Division by Invariant Integers using Multiplication

Torbjörn Granlund*
Cygnus Support
1937 Landings Drive
Mountain View, CA 94043–0801
tege@cygnus.com

Peter L. Montgomery†
Centrum voor Wiskunde en Informatica
780 Las Colindas Road
San Rafael, CA 94903–2346
pmontgom@math.orst.edu

Abstract

Integer division remains expensive on today's processors as the cost of integer multiplication declines. We present code sequences for division by arbitrary nonzero integer constants and run–time invariants using integer multiplication. The algorithms assume a two's complement architecture. Most also require that the upper half of an integer product be quickly accessible. We treat unsigned division, signed division where the quotient rounds towards zero, signed division where the quotient rounds towards $-\infty$, and division where the result is known *a priori* to be exact. We give some implementation results using the C compiler GCC.

1 Introduction

The cost of an integer division on today's RISC processors is several times that of an integer multiplication. The trend is towards fast, often pipelined combinatoric multipliers that perform an operation in typically less than 10 cycles, with either no hardware support for integer division or iterating dividers that are several times slower than the multiplier.

Table 1.1 compares multiplication and division times on some processors. This table illustrates that the discrepancy between multiplication and division timing has been growing.

Integer division is used heavily in base conversions, number theoretic codes, and graphics codes. Compilers generate integer divisions to compute loop counts and subtract pointers. In a static analysis of FORTRAN programs, Knuth [13, p. 9] reports that 39% of 46466 arithmetic operators were additions, 22% subtractions, 27% multiplications, 10% divisions, and 2% exponentiations. Knuth's counts do not distinguish integer and floating point operations, except that 4% of the divisions were divisions by 2.

When integer multiplication is cheaper than integer division, it is beneficial to substitute a multiplication for a division. Multiple authors [2, 11, 15] present algorithms for division by constants, but only when the divisor divides $2^k - 1$ for some small k. Magenheimer et al [16, §7] give the foundation of a more general approach, which Alverson [1] implements on the Tera Computing System. Compiler writers are only beginning to become aware of the general technique. For example, version 1.02 of the IBM RS/6000 xlc and xlf compilers uses the integer multiply instruction to expand signed integer divisions by 3, 5, 7, 9, 25, and 125, but not by other odd integer divisors below 256, and never for unsigned division.

We assume an N–bit two's complement architecture. Unsigned (i.e., nonnegative) integers range from 0 to $2^N - 1$ inclusive; signed integers range from -2^{N-1} to $2^{N-1} - 1$. We denote these integers by **uword** and **sword** respectively. Unsigned doubleword integers (range 0 to $2^{2N} - 1$) are denoted by **udword**. Signed doubleword integers (range -2^{2N-1} to $2^{2N-1} - 1$) are denoted by **sdword**. The type **int** is used for shift counts and logarithms.

Several of the algorithms require the upper half of an integer product obtained by multiplying two **uword**s or two **sword**s. All algorithms need simple operations such as adds, shifts, and bitwise operations (bit–ops) on **uword**s and **sword**s, as summarized in Table 3.1.

We show how to use these operations to divide by arbitrary nonzero constants, as well as by divisors which are loop invariant or repeated in a basic block, using one multiplication plus a few simple instructions per division. The presentation concentrates on three types of

*Work done by first author while at Swedish Institute of Computer Science, Stockholm, Sweden.

†Work done by second author while at University of California, Los Angeles. Supported by U.S. Army fellowship DAAL03–89–G–0063.

Architecture/Implementation	N	Approx. Year	Time (cycles) for HIGH(N–bit $*$ N–bit)	Time (cycles) for N–bit/N–bit divide
Motorola MC68020 [18, pp. 9–22]	32	1985	41–44	76–78 (unsigned) 88–90 (signed)
Motorola MC68040	32	1991	20	44
Intel 386 [9]	32	1985	9–38	38
Intel 486 [10]	32	1989	13–42	40
Intel Pentium	32	1993	10	46
SPARC Cypress CY7C601	32	1989	40^{S}	100^{S}
SPARC Viking [20]	32	1992	5	19
HP PA 83 [16]	32	1985	45^{S}	70^{S}
HP PA 7000	32	1990	3^{FP}	70^{S}
MIPS R3000 [12]	32	1988	12^{P}	35^{P}
MIPS R4000 [17]	32	1991	12^{P}	75
	64		20^{P}	139
POWER/RIOS I [4, 22]	32	1989	5 (signed only)	19 (signed only)
PowerPC/MPC601 [19]	32	1993	5–10	36
DEC Alpha 21064AA [8]	64	1992	23^{P}	200^{S}
Motorola MC88100	32	1989	17^{S}	38
Motorola MC88110	32	1992	3^{P}	18

