine. In fact, a compiler that supports 64-bit integer operations on a 32-bit machine will do the work for you (although probably not as effectively as you can do by hand).

The overall scheme is to load a submatrix with eight *load byte* instructions, and pack the bytes left-to-right into a 64-bit register. Then the transpose of the register's contents is computed. Finally the result is stored in the target area with eight *store byte* instructions.

The transposition of an 8×8 bit matrix is illustrated below, where each character represents a single bit.

```
0123 4567          08go wEMU
89ab cdef          19hp xFNV
ghij klmn          2aiq yGOW
opqr stuv    ⇒     3bjr zHPX
wxyz ABCD          4cks AIQY
EFGH IJKL          5dlt BJRZ
MNOP QRST          6emu CKS$
UVWX YZ$.          7fnv DLT.
```

In terms of doublewords, the transformation to be done is to change the first line to the second line below.

```
01234567 89abcdef ghijklmn opqrstuv wxyzABCD EFGHIJKL MNOPQRST UVWXYZ$.
08g0wEMU 19hpxFNV 2aiqyGOW 3bjrzHPX 4cksAIQY 5dltBJRZ 6emuCKS$ 7fnvDLT.
```

Notice that the bit denoted by 1 moves seven positions to the right, the bit denoted by 2 moves 14 positions to the right, and the bit denoted by 8 moves seven positions to the left. Every bit moves 0, 7, 14, 21, 28, 35, 42, or 49 positions to the left or right. Since there are 56 bits in the doubleword that have to be moved, and only 14 different nonzero movement amounts, an average of about four bits can be moved at once, with appropriate masking and shifting. Straightforward code for this follows.

```
y =  x & 0x8040201008040201LL         |
    (x & 0x0080402010080402LL) <<  7 |
    (x & 0x0000804020100804LL) << 14 |
    (x & 0x0000008040201008LL) << 21 |
    (x & 0x0000000080402010LL) << 28 |
    (x & 0x0000000000804020LL) << 35 |
    (x & 0x0000000000008040LL) << 42 |
    (x & 0x0000000000000080LL) << 49 |
    (x >>  7) & 0x0080402010080402LL |
    (x >> 14) & 0x0000804020100804LL |
    (x >> 21) & 0x0000008040201008LL |
    (x >> 28) & 0x0000000080402010LL |
    (x >> 35) & 0x0000000000804020LL |
    (x >> 42) & 0x0000000000008040LL |
    (x >> 49) & 0x0000000000000080LL;
```

This executes in 43 instructions on the basic RISC, exclusive of mask generation (which is not important in the application of transposing a large bit matrix, because the masks are loop constants). Rotate shifts do not help. Some of the terms are of the form `(x & mask) << s`, and some are of the form `(x >> s) & mask`. This reduces the number of masks required; the last seven are repeats of earlier masks. Notice that each mask after the first can be generated from the first with one *shift right* instruction. Because of this, it is a simple matter to write a more compact version of the code that uses a for-loop that is executed seven times.

Another variation is to employ Steele's method of using *exclusive or* to swap bit fields (described on page 40). That technique does not help much in this application. It results in a function that executes in 42 instructions, exclusive of mask generation. The code starts out

```
t = (x ^ (x >> 7)) & 0x0080402010080402LL;
x = x ^ t ^ (t << 7);
```

and there are seven such pairs of lines.

Although there does not seem to be a *really great* algorithm for this problem, the method to be described beats the straightforward method and its variations described above by approximately a factor of 2 on the basic RISC, for the calculation part (not counting loading and storing the submatrices or generating masks). The method gets its power from its high level of bit-parallelism. It would not be a good method if the matrix elements are words. For that, you can't do better than loading each word and storing it where it goes.

First treat the 8×8-bit matrix as 16 2×2-bit matrices, and transpose each of the 16 2×2-bit matrices. Then treat the matrix as four 2×2 submatrices whose elements are 2×2-bit matrices and transpose each of the four 2×2 submatrices. Finally, treat the matrix as a 2×2 matrix whose elements are 4×4-bit matrices, and transpose the 2×2 matrix. These transformations are illustrated below [Floyd].

```
0123 4567     082a 4c6e     08go 4cks     08go wEMU
89ab cdef     193b 5d7f     19hp 5dlt     19hp xFNV
ghij klmn     goiq ksmu     2aiq 6emu     2aiq yGOW
opqr stuv     hpjr ltnv     3bjr 7fnv     3bjr zHPX
           ⇒             ⇒             ⇒
wxyz ABCD     wEyG AICK     wEMU AIQY     4cks AIQY
EFGH IJKL     xFzH BJDL     xFNV BJRZ     5dlt BJRZ
MNOP QRST     MUOW QYS$     yGOW CKS$     6emu CKS$
UVWX YZ$.     NVPX RZT.     zHPX DLT.     7fnv DLT.
```

A complete procedure in shown in Figure 7–6. Parameter `A` is the address of the first byte of an 8×8 submatrix of the source matrix, and parameter `B` is the address of the first byte of an 8×8 submatrix in the target matrix.

The calculation part of this function executes in 21 instructions. Each of the three major steps is swapping bits, so a version can be written that uses the Steele *exclusive or* bit field swapping device. Using it, the first assignment to `x` in Figure 7–6 becomes:

```
t = (x ^ (x >> 7)) & 0x00AA00AA00AA00AALL;
x = x ^ t ^ (t << 7);
```

```
void transpose8(unsigned char A[8], int m, int n,
                unsigned char B[8]) {
   unsigned long long x;
   int i;

   for (i = 0; i <= 7; i++)      // Load 8 bytes from the
      x = x << 8 | A[m*i];       // input array and pack
                                 // them into x.

   x =   x & 0xAA55AA55AA55AA55LL         |
        (x & 0x00AA00AA00AA00AALL) <<  7 |
        (x >> 7) & 0x00AA00AA00AA00AALL;
   x =   x & 0xCCCC3333CCCC3333LL         |
        (x & 0x0000CCCC0000CCCCLL) << 14 |
        (x >> 14) & 0x0000CCCC0000CCCCLL;
   x =   x & 0xF0F0F0F00F0F0F0FLL         |
        (x & 0x00000000F0F0F0F0LL) << 28 |
        (x >> 28) & 0x00000000F0F0F0F0LL;

   for (i = 7; i >= 0; i--) {    // Store result into
      B[n*i] = x; x = x >> 8;}   // output array B.
}
```

FIGURE 7–6. Transposing an 8×8-bit matrix.

The calculation part of the revised function executes in only 18 instructions, but it has no instruction level parallelism.

The algorithm of Figure 7–6 runs from fine to coarse granularity, based on the lengths of the groups of bits that are swapped. The method can also be run from coarse to fine granularity. To do this, first treat the 8×8-bit matrix as a 2×2 matrix whose elements are 4×4-bit matrices, and transpose the 2×2 matrix. Then treat each the four 4×4 submatrices as a 2×2 matrix whose elements are 2×2-bit matrices, and transpose each of the four 2×2 submatrices, etc. The code for this is the same as that of Figure 7–6 except with the three assignments that do the bit rearranging run in reverse order.

As was mentioned, these functions can be modified for execution on a 32-bit machine by using two registers for each 64-bit quantity. If this is done and any calculations that would result in zero are used to make obvious simplifications, the results are that a 32-bit version of the straightforward method described on page 39 runs in 74 instructions (compared to 43 on a 64-bit machine), and a 32-bit version of the function of Figure 7–6 runs in 36 instructions (compared to 21 on a 64-bit machine). Using Steele's bit-swapping technique gives a reduction in instructions executed at the expense of instruction level parallelism, as in the case of a 64-bit machine.

As part of this change, on page 113, second line from the bottom, change "of Figure 7–2, then" to:

of a 32-bit version of Figure 7–4, then

- - -

Page 119, add the following footnote to "... 127 basic RISC instructions (constant)":

1. Actually, the first *shift left* can be omitted, reducing the instruction count to 126. The quantity `mv` comes out the same with or without it [Dalton].

- - -

Page 122, append to the **Compress Left** subsection the following paragraph and two new sections:

The BESM-6 computer (1967) had an instruction for the compress left function ("Pack Bits in A Masked by X") and its inverse ("Unpack …"), which operated on the machine's 48-bit registers. These instructions are not easy to implement. It is surmised by cryptography experts that their only use was for breaking US codes [Knu8]. The BESM-6 also had the *population count* instruction which, as has been noted, seems to be important to the National Security Agency.