$^{\mathrm{S}}$ – No direct hardware support; approximate cycle count for software implementation
$^{\mathrm{F}}$ – Does not include time for moving data to/from floating point registers
$^{\mathrm{P}}$ – Pipelined implementation (i.e., independent instructions can execute simultaneously)

Table 1.1: Multiplication and division times on different CPUs

division, in order by difficulty: (i) unsigned, (ii) signed, quotient rounded towards zero, (iii) signed, quotient rounded towards $-\infty$. Other topics are division of a **udword** by a run–time invariant **uword**, division when the remainder is known *a priori* to be zero, and testing for a given remainder. In each case we give the mathematical background and suggest an algorithm which a compiler can use to generate the code.

The algorithms are ineffective when a divisor is not invariant, such as in the Euclidean GCD algorithm.

Most algorithms presented herein yield only the quotient. The remainder, if desired, can be computed by an additional multiplication and subtraction.

We have implemented the algorithms in a developmental version of the GCC 2.6 compiler [21]. DEC uses some of these algorithms in its Alpha AXP compilers.

2 Mathematical notations

Let x be a real number. Then $\lfloor x \rfloor$ denotes the largest integer not exceeding x and $\lceil x \rceil$ denotes the least integer not less than x. Let TRUNC(x) denote the integer part of x, rounded towards zero. Formally, TRUNC(x) $= \lfloor x \rfloor$ if $x \geq 0$ and TRUNC(x) $= \lceil x \rceil$ if $x < 0$. The absolute value of x is $|x|$. For $x > 0$, the (real) base 2 logarithm of x is $\log_2 x$.

A multiplication is written $x * y$.

If x, y, and n are integers and $n \neq 0$, then $x \equiv y \pmod{n}$ means $x - y$ is a multiple of n.

Two remainder operators are common in language definitions. Sometimes a remainder has the sign of the dividend and sometimes the sign of the divisor. We use the Ada notations

$$\begin{aligned} n \ \mathbf{rem}\ d &= n - d * \mathrm{TRUNC}(n/d) && \text{(sign of dividend)}, \\ n \ \mathbf{mod}\ d &= n - d * \lfloor n/d \rfloor && \text{(sign of divisor)}. \end{aligned} \tag{2.1}$$

The Fortran 90 names are MOD and MODULO. In C, the definition of remainder is implementation–dependent (many C implementations round signed quotients towards zero and use **rem** remaindering). Other definitions have been proposed [6, 7].

If n is an **udword** or **sdword**, then HIGH(n) and LOW(n) denote the most significant and least significant halves of n. LOW(n) is a **uword**, while HIGH(n) is an **uword** if n is a **udword** and an **sword** if n is a **sdword**. In both cases $n = 2^N * \mathrm{HIGH}(n) + \mathrm{LOW}(n)$.

3 Assumed instructions

The suggested code assumes the operations in Table 3.1, on an N–bit machine. Some primitives, such as loading constants and operands, are implicit in the notation and are not included in the operation counts.

TRUNC(x)	Truncation towards zero; see §2.
HIGH(x), LOW(x)	Upper and lower halves of x: see §2.
MULL(x, y)	Lower half of product $x * y$ (i.e., product modulo 2^N).
MULSH(x, y)	Upper half of signed product $x * y$: If $-2^{N-1} \le x$, $y \le 2^{N-1} - 1$, then $x * y = 2^N * \text{MULSH}(x, y) + \text{MULL}(x, y)$.
MULUH(x, y)	Upper half of unsigned product $x * y$: If $0 \le x$, $y \le 2^N - 1$, then $x * y = 2^N * \text{MULUH}(x, y) + \text{MULL}(x, y)$.
AND(x, y)	Bitwise AND of x and y.
EOR(x, y)	Bitwise exclusive OR of x and y.
NOT(x)	Bitwise complement of x. Equal to $-1 - x$ if x is signed, to $2^N - 1 - x$ if x is unsigned.
OR(x, y)	Bitwise OR of x and y.
SLL(x, n)	Logical left shift of x by n bits ($0 \le n \le N - 1$).
SRA(x, n)	Arithmetic right shift of x by n bits ($0 \le n \le N - 1$).
SRL(x, n)	Logical right shift of x by n bits ($0 \le n \le N - 1$).
XSIGN(x)	-1 if $x < 0$; 0 if $x \ge 0$. Short for SRA(x, $N - 1$) or $-$SRL(x, $N - 1$).
$x + y$, $x - y$, $-x$	Two's complement addition, subtraction, negation.

Table 3.1: Mathematical notations and primitive operations

The algorithm in §8 requires the ability to add or subtract two doublewords, obtaining a doubleword result; this typically expands into 2–4 instructions.

The algorithms for processing constant divisors require compile–time arithmetic on **udword**s.

Algorithms for processing run–time invariant divisors require taking the base 2 logarithm of a positive integer (sometimes rounded up, sometimes down) and require dividing a **udword** by a **uword**. If the algorithms are used only for constant divisors, then these operations are needed only at compile time. If the architecture has a leading zero count (LDZ) instruction, then these logarithms can be found from

$$\lceil \log_2 x \rceil = N - \text{LDZ}(x - 1),$$
$$\lfloor \log_2 x \rfloor = N - 1 - \text{LDZ}(x) \qquad (1 \le x \le 2^N - 1).$$

Some algorithms may produce expressions such as SRL(x, 0) or $-(x - y)$; the optimizer should make the obvious simplifications. Some descriptions show an addition or subtraction of 2^N, which is a no-op.

If an architecture lacks arithmetic right shift, then it can be computed from the identity

$$\text{SRA}(x, \ell) = \text{SRL}(x + 2^{N-1}, \ell) - 2^{N-1-\ell}$$

whenever $0 \le \ell \le N - 1$.

If an architecture has only one of MULSH and MULUH, then the other can be computed using

$$\text{MULUH}(x, y) = \text{MULSH}(x, y) + \text{AND}(x, \text{XSIGN}(y)) + \text{AND}(y, \text{XSIGN}(x))$$

for arbitrary N–bit patterns x, y (interpreted as **uword**s for MULUH and as **sword**s for MULSH).

4 Unsigned division

Suppose we want to compile an unsigned division $q = \lfloor n/d \rfloor$, where $0 < d < 2^N$ is a constant or run–time invariant and $0 \le n < 2^N$ is variable. Let's try to find a rational approximation $m/2^{N+\ell}$ of $1/d$ such that

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{m * n}{2^{N+\ell}} \right\rfloor \quad \text{whenever} \quad 0 \le n \le 2^N - 1. \tag{4.1}$$

Setting $n = d$ in (4.1) shows we require $2^{N+\ell} \le m * d$. Setting $n = q * d - 1$ shows $2^{N+\ell} * q > m * (q * d - 1)$. Multiply by d to derive $\left(m * d - 2^{N+\ell}\right) * (q * d - 1) < 2^{N+\ell}$. This inequality will hold for all values of $q*d-1$ below 2^N if $m * d - 2^{N+\ell} \le 2^\ell$. Theorem 4.2 below states that these conditions are sufficient, because the maximum relative error (1 part in 2^N) is too small to affect the quotient when $n < 2^N$.

Theorem 4.2 *Suppose m, d, ℓ are nonnegative integers such that $d \ne 0$ and*

$$2^{N+\ell} \le m * d \le 2^{N+\ell} + 2^\ell. \tag{4.3}$$

Then $\lfloor n/d \rfloor = \lfloor m * n/2^{N+\ell} \rfloor$ for every integer n with $0 \le n < 2^N$.

Proof. Define $k = m * d - 2^{N+\ell}$. Then $0 \le k \le 2^\ell$ by hypothesis. Given n with $0 \le n < 2^N$, write $n = q * d + r$ where $q = \lfloor n/d \rfloor$ and $0 \le r \le d - 1$. We must show that $q = \lfloor m * n/2^{N+\ell} \rfloor$. A calculation gives

$$\begin{aligned} \frac{m * n}{2^{N+\ell}} - q &= \frac{k + 2^{N+\ell}}{d} * \frac{n}{2^{N+\ell}} - q \\ &= \frac{k * n}{d * 2^{N+\ell}} + \frac{n}{d} - \frac{n - r}{d} \\ &= \frac{k}{2^\ell} * \frac{n}{2^N} * \frac{1}{d} + \frac{r}{d}. \end{aligned} \tag{4.4}$$

This difference is nonnegative and does not exceed $1 * \frac{2^N - 1}{2^N} * \frac{1}{d} + \frac{d-1}{d} = 1 - \frac{1}{2^N * d} < 1$. ■