7–5 *Expand*, or *Generalized Insert*

The inverse of the *compress right* function moves bits from the low-order end of a register to positions given by a mask, while keeping the bits in order. For example, expand(0000abcd, 10011010) = a00bc0d0. Thus

$$\text{compress}(\text{expand}(\boldsymbol{x}, \boldsymbol{m}), \boldsymbol{m}) = \boldsymbol{x}.$$

This function has also been called *unpack*, *scatter*, and *deposit*.

It may be obtained by running the code of Figure 7–6 in reverse [Allen]. To avoid overwriting bits in `x`, it is necessary to move (to the left) the bits that move a large distance first, and to move those that move only one position last. This means that the first five "move" quantities (`mv` in the code) must be computed, saved, and used in the reverse of the order in which they were computed. For many applications this is not a problem, because these applications apply the same mask `m` to large amounts of data, and so would compute the move quantities in advance and reuse them anyway.

The code is shown in Figure 7–7. It executes approximately 168 basic RISC instructions (constant), including five stores and five loads. A 64-bit version for a 64-bit machine would execute approximately 200 instructions.

For a machine that does not have the *and not* instruction, the MUX operation in the second loop can be coded in one less instruction with

```
   x = ((x ^ y) & mv) ^ x;
```

```
unsigned expand(unsigned x, unsigned m) {
   unsigned m0, mk, mp, mv, t;
   unsigned array[5];
   int i;

   m0 = m;              // Save original mask.
   mk = ~m << 1;        // We will count 0's to right.

   for (i = 0; i < 5; i++) {
      mp = mk ^ (mk << 1);             // Parallel suffix.
      mp = mp ^ (mp << 2);
      mp = mp ^ (mp << 4);
      mp = mp ^ (mp << 8);
      mp = mp ^ (mp << 16);
      mv = mp & m;                     // Bits to move.
      array[i] = mv;
      m = (m ^ mv) | (mv >> (1 << i));  // Compress m.
      mk = mk & ~mp;
   }

   for (i = 4; i >= 0; i--) {
      mv = array[i];
      t = x << (1 << i);
      x = (x & ~mv) | (t & mv);
   }
   return x & m0;       // Clear out extraneous bits.
}
```

FIGURE 7–7. Parallel suffix method for the *expand* operation.

7–6 Hardware Algorithms for Compress and Expand

This section gives hardware-oriented algorithms for the *compress right* function and its inverse [Zadeck]. Like the algorithms of the preceding sections, their execution times are proportional to the log of the computer's word size. They are suitable for implementation in hardware, but do not yield fast code if implemented in basic RISC instructions. We simply describe how they work without giving C or machine code.

Compress

To illustrate the operation of the algorithm, we represent each bit of $\boldsymbol{x}$ with a letter, and consider a specific example mask $\boldsymbol{m}$, shown below.

```
Input x =     abcd efgh ijkl mnop qrst uvwx yzAB CDEF
Mask m =      0111 1110 0110 1100 1010 1111 0011 0010
```

The algorithm works in five “phases,” with each phase operating in parallel on “pockets” of size 2^n bits, for n ranging from 1 to 5. At the end of each phase, each pocket of ***x*** contains the original pocket of ***x*** with the bits selected by that pocket of ***m*** compressed to the right. Each pocket of ***m*** will contain an integer that is the number of 0-bits in that pocket of the original ***m***. This is equal to the number of bits of ***x*** that are *not* compressed to the right. They are the known leading 0-bits in the pocket of ***x***.

In each phase, the algorithm performs the following steps, in parallel, on each pocket of ***x*** and ***m***, where w is the pocket size in bits.

6. Set ***L*** = the left half of the pocket of ***x***, extended with $w/2$ 0-bits on the right.
7. Shift ***L*** (all w bits) right by the amount given in the right half of the corresponding pocket of ***m***, inserting 0’s on the left. No 1’s will be shifted out on the right, because the maximum shift amount is $w/2$.
8. Set ***R*** = $w/2$ 0-bits followed by the right half of the pocket of ***x***.
9. Replace the entire w-bit pocket of ***x*** with the *or* of ***R*** and the shifted ***L***.
10. Add the left and right halves of the pocket of ***m***, and replace the entire pocket with the sum.

To apply these steps to the first phase ($w = 2$) would require first *and*’ing ***x*** with ***m***, to clear out irrelevant bits of ***x***, and complementing ***m***, so that each bit of ***m*** is the number of 0-bits in each 1-bit half pocket. It is simpler to make an exception of the first phase, and combine these steps with the first compression operation by applying the logic shown in the truth table below to each 2-bit pocket of ***x*** and ***m***.

Input		**Output**	
x	***m***	***x***	***m***
ab	00	00	10
ab	01	0b	01
ab	10	0a	01
ab	11	ab	00

The third line, for example, has ***m*** = 10 (binary). This means that the left bit of ***x*** is selected to be part of the result, but the right bit is not. Thus the left bit (a) is compressed to the right. The other bit of ***x*** is cleared, which ensures that in the final result, all the high-order (unselected) bits will be 0.

Applying this logic to the original ***x*** and ***m*** gives:

```
Bit pairs, x = 0bcd ef0g 0j0k mn00 0q0s uvwx 00AB 000E
           m = 0100 0001 0101 0010 0101 0000 1000 1001
```

In the second phase, consider for example the second nibble above (`ef0g`). The quantities ***L*** = `ef00` and ***R*** = `000g` are formed. ***L*** is shifted right by one position (given by the right half of the nibble of ***m***), giving `0ef0`. This is *or*'ed with ***R***, giving `0efg` as the new value of the nibble. The left and right halves of ***m*** are added, giving `0001` (no change).

```
Nibbles,   x = 0bcd 0efg 00jk 00mn 00qs uvwx 00AB 000E
           m = 0001 0001 0010 0010 0010 0000 0010 0011
```

Similarly, for the third, fourth, and fifth phases, each byte, halfword, and word of ***x*** is compressed, and ***m*** is updated, as follows.

```
Bytes,     x = 00bc defg 0000 jkmn 00qs uvwx 0000 0ABE
           m = 0000 0010 0000 0100 0000 0010 0000 0101

Halfwords, x = 0000 00bc defg jkmn 0000 000q suvw xABE
           m = 0000 0000 0000 0110 0000 0000 0000 0111

Words,     x = 0000 0000 0000 0bcd efgj kmnq suvw xABE
           m = 0000 0000 0000 0000 0000 0000 0000 1101
```

Upon completion, ***m*** is an integer that gives the number of known leading 0's in ***x***. Subtracting this from the word size gives the number of compressed bits in ***x***, which equals the number of 1-bits in the original mask ***m***.

The reason this is not a very good algorithm for implementation with basic RISC instructions is that it is hard to shift the half-pockets right by differing amounts. But it might possibly be useful on a SIMD machine that has instructions that operate on the pockets of a word in parallel and independently.

Expand

The hardware compression algorithm can be turned into an expansion algorithm by, essentially, running it first forward and then in reverse. As in the algorithms based on the parallel suffix method, the five masks of the hardware compression algorithm are computed, saved, and used in the reverse of the order in which they were computed. Actually, the last mask is not used (nor is it used in the compression algorithm), but an additional one is required (***m0***), that is simply the complement of the original mask. In the forward pass, only the steps for computing the masks need be done; those involving the data ***x*** can be omitted.

To illustrate, suppose we have

```
Input x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
Mask  m = 0111 1110 0110 1100 1010 1111 0011 0010
```

Then the result of the expansion should be

```
          0nop qrs0 0tu0 vw00 x0y0 zABC 00DE 00F0.
```

The masks are shown below.

```
m0 = 1000 0001 1001 0011 0101 0000 1100 1101
m1 = 0100 0001 0101 0010 0101 0000 1000 1001
m2 = 0001 0001 0010 0010 0010 0000 0010 0011
m3 = 0000 0010 0000 0100 0000 0010 0000 0101
m4 = 0000 0000 0000 0110 0000 0000 0000 0111
```

The integer values of each half of ***m4*** give the number of 0-bits in the corresponding half of the original mask ***m***. In particular, the right half of ***m*** has seven 0-bits. This means that the seven high-order bits of the right half of ***x*** do not belong there—they should be in the left half of ***x***. Thus bits 9 through 15 of ***x*** should be shifted left just enough to put them in the left half of ***x***, and higher order bits of ***x*** should be shifted left to accomodate them. This can be accomplished by shifting left the entire 32-bit word ***x*** by seven positions, and replacing the left half of ***x*** with the left half of the shifted quantity. This gives

```
x = hijk lmno pqrs tuvw qrst uvwx yzAB CDEF.
```

In general, the algorithm works with pocket sizes from 32 down to 2, in five phases, using masks ***m4*** down to ***m0***. Each pocket (in parallel) is shifted left, discarding bits that are shifted out on the left, and supplying 0’s to vacated positions on the right, so that the shifted quantity is the same length as the pocket from which it came. Then the left half of the pocket is replaced by the left half of the shifted quantity. This will leave “garbage” bits in both halves of the pocket. They will be zeroed-out after the last phase, by *and*ing with the original mask.

Continuing, we treat ***m3*** as two 16-bit pockets. The left pocket has the integer 4 in its right half, so the left pocket of ***x*** is shifted left four positions (giving `lmno pqrs tuvw 0000`), and the left half of this replaces the left half of the left pocket in ***x***, making the left pocket of ***x*** = `lmno pqrs`. Performing the same operation on the right 16-bit pocket of ***x*** gives

```
x = lmno pqrs pqrs tuvw vwxy zABC yzAB CDEF.
```

The next phase uses ***m2***, which consists of four eight-bit pockets. Applying it to ***x*** gives

```
x = mnop pqrs rstu tuvw vwxy zABC BCDE CDEF.
```

The next phase uses ***m1***, which consists of eight four-bit pockets. Applying it to ***x*** gives

```
x = mnop qrrs sttu vwvw wxxy zABC BCDE DEEF.
```

The last phase uses ***m0***, which consists of 16 two-bit pockets. Applying it to ***x*** gives

```
x = mnop qrss stuu vwww xxyy zABC CCDE EEFF.
```

The final step is to *and* this with the original mask, to clear irrelevant bits. This gives

```
x = 0nop qrs0 0tu0 vw00 x0y0 zABC 00DE 00F0.
```

The half-pockets of each computed mask contain a count of the number of 0-bits in the corresponding half-pocket of the original mask ***m***. Therefore, as an alternative to computing the masks and saving them, the machine could employ circuits for doing a *population count* of the 0's in the half-pockets "on the fly."

- - -

Page 126, bottom, add the paragraph:

There is a neat hack to add 1 to the goats—that is, to compute

$$\mathrm{SAG}^{-1}(\mathrm{SAG}(\boldsymbol{x}, \boldsymbol{m}) + 1, \boldsymbol{m})$$

without using the SAG function or its inverse [Knu8]. Here we assume SAG($\boldsymbol{x}$, $\boldsymbol{m}$) puts the goats on the right, and the addition does not overflow into the "sheep" field. We leave to the reader the pleasure of discovering this trick.

- - -

Page 128, add the following section after Section 7–6:

7–7 An LRU Algorithm

Ever wonder how your computer keeps track of which cache line is the least recently used? Here we describe one such algorithm, known as the *reference matrix* method. It is primarily a hardware algorithm, but it might have application in software.

We won't go into a long discussion of the intriguing world of caches, but only say that we have in mind the high speed caches that buffer data between a computer's main memory and the processor. These caches may get a request for a word every computer cycle, and they should usually respond with the data within a cycle or two, so there is not much time for a complicated algorithm.

A cache contains a copy of a subset of the data in main memory, and the problem we are addressing is: when a cache miss occurs (that is, when a word at a certain address is requested, and the data at that address is not in the cache), how does the computer decide which block (or *line*, in cache jargon) to replace with the requested data? Ideally, it should replace the data in the line that will not be referenced for the longest time in the future. But we cannot know the future, so we have to guess. The best guess over a wide variety of application programs seems to be the *least recently used* (LRU) policy. This policy replaces the line that has not been referenced for the longest time.

Caches come in three varieties, called *direct-mapped, fully associative*, and *set-associative*. In a direct-mapped cache, certain bits of the address of the load or store instruction directly address a particular cache line. When a miss occurs, there is no question of what line to replace, it must be the addressed line. There is no need for an LRU or any other guessing policy.

In a fully associative cache, a block from main memory can be placed in *any* cache line. When a load or store is executed, the address is looked up to see if it is in the cache. If not, it is necessary to replace the contents of some line. The machine has complete flexibility in the choice of line to replace. Several strategies have been used (FIFO, random, and LRU are the most common) and, as mentioned above, LRU seems to be the one that most often results in the lowest miss rate. But LRU is the most expensive to implement when there are many lines to consider for replacement.

Often the set-associative organization is chosen. It is a compromise between direct-mapped and fully associative. The designer decides on the degree of associativity, which is usually 2, 4, 8, or 16. The cache is divided into a number of "sets," each of which contains 2, 4, 8, or 16 lines (typically). The set is directly addressed, using certain bits of the load or store address, but the line within the set must be looked up. The lookup in the set is done much the same as in the case of a fully associative cache. Now, when it is necessary to replace a line, the LRU algorithm need only determine which of the lines within one set is the least recently used, and replace that.

With this brief background, we can describe the reference matrix method. To illustrate, assume the cache is four-way set-associative. This means that there are four lines for which we wish to keep track of the least recently used (referenced). The cache may be fully associative and consist of only four lines, or it may be set-associative with four lines per set.

The reference matrix method employs a square bit matrix of dimension equal to the degree of associativity (in principle; we will modify this statement later). Each associative set has one such matrix. The essence of the method is that when line *i* is referenced, row *i* of the matrix is set to 1's, and then column *i* is set to 0's.The figure below illustrates the changes in the matrix from an initial state to its configuration after a reference to lines 3, 1, 0, 2, 0, 3, and 2, in that order.

	Init	**3**	**1**	**0**	**2**	**0**	**3**	**2**
Line	**0123**	**0123**	**0123**	**0123**	**0123**	**0123**	**0123**	**0123**
0	0111	0110	0010	0111	0101	0111	0110	0100
1	0011	0010	1011	0011	0001	0001	0000	0000
2	0001	0000	0000	0000	1101	0101	0100	1101
3	0000	1110	1010	0010	0000	0000	1110	1100

FIGURE 7–8. Illustration of the reference matrix method.

Each matrix has a row containing three 1’s, two 1’s, one 1, and no 1’s. The number of the row with no 1’s is the least recently used line. The number of the row with one 1 is the next least recently used line, and so on. When a cache miss occurs, the machine finds the row with all 0’s and replaces the corresponding line. It then records it as the *most* recently used line by setting its row to all 1’s and its column to all 0’s.

Why does this work? Denoting the matrix by *M*, the reason it works is that M_{ij} indicates whether or not line *i* is more recently used than line *j*. If $M_{ij} = 1$, line *i* is more recently used than line *j*, and if $M_{ij} = 0$, line *i* is not more recently used than line *j*.

Consider an arbitrary 4×4 matrix for which line 2 is referenced. Then the matrix changes as shown below.

$$\begin{array}{cc|cccc|c|cccc|}
 & & \multicolumn{4}{c}{\textbf{Init}} & & \multicolumn{4}{c}{\textbf{2}} \\
 & & \textbf{0} & \textbf{1} & \textbf{2} & \textbf{3} & & \textbf{0} & \textbf{1} & \textbf{2} & \textbf{3} \\
\cline{3-6}\cline{8-11}
\textbf{Line} & \textbf{0} & a & b & c & d & & a & b & 0 & d \\
 & \textbf{1} & e & f & g & h & & e & f & 0 & h \\
 & \textbf{2} & i & j & k & l & & 1 & 1 & 0 & 1 \\
 & \textbf{3} & m & n & o & p & & m & n & 0 & p \\
\cline{3-6}\cline{8-11}
\end{array}$$

FIGURE 7–9. One step of the reference matrix method.

Setting row *i* to 1’s (except for the element on the main diagonal) is recording that line *i* is more recently used than line *j*, for all $j \neq i$. Setting column *i* to 0’s is recording that line *j* is not more recently used than line *i*, for all *j*. Relations among cache lines other than *i* are not changed. When all the lines have been referenced, all the “more recently used” relations will be established.