Theorem 4.2 allows division by d to be replaced with multiplication by $m/2^{N+\ell}$ if (4.3) holds. In general we require $2^\ell \geq d-1$ to ensure that a suitable multiple of d exists in the interval $[2^{N+\ell},\ 2^{N+\ell}+2^\ell]$. For compatibility with the algorithms for signed division (§5 and §6), it is convenient to choose $m * d > 2^{N+\ell}$ even though Theorem 4.2 permits equality. Since m can be almost as large as 2^{N+1}, we don't multiply by m directly, but instead by 2^N and $m - 2^N$. This leads to the code in Figure 4.1. Its cost is 1 multiply, 2 adds/subtracts, and 2 shifts per quotient, after computing constants dependent only on the divisor.

```
Initialization (given uword d with 1 ≤ d < 2^N):
        int ℓ = ⌈log2 d⌉;
                        /* d ≤ 2^ℓ ≤ 2*d − 1 */
        uword m' = ⌊2^N * (2^ℓ − d)/d⌋ + 1;
                    /* m' = ⌊2^(N+ℓ)/d⌋ − 2^N + 1 */
        int sh1 = min(ℓ, 1);
        int sh2 = max(ℓ − 1, 0);
                                /* sh2 = ℓ − sh1 */
For q = n/d, all uword:
        uword t1 = MULUH(m', n);
        q = SRL(t1 + SRL(n − t1, sh1), sh2);
```

Figure 4.1: Unsigned division by run–time invariant divisor

Explanation of Figure 4.1. If $d = 1$, then $\ell = 0$, so $m' = 1$ and $sh_1 = sh_2 = 0$. The code computes $t_1 = \lfloor 1 * n/2^N \rfloor = 0$ and $q = n$.

If $d > 1$, then $\ell \geq 1$, so $sh_1 = 1$ and $sh_2 = \ell - 1$. Since $m' \leq \frac{2^N * (2^\ell - d)}{d} + 1 \leq \frac{2^N * (d-1)}{d} + 1 < 2^N$, the value of m' fits in a **uword**. Since $0 \leq t_1 \leq n$, the formula for q simplifies to

$$\begin{aligned} q &= \mathrm{SRL}(t_1 + \mathrm{SRL}(n - t_1,\ 1),\ \ell - 1) \\ &= \left\lfloor \frac{t_1 + \lfloor (n - t_1)/2 \rfloor}{2^{\ell-1}} \right\rfloor \\ &= \left\lfloor \frac{\lfloor (t_1 + n)/2 \rfloor}{2^{\ell-1}} \right\rfloor = \left\lfloor \frac{t_1 + n}{2^\ell} \right\rfloor. \end{aligned} \tag{4.5}$$

But $t_1 + n = \lfloor m' * n/2^N \rfloor + n = \lfloor (m' + 2^N) * n/2^N \rfloor$. Set $m = m' + 2^N = \lfloor 2^{N+\ell}/d \rfloor + 1$. The hypothesis of Theorem 4.2 is satisfied since $2^{N+\ell} < m * d \leq 2^{N+\ell} + d \leq 2^{N+\ell} + 2^\ell$. ■

Caution. Conceptually q is $\mathrm{SRL}(n + t_1,\ \ell)$, as in (4.5). Do not compute q this way, since $n + t_1$ may overflow N bits and the shift count may be out of bounds.

Improvement. If d is constant and a power of 2, replace the division by a shift.

Improvement. If d is constant and $m = m' + 2^N$ is even, then reduce $m/2^\ell$ to lowest terms. The reduced multiplier fits in N bits, unlike the original. In rare cases (e.g., $d = 641$ on a 32–bit machine, $d = 274177$ on a 64–bit machine) the final shift is zero.

Improvement. If d is constant and even, rewrite $\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{\lfloor n/2^e \rfloor}{d/2^e} \right\rfloor$ for some $e > 0$. Then $\lfloor n/2^e \rfloor$ can be computed using SRL. Since $n/2^e < 2^{N-e}$, less precision is needed in the multiplier than before.

These ideas are reflected in Figure 4.2, which generates code for n/d where n is unsigned and d is constant. Procedure CHOOSE_MULTIPLIER, which is shared by this and later algorithms, appears in Figure 6.2.

```
Inputs: uword d and n, with d constant.
uword d_odd, t1;
udword m;
int e, ℓ, ℓ_dummy, sh_post, sh_pre;
(m, sh_post, ℓ) = CHOOSE_MULTIPLIER(d, N);
if m ≥ 2^N and d is even then
   Find e such that d = 2^e * d_odd and d_odd is odd.
                              /* 2^e = AND(d, 2^N − d) */
   sh_pre = e;
   (m, sh_post, ℓ_dummy)
         = CHOOSE_MULTIPLIER(d_odd, N − e);
else
   sh_pre = 0;
end if
if d = 2^ℓ then
   Issue q = SRL(n, ℓ);
else if m ≥ 2^N then
   assert sh_pre = 0;
   Issue t1 = MULUH(m − 2^N, n);
   Issue q = SRL(t1 + SRL(n − t1, 1), sh_post − 1);
else
   Issue q = SRL(MULUH(m, SRL(n, sh_pre)),
                 sh_post);
end if
```

Figure 4.2: Optimized code generation of unsigned $q = \lfloor n/d \rfloor$ for constant nonzero d

The following three examples illustrate the cases in Figure 4.2. All assume unsigned 32–bit arithmetic.

Example. $q = \lfloor n/10 \rfloor$. CHOOSE_MULTIPLIER finds $m_{\mathrm{low}} = (2^{36} - 6)/10$ and $m_{\mathrm{high}} = (2^{36} + 14)/10$. After one round of divisions by 2, it returns $(m,\ 3,\ 4)$, where $m = (2^{34} + 1)/5$. The suggested code $q = \mathrm{SRL}(\mathrm{MULUH}((2^{34} + 1)/5,\ n),\ 3)$ eliminates the pre–shift by 0. See Table 11.1.

Example. $q = \lfloor n/7 \rfloor$. Here $m = (2^{35} + 3)/7 > 2^{32}$. This example uses the longer sequence in Figure 4.1.

Example. $q = \lfloor n/14 \rfloor$. CHOOSE_MULTIPLIER first returns the same multiplier as when $d = 7$. The

suggested code uses separate divisions by 2 and 7:

$$q = \mathrm{SRL}(\mathrm{MULUH}((2^{34}+5)/7,\ \mathrm{SRL}(n,\ 1)),\ 2).$$

5 Signed division, quotient rounded towards 0

Suppose we want to compile a signed division $q = \mathrm{TRUNC}(n/d)$, where d is constant or run–time invariant, $0 < |d| \le 2^{N-1}$, and where $-2^{N-1} \le n \le 2^{N-1}-1$ is variable. All quotients are to be rounded towards zero. We could prove a theorem like Theorem 4.2 about when $\mathrm{TRUNC}(n/d) = \mathrm{TRUNC}(m * n/2^{N+\ell})$ for all n in a suitable range (cf. (7.1)), but it wouldn't help since we can't compute the right side given only $\lfloor m*n/2^N \rfloor$. Instead we show how to adjust the estimated quotient when the dividend or divisor is negative.

Theorem 5.1 *Suppose m, d, ℓ are integers such that $d \ne 0$ and $0 < m * |d| - 2^{N+\ell-1} \le 2^\ell$. Let n be an arbitrary integer such that $-2^{N-1} \le n \le 2^{N-1}-1$. Define $q_0 = \lfloor m * n/2^{N+\ell-1} \rfloor$. Then*

$$\mathrm{TRUNC}\left(\frac{n}{d}\right) = \begin{cases} q_0 \text{ if } n \ge 0 \text{ and } d > 0, \\ 1 + q_0 \text{ if } n < 0 \text{ and } d > 0, \\ -q_0 \text{ if } n \ge 0 \text{ and } d < 0, \\ -1 - q_0 \text{ if } n < 0 \text{ and } d < 0. \end{cases}$$

Proof. When $n \ge 0$ and $d > 0$, this is Theorem 4.2 with N replaced by $N-1$.

Suppose $n < 0$ and $d > 0$, say $n = q * d - r$ where $0 \le r \le d-1$. Define $k = m * d - 2^{N+\ell-1}$. Then

$$q - \frac{m*n}{2^{N+\ell-1}} = \frac{k}{2^\ell} * \frac{-n}{2^{N-1}} * \frac{1}{d} + \frac{r}{d}, \qquad (5.2)$$

as in (4.4). Since $0 < k \le 2^\ell$ by hypothesis, the first fraction on the right of (5.2) is positive and r/d is nonnegative. The sum is at most $1/d + (d-1)/d = 1$, so $q_0 = \lfloor m * n/2^{N+\ell-1} \rfloor = q - 1$, as asserted.

For $d < 0$, use $\mathrm{TRUNC}(n/d) = -\mathrm{TRUNC}(n/|d|)$. ■

Caution. When $d < 0$, avoid rewriting the quotient as $\mathrm{TRUNC}((-n)/|d|)$, which fails for $n = -2^{N-1}$.

For a run–time invariant divisor, this leads to the code in Figure 5.1. Its cost is 1 multiply, 3 adds, 2 shifts, and 1 bit–op per quotient.