Thus the reference matrix is antisymmetric and the main diagonal is always all 0’s. Therefore, only part of the matrix, either the elements above the main diagonal or those below the main diagonal, need be stored in the cache. That is what is done in practice. For an *n*-way associative set, $n(n-1)/2$ memory bits are required. For $n = 4$, this is six; for $n = 8$, it is 28. Twenty-eight is getting to be a bit large, so the reference matrix method, and in fact the true LRU policy, is not often used for degrees of associativity greater than 8. Instead, there are approximate LRU methods and methods that are not LRU at all.

In software, the LRU policy would probably be implemented with a list of the line numbers (either a simple vector or a linked list). When line *i* is referenced, the list is searched for *i*, and then *i* is moved to the top of the list. The least recently used line number then migrates to the bottom of the list.

That method is relatively slow on references (because of rearranging the list), but fast in deciding which line to replace. Another method, with the opposite speed characteristics, is to have a vector of length equal to the degree of associativity, with position *i* holding both the address that line *i* holds and its “age” (actually “newness”) encoded as an integer. When line *i* is referenced, a single variable that holds the current “age” is incremented, and the resulting value is stored in the vector at position *i*. To find the least recently used line, the vector is searched for the line with the smallest value of "age." This method fails if the "age" integer overflows.

There might be one "age" integer per associative set, or only one for the whole cache, or in hardware, a cycle counter could be used.

The reference matrix method might be useful in software when the degree of associativity is small. For example, suppose an application uses eight-way set-associativity and is to run on a 64-bit machine. Then the reference matrix can be stored in a single 64-bit register. Let the low-order eight bits of the register hold row 0 of the matrix, the next eight bits hold row 1, and so forth. Then when line *i* is referenced, byte *i* of the register should be set to 1's, and bits $i,\ i+8,\ \ldots,\ i+56$ should be cleared. Denoting the register by $\boldsymbol{m}$, this is accomplished as shown below.

$$\boldsymbol{m} \leftarrow \boldsymbol{m} \mid (\mathbf{0xFF} \ll (8 * \boldsymbol{i}))$$

$$\boldsymbol{m} \leftarrow \boldsymbol{m} \,\&\, \neg(\mathbf{0x01010101\,01010101} \ll \boldsymbol{i})$$

This amounts to five or six instructions, plus a few to load constants. To find the least recently used line, search for an all-zero byte (see Section 6–1). The advantage of this method over the other software methods briefly outlined above is that all the work is done in a register.

- - -

Page 153 (end of Chapter 9), insert the following section:

9–5 Doubleword Division from Long Division

This section considers how to do $64 \div 64 \Rightarrow 64$ division from $64 \div 32 \Rightarrow 32$ division, for both the unsigned and signed cases. The algorithms given below are most suited to a machine that has an instruction for long division $(64 \div 32)$ at least for the unsigned case. It is also helpful if the machine has the *number of leading zeros* instruction. The machine may have either 32-bit or 64-bit registers, but we will assume that if it has 32-bit registers, then the compiler implements basic operations such as adds and shifts on 64-bit operands (the "long long" data type in C).

These functions are known as "__udivdi3" and "__divdi3" in the GNU C world, and similar names are used here.

Unsigned Doubleword Division

A procedure for this operation is shown in Figure 9–5.

This code distinguishes three cases: (1) the case in which a single execution of the machine's unsigned long division instruction (DIVU) may be used, (2) the case in which (1) does not apply but the divisor is a 32-bit quantity, and (3) the cases in which the divisor cannot be represented in 32 bits. It is not too hard to see

```
unsigned long long udivdi3(unsigned long long u,
                           unsigned long long v) {

   unsigned long long u0, u1, v1, q0, q1, k, n;

   if (v >> 32 == 0) {           // If v < 2**32:
      if (u >> 32 < v)           // If u/v cannot overflow,
         return DIVU(u, v)       // just do one division.
            & 0xFFFFFFFF;
      else {                     // If u/v would overflow:
         u1 = u >> 32;           // Break u up into two
         u0 = u & 0xFFFFFFFF;    // halves.
         q1 = DIVU(u1, v)        // First quotient digit.
            & 0xFFFFFFFF;
         k = u1 - q1*v;          // First remainder, < v.
         q0 = DIVU((k << 32) + u0, v) // 2nd quot. digit.
            & 0xFFFFFFFF;
         return (q1 << 32) + q0;
      }
   }
                                 // Here v >= 2**32.
   n = nlz64(v);                 // 0 <= n <= 31.
   v1 = (v << n) >> 32;          // Normalize the divisor
                                 // so its MSB is 1.
   u1 = u >> 1;                  // To ensure no overflow.
   q1 = DIVU(u1, v1)             // Get quotient from
       & 0xFFFFFFFF;             // divide unsigned insn.
   q0 = (q1 << n) >> 31;         // Undo normalization and
                                 // division of u by 2.
   if (q0 != 0)                  // Make q0 correct or
      q0 = q0 - 1;               // too small by 1.
   if ((u - q0*v) >= v)
      q0 = q0 + 1;               // Now q0 is correct.
   return q0;
}
```

FIGURE 9–5. Unsigned doubleword division from long division.

that the above code is correct for cases (1) and (2). For case (2), think of the grade school method of doing long division.

Case (3), though, deserves proof, because it is very close to not working in some cases. Notice that in this case only a single execution of DIVU is needed, but the *number of leading zeros* and *multiply* operations are needed.

For the proof, we need these basics (for integer variables):

$$\lfloor \lfloor a/b \rfloor / d \rfloor = \lfloor a/(bd) \rfloor \qquad (4)$$

$$b\lfloor a/b \rfloor = a - \text{rem}(a, b) \qquad (5)$$

From the first line in the section of the procedure of interest (we assume that $v \neq 0$),

$$0 \le n \le 31.$$

In computing v_1, the left shift clearly cannot overflow. Therefore

$$\begin{aligned} v_1 &= \lfloor v/2^{32-n} \rfloor, \quad \text{and} \\ u_1 &= \lfloor u/2 \rfloor. \end{aligned}$$

In computing q_1, u_1 and v_1 are in range for the DIVU instruction and it cannot overflow. Hence

$$q_1 = \lfloor u_1/v_1 \rfloor.$$

In the first computation of q_0, the left shift cannot overflow because $q_1 < 2^{32}$ (because the maximum value of u_1 is $2^{63} - 1$ and the minimum value of v_1 is 2^{31}). Therefore

$$q_0 = \lfloor q_1/2^{31-n} \rfloor.$$

Now, for the main part of the proof, we want to show that

$$\lfloor u/v \rfloor \le q_0 \le \lfloor u/v \rfloor + 1,$$

which is to say, the first computation of q_0 is the desired result or is that plus 1.

Using equation (4) twice gives

$$\begin{aligned} q_0 &= \left\lfloor \frac{u}{2^{32-n} v_1} \right\rfloor \\ &= \left\lfloor \frac{u}{2^{32-n} \left\lfloor \frac{v}{2^{32-n}} \right\rfloor} \right\rfloor. \end{aligned}$$

Using equation (5) gives

$$q_0 = \left\lfloor \frac{u}{v - \operatorname{rem}(v,\, 2^{32-n})} \right\rfloor.$$

Using algebra to get this in the form $u/v +$ something:

$$q_0 = \left\lfloor \frac{u}{v} + \frac{u \operatorname{rem}(v,\, 2^{32-n})}{v(v - \operatorname{rem}(v,\, 2^{32-n}))} \right\rfloor.$$

This is of the form

$$\left\lfloor \frac{u}{v} + \delta \right\rfloor,$$

and we will now show that $\delta < 1$.

δ is largest when $\text{rem}(v, 2^{32-n})$ is as large as possible and, given that, when v is as small as possible. The maximum value of $\text{rem}(v, 2^{32-n})$ is $2^{32-n} - 1$. Because of the way n is defined in terms of v, $v \ge 2^{63-n}$. Thus the smallest value of v having that remainder is

$$2^{63-n} + 2^{32-n} - 1.$$

Thus

$$\delta \le \frac{u(2^{32-n} - 1)}{(2^{63-n} + 2^{32-n} - 1)2^{63-n}}$$
$$< \frac{u(2^{32-n} - 1)}{(2^{63-n})^2}.$$

By inspection, for n in its range of 0 to 31,

$$\delta < \frac{u}{2^{64}}.$$

Since u is at most $2^{64} - 1$, $\delta < 1$. Because $q_0 = \lfloor u/v + \delta \rfloor$ and $\delta < 1$ (and obviously $\delta \ge 0$),

$$\left\lfloor \frac{u}{v} \right\rfloor \le q_0 \le \left\lfloor \frac{u}{v} \right\rfloor + 1.$$

To correct this result by subtracting 1 when necessary, we would like to code

```
if (u < q0*v) q0 = q0 - 1;
```

(i.e., if the remainder $u - q_0 v$ is negative, subtract 1 from q_0). However, this doesn't quite work because $q_0 v$ can overflow (e.g., for $u = 2^{64} - 1$ and $v = 2^{32} + 3$). Instead, we subtract 1 from q_0, so that it is either correct or too *small* by 1. Then, $q_0 v$ will not overflow. We must avoid subtracting 1 if $q_0 = 0$ (if $q_0 = 0$, it is already the correct quotient).