Explanation of Figure 5.1. The multiplier m satisfies $2^{N-1} < m < 2^N$ except when $d = \pm 1$; in the latter cases $m = 2^N + 1$. In either case $m' = m - 2^N$ fits in an **sword**. We compute $\lfloor m * n/2^N \rfloor$ as $n + \lfloor (m-2^N) * n/2^N \rfloor$, using MULSH. The subtraction of XSIGN(n) adds one if $n < 0$. The last line negates the tentative quotient if $d < 0$ (i.e., if $d_{\mathrm{sign}} = -1$). ■

Variation. An alternate computation of m' is $m' = -\mathrm{TRUNC}\left(\dfrac{2^N * (2^{\ell-1} - |d|) + 1}{-|d|}\right)$. This uses signed $(2N)$–bit/N–bit division, with N–bit quotient.

```
Initialization (given constant sword d with d ≠ 0):
        int ℓ = max(⌈log2 |d|⌉, 1);
        udword m = 1 + ⌊2^(N+ℓ−1)/|d|⌋;
        sword m' = m − 2^N;
        sword d_sign = XSIGN(d);
        int sh_post = ℓ − 1;
For q = TRUNC(n/d), all sword:
        sword q0 = n + MULSH(m', n);
        q0 = SRA(q0, sh_post) − XSIGN(n);
        q = EOR(q0, d_sign) − d_sign;
```

Figure 5.1: Signed division by run–time invariant divisor, rounded towards zero

Overflow detection. The quotient n/d overflows if $n = -2^{N-1}$ and $d = -1$. The algorithm in Figure 5.1 returns -2^{N-1}. If overflow detection is required, the final subtraction of d_{sign} should check for overflow.

Improvement. If m is constant and even, then reduce $m/2^\ell$ to lowest terms, as in the unsigned case.

This improvement is reflected in Figure 5.2, which generates code for $\mathrm{TRUNC}(n/d)$ where d is a nonzero constant. Figure 5.2 also checks for divisor being a power of 2 or negative thereof.

```
Inputs: sword d and n, with d constant and d ≠ 0.
udword m;
int ℓ, sh_post;
(m, sh_post, ℓ) = CHOOSE_MULTIPLIER(|d|, N − 1);
if |d| = 1 then
   Issue q = d;
else if |d| = 2^ℓ then
   Issue q = SRA(n + SRL(SRA(n, ℓ − 1), N − ℓ), ℓ);
else if m < 2^(N−1) then
   Issue q = SRA(MULSH(m, n), sh_post) − XSIGN(n);
else
   Issue q = SRA(n + MULSH(m − 2^N, n), sh_post)
             − XSIGN(n);
   Cmt. Caution — m − 2^N is negative.
end if

if d < 0 then
   Issue q = −q;
end if
```

Figure 5.2: Optimized code generation of signed $q = \mathrm{TRUNC}(n/d)$ for constant $d \ne 0$

Example. $q = \mathrm{TRUNC}(n/3)$. On a 32–bit machine. CHOOSE_MULTIPLIER(3, 31) returns $sh_{\mathrm{post}} = 0$ and $m = (2^{32}+2)/3$. The code $q = \mathrm{MULSH}(m, n) - \mathrm{XSIGN}(n)$ uses one multiply, one shift, one subtract.

6 Signed division, quotient rounded towards $-\infty$

Some languages require negative quotients to round towards $-\infty$ rather than zero. With some ingenuity, we can compute these quotients in terms of quotients which round towards zero, even if the signs of the dividend and divisor are unknown at compile time.

If n and d are integers, then the identities

$$\left\lfloor \frac{n}{d} \right\rfloor = \begin{cases} \text{TRUNC}(n/d) & \text{if } n \geq 0 \text{ and } d > 0, \\ \text{TRUNC}((n+1)/d) - 1 & \text{if } n < 0 \text{ and } d > 0, \\ \text{TRUNC}((n-1)/d) - 1 & \text{if } n > 0 \text{ and } d < 0, \\ \text{TRUNC}(n/d) & \text{if } n \leq 0 \text{ and } d < 0 \end{cases}$$

are easily verified. Since the new numerators $n \pm 1$ never overflow, these identities can be used for computation. They are summarized by

$$\left\lfloor \frac{n}{d} \right\rfloor = \text{TRUNC}\left(\frac{n + d_{\text{sign}} - n_{\text{sign}}}{d} \right) + q_{\text{sign}}, \quad (6.1)$$

where $d_{\text{sign}} = \text{XSIGN}(d)$, $n_{\text{sign}} = \text{XSIGN}(\text{OR}(n,\, n + d_{\text{sign}}))$, and $q_{\text{sign}} = \text{EOR}(n_{\text{sign}},\, d_{\text{sign}})$. The cost is 2 shifts, 3 adds/subtracts, and 2 bit–ops, plus the divide ($n + d_{\text{sign}}$ is a repeated subexpression).

For remainders, a corollary to (2.1) and (6.1) is

$$\begin{aligned} n \textbf{ mod } d &= n - d * \text{TRUNC}((n + d_{\text{sign}} - n_{\text{sign}})/d) \\ &\qquad - d * q_{\text{sign}} \\ &= ((n + d_{\text{sign}} - n_{\text{sign}}) \textbf{ rem } d) \\ &\qquad - d_{\text{sign}} + n_{\text{sign}} - d * q_{\text{sign}} \\ &= ((n + d_{\text{sign}} - n_{\text{sign}}) \textbf{ rem } d) \\ &\qquad + \text{AND}(d - 2 * d_{\text{sign}} - 1,\ q_{\text{sign}}). \end{aligned} \quad (6.2)$$

The last equality in (6.2) can be verified by separately checking the cases $q_{\text{sign}} = n_{\text{sign}} - d_{\text{sign}} = 0$ and $q_{\text{sign}} = n_{\text{sign}} + d_{\text{sign}} = -1$. The subexpression $d - 2 * d_{\text{sign}} - 1$ depends only on d.

For rounding towards $+\infty$, an analog of (6.1) is

$$\left\lceil \frac{n}{d} \right\rceil = \text{TRUNC}\left(\frac{n - d_{\text{sign}} + n_{\text{pos}}}{d} \right) - \text{EOR}(d_{\text{sign}},\, n_{\text{pos}}),$$

where $d_{\text{sign}} = \text{XSIGN}(d)$ and $n_{\text{pos}} = -(n > d_{\text{sign}})$.

Improvement. If $d > 0$ is constant, then $d_{\text{sign}} = 0$. Then (6.1) becomes

$$\left\lfloor \frac{n}{d} \right\rfloor = \text{TRUNC}\left(\frac{n - n_{\text{sign}}}{d} \right) + n_{\text{sign}},$$

where $n_{\text{sign}} = \text{XSIGN}(n)$. Since $\text{TRUNC}(-x) = -\text{TRUNC}(x)$ and $\text{EOR}(-1,\, n) = -1 - n = -(n+1)$, this is equivalent to

$$\left\lfloor \frac{n}{d} \right\rfloor = \text{EOR}\left(n_{\text{sign}},\, \text{TRUNC}\left(\frac{\text{EOR}(n_{\text{sign}},\, n)}{d} \right) \right) \quad (6.3)$$
$$(d > 0).$$

The dividend and divisor on the right of (6.3) are both nonnegative and below 2^{N-1}. One can view them as signed or as unsigned when applying earlier algorithms.

Improvement. The $\text{XSIGN}(\text{OR}(n,\, n + d_{\text{sign}}))$ is equivalent to $-(n \leq \text{NOT}(d_{\text{sign}}))$ and to $-(n < -d_{\text{sign}})$, where the relationals produce 1 if true and 0 if false. On the MIPS R2000/R3000 [12], for example, one can compute

$$\begin{aligned} &-d_{\text{sign}} = \text{SRL}(d,\, N-1); \\ &-n_{\text{sign}} = (n < -d_{\text{sign}}); \qquad \text{/* SLT, signed */} \\ &-q_{\text{sign}} = \text{EOR}(-n_{\text{sign}},\, -d_{\text{sign}}); \\ &q = \text{TRUNC}((n - (-d_{\text{sign}}) + (-n_{\text{sign}}))/d) - (-q_{\text{sign}}); \end{aligned}$$

(six instructions plus the divide), saving an instruction over (6.1).

Improvement. If n known to be nonzero, then n_{sign} simplifies to $\text{XSIGN}(n)$.

For constant divisors, one can use (6.1) and the algorithm in Figure 5.2. For constant $d > 0$ a shorter algorithm, based on (6.3), appears in Figure 6.1.