Then the final correction is:

```
if ((u - q0*v) >= v) q0 = q0 - 1;
```

To see that this is a valid computation, we already noted that $q_0 v$ does not overflow. It is easy to show that

$$0 \le u - q_0 v < 2v.$$

If v is very large ($\ge 2^{63}$), can the subtraction overflow by trying to produce a result greater than v? No, because $u < 2^{64}$ and $q_0 v \ge 0$.

Incidentally, there are alternatives to the lines

```
if (q0 != 0)          // Make q0 correct or
   q0 = q0 - 1        // too small by 1.
```

that may be preferable on some machines. One is to replace them with

```
if (q0 == 0) return 0;
```

Another is to place at the beginning of this section of the procedure, or at the beginning of the whole procedure, the line

```
if (u < v) return 0; // Avoid a problem later.
```

These alternatives are preferable if branches are not costly. The code shown in Figure 9–5 works well if the machine's comparison instructions produce a 0/1 integer result in a general register. Then, the compiler can change it to, in effect,

```
q0 = q0 - (q0 != 0);
```

(or you can code it that way if your compiler doesn't do this optimization). This is just a *compare* and *subtract* on such machines.

Signed Doubleword Division

In the signed case, there seems to be no better way to do doubleword division than to divide the absolute values of the operands, using function `udivdi3`, and then negate the sign of the quotient if the operands have different signs. If the machine has a signed long division instruction, which we designate here as DIVS, then it may be advantageous to single out the cases in which DIVS can be used, rather than invoking `udivdi3`. This presumes that these cases are common. Such a function is shown in Figure 9–6.

The "`#define`" in this code uses the GCC facility of enclosing a compound statement in parentheses to construct an expression, a facility that most C compilers do not have. Some other compilers may have `llabs(x)` as a built-in function.

The test that `v` is in range is not precise; it misses the case in which $v = -2^{31}$. If it is important to use the DIVS instruction in that case, the test

```
if ((v << 32) >> 32 == v) {  // If v is in range and
```

may be used in place of the third executable line in Figure 9–6 (at a cost of one instruction). Similarly, the test that $|u| / |v|$ cannot overflow is simplified and a few "corner cases" will be missed; the code amounts to using $\delta = 0$ in the signed

```
#define llabs(x) \
({unsigned long long t = (x) >> 63; ((x) ^ t) - t;})

long long divdi3(long long u, long long v) {

   unsigned long long au, av;
   long long q, t;

   au = llabs(u);
   av = llabs(v);
   if (av >> 31 == 0) {            // If |v| < 2**31 and
      if (au < av << 31) {         // |u|/|v| cannot
         q = DIVS(u, v);           // overflow, use DIVS.
         return (q << 32) >> 32;
      }
   }
   q = au/av;                      // Invoke udivdi3.
   t = (u ^ v) >> 63;              // If u, v have different
   return (q ^ t) - t;             // signs, negate q.
}
```

FIGURE 9–6. Signed doubleword division from unsigned doubleword division.

division overflow test scheme shown in Section 2–12 [not in *Hacker's Delight*, see this document, new material for page 33].

- - -

Page 184, just before Section 10–11, add the paragraph:

To use the results of this program, the compiler should generate the `li` and `mulhu` instructions and, if the “add” indicator `a` = 0, generate the `shri` of `s` (if `s` > 0), as illustrated by the unsigned divide by 3 example on page 178. If `a` = 1 and the machine has the `shrxi` instruction, the compiler should generate the `add` and `shrxi` of `s` as illustrated by the unsigned divide by 7 example on page 179. If `a` = 1 and the machine does not have the `shrxi` instruction, use the example on page 180: generate the `sub`, the `shri` of 1, the `add`, and finally the `shri` of `s` – 1 (if `s` – 1 > 0; `s` will not be 0 at this point except in the trivial case of division by 1, which we assume the compiler deletes).

- - -

Page 190, just before Section 10–15, insert the following section:

10–15 Simple Code in Python

Computing a magic number is greatly simplified if one is not limited to doing the calculations in the same word size as that of the environment in which the magic number will be used. For the unsigned case, for example, in Python it is straightforward to compute n_c and then evaluate equations (27) and (26), as described in Section 10-9. Figure 10–4 shows such a function.

```
def magicgu(nmax, d):
   nc = (nmax//d)*d - 1
   nbits = int(log(nmax, 2)) + 1
   for p in range(0, 2*nbits + 1):
      if 2**p > nc*(d - 1 - (2**p - 1)%d):
         m = (2**p + d - 1 - (2**p - 1)%d)//d
         return (m, p)
   print "Can't find p, something is wrong."
   sys.exit(1)
```

FIGURE 10–4. Python code for computing the magic number for unsigned division.

The function is given the maximum value of the dividend nmax, and the divisor d. It returns a pair of integers: the magic number m and a shift amount p. To divide a dividend x by d, one multiplies x by m and then shifts the (full length) product right p bits.

This program is more general than the others in this chapter in two ways: (1) one specifies the maximum value of the dividend (nmax), rather than the number of bits required for the dividend, and (2) the program can be used for arbitrarily large dividends and divisors ("bignums"). The advantage of specifying the maximum value of the dividend is that one sometimes gets a smaller magic number than would be obtained if one used the next power of two less 1 for the maximum value. For example, suppose the maximum value of the dividend is 90, and the divisor is 7. Then function magicgu returns (37, 8), meaning that the magic number is 37 (a six-bit number) and the shift amount is 8. But if we asked for a magic number that can handle divisors up to 127, then the result is (147, 10), and 147 is an eight-bit number.

- - -

Page 226, delete the last sentence of the third paragraph ("There does not seem..."), and add the following material after that paragraph:

For base –2, there is no device quite that simple, but a method that is nearly as simple is to complement the minuend (meaning to invert each bit), add the complemented minuend to the subtrahend, and then complement the sum [Lang].

Below is an example showing the subtraction of 13 from 6 using this scheme, on an eight-bit machine.

```
00011010   6
00011101   13
11100101   6 complemented
--------
11110110   (6 complemented) + 13
00001001   Complement of the sum (-7)
```

This method is using

$$\boldsymbol{A} - \boldsymbol{B} = \boldsymbol{I} - ((\boldsymbol{I} - \boldsymbol{A}) + \boldsymbol{B})$$

in base –2 arithmetic, with $\boldsymbol{I}$ a word of all 1’s.

- - -

Page 240, at end of chapter 13 (after Figure 13-3), insert the following:

It is possible to construct cyclic Gray codes for rotational sensors that require only one ring of conducting and nonconducting areas, at some expense in resolution for a given number of brushes. The brushes are spaced around the ring, rather than on a radial line. These codes are called *single track Gray codes*, or STGCs.

The idea is to find a code for which, when written out as in Figure 13-1, every column is a rotation of the first column (and which is cyclic, assuming the code is for a rotational device). The reflected Gray code for $n = 2$ is trivially an STGC. STGCs for $n = 2$ through 4 are shown below.

$n = 2$	$n = 3$	$n = 4$
00	000	0000
01	001	0001
11	011	0011
10	111	0111
	110	1111
	100	1110
		1100
		1000

STGCs allow the construction of more compact rotational position sensors. A rotational STGC device for $n = 3$ is shown in Figure 13-4.

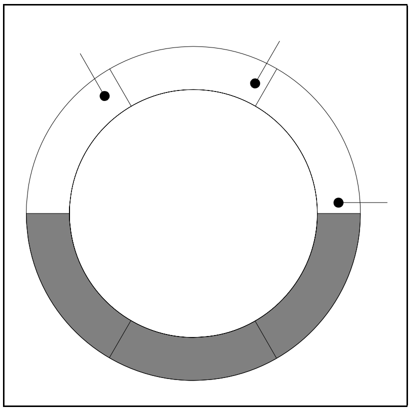


FIGURE 13–4. Single track rotational position sensor.

These are all very similar, simple, and rather uninteresting patterns. Following these patterns, an STGC for the case $n = 5$ would have 10 code words, giving a resolution of 36 degrees. However, it is possible to do much better. Figure 13–5 shows an STGC for $n = 5$ with 30 code words, giving a resolution of 12 degrees. It is close to the optimum of 32 codewords.

10000	01000	00100	00010	00001
10100	01010	00101	10010	01001
11100	01110	00111	10011	11001
11110	01111	10111	11011	11101
11010	01101	10110	01011	10101
11000	01100	00110	00011	10001

FIGURE 13–5. An STGC for $n = 5$.

All the STGCs shown above are the best possible, in the sense that for $n = 2$ through 5, the largest number of code words possible is 4, 6, 8, and 30.

According to an article in Wikipedia, an STGC has been constructed with exactly 360 code words, with $n = 9$ (the smallest possible, because any code for $n = 8$ has at most 256 code words).

- - -

Page 259, after the last paragraph, add:

The Hilbert curve has been used to assign jobs to processors when the processors are interconnected in a rectangular 2D or 3D grid [Cplant]. The processor allocation system software uses a linear list of the processors that follows a Hilbert curve over the grid. When a job that requires a number of processors is scheduled to run, the allocator allocates them from the linear list, much as a memory allocator would do. The allocated processors tend to be close together on the grid, which leads to good intercommunication properties.

- - -

Page 263, just before Section 15–2, insert the following section:

15–2 Floating-Point To/From Integer Conversions

Table 15–2 gives some formulas for conversion between IEEE floating-point format and integers. These methods are concise and fast, but they do not give the correct result for the full range of input values. The ranges over which they do give the precisely correct result are given in the table. They all give the correct result for ±0.0 and for denorms within the stated ranges. Most do not give a reasonable result for a NaN or infinity. These formulas may be suitable for direct use in some applications, or in a library routine to get the common cases quickly.

The Type column denotes the type of conversion desired, including the rounding mode: n for round to nearest even, d for round down, u for round up, and z for round toward zero. The R column denotes the rounding mode that the machine must be in for the formula to give the correct result. (On some machines, such as the Intel IA-32, the rounding mode can be specified in the instruction itself, rather than in a “mode” register.)

A “double” is an IEEE double, which is 64 bits in length. A “float” is an IEEE single, which is 32 bits in length.

The notation “ulp” means one unit in the last position. For example, 1.0 – ulp denotes the IEEE-format number that is closest to 1.0 but less than 1.0, something like 0.99999…. The notation “int64” denotes a signed 64-bit integer (two’s complement), and “int32” denotes a signed 32-bit integer. “uint64” and “uint32” have similar meanings, but for unsigned interpretations.

The function low32($\boldsymbol{x}$) extracts the low-order 32 bits of $\boldsymbol{x}$.

The operators $\overset{d}{+}$ and $\overset{s}{+}$ denote double- and single-precision floating-point addition, respectively. Similarly, the operators $\overset{d}{-}$ and $\overset{s}{-}$ denote double- and single-precision subtraction.

TABLE 15–2. FLOATING-POINT CONVERSIONS

Type	R	Formula	Range	Notes
Double to int64, n	n	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521}) - \boldsymbol{c}_{521}$	-2^{51} to $2^{51} + 0.5$	1
Double to int64, d	d	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521}) - \boldsymbol{c}_{521}$	$-2^{51} - 0.5$ to $2^{51} + 0.5$	1

TABLE 15–2. FLOATING-POINT CONVERSIONS

Type	R	Formula	Range	Notes
Double to int64, u	u	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521}) - \boldsymbol{c}_{521}$	-2^{51} to $2^{51} + 1$	1

Double to int64, z	d or z	if $(\boldsymbol{x} \geq 0.0)$
		$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{52}) - \boldsymbol{c}_{52}$
		else
		$\boldsymbol{c}_{52} - (\boldsymbol{c}_{52} \overset{d}{-} \boldsymbol{x})$
			-2^{52} to 2^{52}	1

Double to uint64, n	n	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{52}) - \boldsymbol{c}_{52}$	-0.25 to 2^{52}	1
Double to uint64, d	d	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{52}) - \boldsymbol{c}_{52}$	0 to 2^{52}	1
Double to uint64, u	u	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{52}) - \boldsymbol{c}_{52}$	-0.5 + ulp to $2^{52} + 1$	1
Double to int32 or uint32, n	n	low32$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521})$	$-2^{31} - 0.5$ to $2^{31} - 0.5$ – ulp, or -0.5 to $2^{32} - 0.5$ – ulp	1
Double to int32 or uint32, d	d	low32$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521})$	-2^{31} to 2^{31} – ulp, or 0 to 2^{32} – ulp	1
Double to int32 or uint32, u	u	low32$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521})$	$-2^{31} - 1$ + ulp to $2^{31} - 1$, or -1 + ulp to $2^{32} - 1$	1

Double to int32 or uint32, z	d or z	if $(\boldsymbol{x} \geq \mathbf{0.0})$
		low32$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521})$
		else
		$-$low32$(\boldsymbol{c}_{521} \overset{d}{-} \boldsymbol{x})$
			$-2^{31} - 1$ + ulp to 2^{31} – ulp,
			or -1 + ulp to 2^{32} – ulp
				1

Float to int32, n	n	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{231}) - \boldsymbol{c}_{231}$	-2^{22} to $2^{22} + 0.5$	
Float to int32, d	d	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{231}) - \boldsymbol{c}_{231}$	$-2^{22} - 0.5$ to $2^{22} + 0.5$	
Float to int32, u	u	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{231}) - \boldsymbol{c}_{231}$	-2^{22} to $2^{22} + 1$	

Float to int32, z	d or z	if $(\boldsymbol{x} \geq \mathbf{0.0})$
		$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{23}) - \boldsymbol{c}_{23}$
		else
		$\boldsymbol{c}_{23} - (\boldsymbol{c}_{23} \overset{s}{-} \boldsymbol{x})$
			-2^{23} to 2^{23}	

Float to uint32, n	n	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{23}) - \boldsymbol{c}_{23}$	-0.25 to 2^{23}	

TABLE 15–2. FLOATING-POINT CONVERSIONS

Type	R	Formula	Range	Notes
Float to uint32, d	d	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{23}) - \boldsymbol{c}_{23}$	0 to 2^{23}	
Float to uint32, u	u	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{23}) - \boldsymbol{c}_{23}$	$-0.5 + \text{ulp}$ to $2^{23} + 1$	
Round double to nearest	n	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{521}) \overset{d}{-} \boldsymbol{c}_{521}$	-2^{51} to $2^{51} + 0.5$	1
Round non-negative double to nearest	n	$(\boldsymbol{x} \overset{d}{+} \boldsymbol{c}_{52}) \overset{d}{-} \boldsymbol{c}_{52}$	-0.25 to 2^{52}	1, 3
Round float to nearest	n	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{231}) \overset{s}{-} \boldsymbol{c}_{231}$	-2^{22} to $2^{22} + 0.5$	2
Round non-negative float to nearest	n	$(\boldsymbol{x} \overset{s}{+} \boldsymbol{c}_{23}) \overset{s}{-} \boldsymbol{c}_{23}$	-0.25 to 2^{23}	2, 3
Int64 to double	–	$(\boldsymbol{x} + \boldsymbol{c}_{521}) \overset{d}{-} \boldsymbol{c}_{521}$	-2^{51} to 2^{51}	4
Uint64 to double	–	$(\boldsymbol{x} + \boldsymbol{c}_{52}) \overset{d}{-} \boldsymbol{c}_{52}$	0 to $2^{52} - 1$	4
Int32 to float	–	$(\boldsymbol{x} + \boldsymbol{c}_{231}) \overset{s}{-} \boldsymbol{c}_{231}$	-2^{22} to 2^{22}	
Uint32 to float	–	$(\boldsymbol{x} + \boldsymbol{c}_{23}) \overset{s}{-} \boldsymbol{c}_{23}$	0 to 2^{23}	

Constants:

$\boldsymbol{c}_{521}$ = 0x4338 0000 0000 0000 = $2^{52} + 2^{51}$
$\boldsymbol{c}_{52}$ = 0x4330 0000 0000 0000 = 2^{52}
$\boldsymbol{c}_{231}$ = 0x4B40 0000 = $2^{23} + 2^{22}$
$\boldsymbol{c}_{23}$ = 0x4B00 0000 = 2^{23}

Notes:

1. The floating-point operations must be done in IEEE double-precision (53 bits of precision), and no more. Most Intel machines do not, by default, operate in this mode. On those machines it is necessary to set the precision (PC field in the FPU Control Word) to double-precision.
2. The floating-point operations must be done in IEEE single-precision (24 bits of precision), and no more. Most Intel machines are not, by default, operated in this mode. On those machines it is necessary to set the precision (PC field in the FPU Control Word) to single-precision.

3. “Nonnegative” means –0.0 or greater than or equal to 0.0.
4. To convert a 32-bit signed or unsigned integer to double, sign- or zero-extend the 32-bit integer to 64 bits, and use the appropriate one of these formulas.

It might seem curious that on most Intel machines, the double to integer (of any size) conversions require that the machine’s precision mode be reduced to 53 bits, whereas for *float* to integer conversions, the reduction in precision is not necessary—the correct result is obtained with the machine running in extended-precision mode (64 bits of precision). This is because for the double-precision add of the constant, the fraction might be shifted right as many as 52 bits, which may cause 1-bits to be shifted beyond the 64-bit limit, and hence lost. Thus two roundings occur—first to 64 bits and then to 53 bits. On the other hand, for the single-precision add of the constant, the maximum shift is 23 bits. With that small shift amount, no bit can be shifted beyond the 64-bit boundary, so that only one rounding operation occurs. The conversions from float to integer get the correct result on Intel machines in all three precision modes.

On Intel machines running in extended-precision mode, the conversions from double to int64 and uint64 may be done without changing the precision mode by using different constants and one more floating-point operation. The calculation is

$$((\boldsymbol{x} \overset{e}{+} \boldsymbol{c}_1) \overset{e}{-} \boldsymbol{c}_2) - \boldsymbol{c}_3,$$

where $\overset{e}{+}$ and $\overset{e}{-}$ denote extended-precision addition and subtraction, respectively. (The result of the *add* must remain in the 80-bit register for use by the extended-precision subtract operation.)

For double to int64,

$\boldsymbol{c}_1$ = 0x43E0 0300 0000 0000 = $2^{63} + 2^{52} + 2^{51}$
$\boldsymbol{c}_2$ = 0x43E0 0000 0000 0000 = 2^{63}
$\boldsymbol{c}_3$ = 0x4338 0000 0000 0000 = $2^{52} + 2^{51}$

For double to uint64,

$\boldsymbol{c}_1$ = 0x43E0 0200 0000 0000 = $2^{63} + 2^{52}$
$\boldsymbol{c}_2$ = 0x43E0 0000 0000 0000 = 2^{63}
$\boldsymbol{c}_3$ = 0x4330 0000 0000 0000 = 2^{52}

Using these constants, similar expressions can be derived for the conversion and rounding operations shown in Table 15–2 that are flagged by Note 1. The ranges of applicability are close to those shown in the table.

However, for the round double to nearest operation, if the calculation subtracts first and then adds, that is,

$$((\boldsymbol{x} \overset{e}{-} \boldsymbol{c}_1) \overset{e}{+} \boldsymbol{c}_2) + \boldsymbol{c}_3,$$

(using the first set of constants above), then the range for which the correct result is obtained is $-2^{51} - 0.5$ to ∞, but not a NaN.

- - -

Page 264, just before the book's Section 15–3, The Distribution of Leading Digits, insert the following section:

15–4 An Approximate Reciprocal Square Root Routine

In the early 2000s there was some buzz in programming circles about an amazing routine for computing an approximation to the reciprocal square root of a number in IEEE single format. The routine is useful in graphics applications, for example to normalize a vector by multiplying its components x, y, and z by $1/\sqrt{x^2 + y^2 + z^2}$. C code for the function is shown in Figure 15–1 [Taro].

```
float rsqrt(float x) {
   float xhalf = 0.5f*x;
   int i = *(int *)&x;          // View x as an int.
   i = 0x5f375a82 - (i >> 1);   // Initial guess.
   x = *(float *)&i;            // View i as float.
   x = x*(1.5f - xhalf*x*x);    // Newton step.
   return x;
}
```

FIGURE 15–1. Approximate reciprocal square root.

The relative error of the result is in the range 0 to –0.00176 for all normalized single-precision numbers (it errs on the low side). It gives the correct IEEE result (NaN) if its argument is a NaN. However, it gives an unreasonable result if its argument is $\pm\infty$, a negative number, or –0. If the argument is +0 or a positive denorm, the result is not what it should be, but it is a large number (greater than 9×10^{18}), which might be acceptable in some applications.

The relative error can be reduced in magnitude, to the range ±0.000892, by changing the constant 1.5 in the Newton step to 1.5008908.

Another possible refinement is to replace the multiplication by 0.5 with a subtract of 1 from the exponent of `x`. That is, replace the definition of `xhalf` with

```
   int ihalf = *(int *)&x - 0x00800000;
   float xhalf = *(float *)&ihalf;
```

However, the function then gives inaccurate results (although greater than 6×10^{18}) for `x` a normalized number less than about 2.34×10^{-38}, and NaN for `x` a denormalized number. For `x` = 0 the result is $+\infty$ (which is correct).

The Newton step is a standard Newton-Raphson calculation for the reciprocal square root function (see Appendix B). Simply repeating this step reduces the relative error to the range 0 to –0.0000047. The optimal constant for this is 0x5F37599E.

On the other hand, deleting the Newton step results in a substantially faster function with a relative error within ±0.035, using a constant of 0x5F37642F. It consists of only two integer instructions plus code to load the constant. (The variable `xhalf` can be deleted.)

To get an inkling of why this works, suppose $x = 2^n(1 + f)$, where n is the unbiased exponent and f is the fraction $(0 \le f < 1)$. Then

$$\frac{1}{\sqrt{x}} = 2^{-n/2}(1 + f)^{-1/2}.$$

Ignoring the fraction, this shows that we must change the biased exponent from $127 + n$ to $127 - n/2$. If $e = 127 + n$, then $127 - n/2 = 127 - (e - 127)/2 = 190.5 - e/2$. Therefore it appears that a calculation something like shifting x right one position and subtracting it from 190 in the exponent position might give a very rough approximation to $1/\sqrt{x}$. In C, this can be expressed as[5]

```
0x5F000000 - (*(int *)&x >> 1)
```

To find a better value for the constant 0x5F000000 by analysis is difficult. Four cases must be analyzed: the cases in which a 0-bit or a 1-bit is shifted from the exponent field to the fraction field, and the cases in which the subtraction does or does not generate a borrow that propagates to the exponent field. This analysis is done in [Lomo]. Here, we make some simple observations.

Using rep(x) to denote the representation of the floating-point number x in IEEE single format, we want a formula of the form

$$\text{rep}(1/\sqrt{x}) \approx k - (\text{rep}(x) \overset{s}{\gg} 1)$$

for some constant k. (Whether the shift is signed or unsigned makes no difference, because we exclude negative values of x and –0.) We can get an idea of roughly what k should be from

$$k \approx \text{rep}(1/\sqrt{x}) + (\text{rep}(x) \overset{s}{\gg} 1),$$

5. This is not officially sanctioned C, but with most compilers it works.

and trying a few values of x. The results are shown in Table 15–3 (in hexadecimal).

TABLE 15–3. DETERMINING THE CONSTANT

Trial x	**rep(x)**	**rep($1/\sqrt{x}$)**	**k**
1.0	`3F800000`	`3F800000`	`5F400000`
1.5	`3FC00000`	`3F5105EC`	`5F3105EC`
2.0	`40000000`	`3F3504F3`	`5F3504F3`
2.5	`40200000`	`3F21E89B`	`5F31E89B`
3.0	`40400000`	`3F13CD3A`	`5F33CD3A`
3.5	`40600000`	`3F08D677`	`5F38D677`
4.0	`40800000`	`3F000000`	`5F400000`

It looks like k is approximately a constant. Notice that the same value is obtained for $x = 1.0$ and 4.0. In fact, the same value of k results from any number x and $4x$ (provided they are both normalized numbers). This is because, in the formula for k, if x is quadrupled, then the term $\text{rep}(1/\sqrt{x})$ decreases by 1 in the exponent field, and the term $\text{rep}(x) \overset{s}{\gg} 1$ increases by 1 in the exponent field.

More significantly, the relative errors for x and $4x$ are exactly the same, provided both quantities are normalized. To see this, it can be shown that if the argument `x` of the `rsqrt` function is quadrupled, the result of the function is exactly halved, and this is true no matter how many Newton steps are done. Of course, $1/\sqrt{x}$ is also halved. Therefore, the relative error is unchanged.

This is important, because it means that if we find an optimal value (by some criterion, such as minimizing the maximum absolute value of the error) for values of x in the range 1.0 to 4.0, then the same value of k is optimal for all normalized numbers.

It is then a straightforward task to write a program that, for a given value of k, calculates the true value of $1/\sqrt{x}$ (using a known accurate library routine) and the estimated value for some 10,000 or so values of x from 1.0 to 4.0, and calculates the maximum error. The optimal value of k can be determined by hand, which is tedious but sometimes illuminating. It is quite amazing that there is a constant for which the error is less than ±3.5% in a function that uses only two integer operations and no table lookup.

- - -

Add the references:

[Agrell] Agrell, Erik. http://www.s2.chalmers.se/~agrell/bounds/, October 2003.

[Allen] Allen, Joseph H. Private communication.

[Arndt] Arndt, Jörg. *Matters Computational: Ideas, Algorithms, Source Code.* Springer-Verlag, 2010. Also available at http://www.jjj.de/fxt/#fxtbook.

[Baum] D. E. Knuth attributes the ternary method to an unpublished memo from the mid-1970s by Bruce Baumgart, which compares about 20 different methods for bit reversal on the PDP10.

[Black] Black, Richard. Web site www.cl.cam.ac.uk/Research/SRG/bluebook/21/crc/crc.html. University of Cambridge Computer Laboratory Systems Research Group, February 1994.

[Bonz] Bonzini, Paolo. Private communication.

[Brou] Brouwer, Andries E. http://www.win.tue.nl/~aeb/binary-1.html, March 2004.

[CavWer] Cavagnino, D. and Werbrouck, A. E. "Efficient Algorithms for Integer Division by Constants Using Multiplication." *The Computer Journal 51*, 4 (2008), 470–480.

[CC] Caldwell, Chris K. and Cheng, Yuanyou. "Determining Mills' Constant and a Note on Honaker's Problem." *Journal of Integer Sequences 8*, 4 (2005), article 05.4.1, 9 pp. Also available at http://www.cs.uwaterloo.ca/journals/JIS/VOL8/Caldwell/caldwell78.pdf.

[Cplant] Leung, Vitus J., et al: "Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies." In *Proceedings 4th IEEE International Conference on Cluster Computing*, September 2002, 296–304.

[Dalton] Dalton, Michael. Private communication.

[Danne] Dannemiller, Christopher M. Private communication. He attributes this code to the Linux Source base, www.gelato.unsw.edu.au/lxr/source/lib/crc32.c, lines 105–111.

[Dietz] Dietz, Henry G. http://aggregate.org/MAGIC/.

[Ditlow] Ditlow, Gary S. Private communication.

[Dubé] Dubé, Danny. Newsgroup comp.compression.research, October 3, 1997.

[Etzion] Etzion, Tuvi. "Constructions for Perfect 2-Burst-Correcting Codes," *IEEE Transactions on Information Theory 47*, 6 (September 2001), 2553–2555.

[Floyd] Floyd, Robert W. "Permuting Information in Idealized Two-Level Storage." In *Complexity of Computer Computations* (Conference proceedings), Plenum Press, 1972, 105–109. This is the earliest reference I know of for this method of transposing a $2^n \times 2^n$ matrix.

[Gaud] Gaudet, Dean. Private communication.

[Gor] Goryavsky, Julius. Private communication.

[Ham] Hamming, Richard W., "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal 26*, 2 (April 1950), 147–160. NB: We interchange the roles of the variables *k* and *m* relative to how they are used by Hamming. We follow the more modern usage found in [LC] and [MS], for example.

[Harley] Harley, Robert. Newsgroup comp.arch, July 12, 1996.

[Hill] Hill, Raymond. *A First Course in Coding Theory.* Clarendon Press, 1986.

[Hsieh] Hsieh, Paul. Newsgroup comp.lang.c, April 29, 2005.

[Huef] Hueffner, Falk. Private communication.

[Karat] Karatsuba, A. and Ofman, Yu. Multiplication of multidigit numbers on automata. *Soviet Physics-Doklady 7*, 7 (January 1963), 595–596. They show the theoretical result that multiplication of *m*-bit integers is $O(m^{\log_2 3}) \approx O(m^{1.585})$, but the details of their method are more cumbersome than the method based on Gauss's three-multiplication scheme for complex numbers.

[Karv] Karvonen, Vesa. Found at "The Assembly Gems" web page, www.df.lth.se/~john_e/fr_gems.html.

[Keane] Keane, Joe. Newsgroup sci.math.num-analysis, July 9, 1995.

[Knu4] Knuth, Donald E. *MMIXware, A RISC Computer for the Third Millennium*, Springer (Lecture Notes in Computer Science; 1750), 1999.

[Knu5] Knuth, Donald E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part I,* Addison-Wesley, January 2011, Section 7.1.1.

[Knu6] *Ibid*, Section 7.1.3. Knuth attributes the equality relation to W. C. Lynch in 2006.

[Knu7] *Ibid*, Section 7.2.1.1 Exercise 80.

[Knu8] Knuth, Donald E. Private communication.

[Kumar] This figure was suggested by Gowri Kumar (private communication).

[Lang] Langdon, Glen G. Jr., "Subtraction by Minuend Complementation," *IEEE Transactions on Computers C-18*, 1 (January 1969), 74–76.

[LC] Lin, Shu and Costello, Daniel J., Jr. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[Lomo] Lomont, Chris. *Fast Inverse Square Root.* www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

[LPR] Leiserson, Charles E., Prokop, Harald, and Randall, Keith H. *Using de Bruijn Sequences to Index a 1 in a Computer Word*. MIT Laboratory for Computer Science, July 7, 1998. Also available at http://supertech.csail.mit .edu/papers/debruijn.pdf.

[MS] MacWilliams, Florence J. and Sloane, Neil. J. A. *The Theory of Error-Correcting Codes Part II.* North-Holland, 1977.

[Mycro] Mycroft, Alan. Newsgroup comp.arch, April 8, 1987.

[Neum] Neumann, Jasper L. Private communication.

[PeBr] Peterson, W. W. and Brown, D.T. "Cyclic Codes for Error Detection." In *Proceedings of the IRE*, January 1961, 228–235.

[Reiser] Reiser, John. Newsgroup comp.arch.arithmetic, December 11, 1998.

[Roman] Roman, Steven. *Coding and Information Theory.* Springer-Verlag, 1992.

[Seal1] Seal, David. Newsgroup comp.arch.arithmetic, May 13, 1997. Harley was the first known to this writer to apply the CSA to this problem, and Seal showed a particularly good way to use it for counting the bits in a large array (as illustrated in Figures 5–8 and 5–9), and also for an array of size seven (similar to the plan of Figure 5–10).

[Seal2] Seal, David. Newsgroup comp.sys.acorn.tech, February 16, 1994.

[Strach] Strachey, Christopher. Bitwise Operations. *Communications of the ACM 4*, 3 (March 1961), 146. This issue contains another paper which gives two methods for bit reversal (Two Methods for Word Inversion on the IBM 709, by Robert A. Price and Paul Des Jardins; there is a small correction on page A13 of the March 1961 issue). These methods are not discussed in this book because they rely on the somewhat exotic *Convert by Addition from the MQ* (CAQ) instruction of that machine. That instruction does a series of indexed table lookups, adding the word fetched from memory to the accumulator. It is not a RISC instruction.

[Tanen] Tanenbaum, Andrew S. *Computer Networks*, Second Edition. Prentice Hall, 1988.

[Taro] The author of this program seems to be lost in history. One of the earliest people to use it and to tweak the constant a bit was Gary Tarolli, probably while he was at SGI. He also helped to make it more widely known, and says it goes back to 1995 or earlier. For more on the history see http://www.beyond3d.com/content/articles/8/.

[Wood] Woodrum, Luther. Private communication. The second formula uses no literals and works well on the IBM System/370.

[Zadeck] Zadeck, F. Kenneth. Private communication.