Inputs: **sword** n and d, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m,\, sh_{\text{post}},\, \ell) = \text{CHOOSE\_MULTIPLIER}(d,\, N-1)$;
if $d = 2^{\ell}$ **then**
  Issue $q = \text{SRA}(n,\, \ell)$;
else
  assert $m < 2^N$;
  Issue **sword** $n_{\text{sign}} = \text{XSIGN}(n)$;
  Issue **uword** $q_0 = \text{MULUH}(m,\, \text{EOR}(n_{\text{sign}},\, n))$;
  Issue $q = \text{EOR}(n_{\text{sign}},\ \text{SRL}(q_0,\, sh_{\text{post}}))$;
end if

Figure 6.1: Optimized code generation of signed $q = \lfloor n/d \rfloor$ for constant $d > 0$

Example. Using signed 32–bit arithmetic, the code for $r = n$ **mod** 10 (nonnegative remainder) can be

$$\begin{aligned} &\textbf{sword } n_{\text{sign}} = \text{XSIGN}(n); \\ &\textbf{uword } q_0 = \text{MULUH}((2^{33}+3)/5,\, \text{EOR}(n_{\text{sign}},\, n)); \\ &\textbf{sword } q = \text{EOR}(n_{\text{sign}},\, \text{SRL}(q_0,\, 2)); \\ &r = n - \text{SLL}(q,\, 1) - \text{SLL}(q,\, 3);. \end{aligned}$$

The cost is 1 multiply, 4 shifts, 2 bit–ops, 2 subtracts.

Alternately, if one has a fast signed division algorithm which rounds quotients towards 0 and returns remainders, then (6.2) justifies the code

$$r = ((n - \text{XSIGN}(n)) \textbf{ rem } 10) + \text{AND}(9,\, \text{XSIGN}(n)).$$

The cost is 1 divide, 1 shift, 1 bit–op, 2 adds/subtracts.

procedure CHOOSE_MULTIPLIER(**uword** d, **int** $prec$);
Cmt. d – Constant divisor to invert. $1 \le d < 2^N$.
Cmt. $prec$ – Number of bits of precision needed, $1 \le prec \le N$.
Cmt. Finds m, $sh_{\rm post}$, ℓ such that:
Cmt. $\quad 2^{\ell-1} < d \le 2^\ell$.
Cmt. $\quad 0 \le sh_{\rm post} \le \ell$. If $sh_{\rm post} > 0$, then $N + sh_{\rm post} \le \ell + prec$.
Cmt. $\quad 2^{N+sh_{\rm post}} < m * d \le 2^{N+sh_{\rm post}} * (1 + 2^{-prec})$.
Cmt. Corollary. If $d \le 2^{prec}$, then $m < 2^{N+sh_{\rm post}} * (1 + 2^{1-\ell})/d \le 2^{N+sh_{\rm post}-\ell+1}$.
Cmt. $\quad$ Hence m fits in $\max(prec, N-\ell)+1$ bits (unsigned).
Cmt.
int $\ell = \lceil \log_2 d \rceil$, $\quad sh_{\rm post} = \ell$;
udword $m_{\rm low} = \lfloor 2^{N+\ell}/d \rfloor$, $\quad m_{\rm high} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
Cmt. To avoid numerator overflow, compute $m_{\rm low}$ as $2^N + (m_{\rm low} - 2^N)$.
Cmt. Likewise for $m_{\rm high}$. Compare m' in Figure 4.1.
Invariant. $m_{\rm low} = \lfloor 2^{N+sh_{\rm post}}/d \rfloor < m_{\rm high} = \lfloor 2^{N+sh_{\rm post}} * (1 + 2^{-prec})/d \rfloor$.
while $\lfloor m_{\rm low}/2 \rfloor < \lfloor m_{\rm high}/2 \rfloor$ **and** $sh_{\rm post} > 0$ **do**
$\quad m_{\rm low} = \lfloor m_{\rm low}/2 \rfloor$; $\quad m_{\rm high} = \lfloor m_{\rm high}/2 \rfloor$; $\quad sh_{\rm post} = sh_{\rm post} - 1$;
end while; /* Reduce to lowest terms. */
return ($m_{\rm high}$, $sh_{\rm post}$, ℓ); /* Three outputs. */
end CHOOSE_MULTIPLIER;

Figure 6.2: Selection of multiplier and shift count

7 Use of floating point

One alternative to MULUH and MULSH uses floating point arithmetic. Let the floating point mantissa be F bits wide (e.g., $F = 53$ for IEEE double precision arithmetic). Then any floating point operation has relative error at most 2^{1-F}, regardless of the rounding mode, unless exponent overflow or underflow occurs.

Suppose $N \ge 1$ and $F \ge N + 3$. We claim that

$$\begin{aligned} &\text{TRUNC}\left(\frac{n}{d}\right) = \text{TRUNC}(q_{\rm est}), \\ \text{where}\quad &q_{\rm est} \approx n * \left(\frac{1 + 2^{2-F}}{d}\right), \end{aligned} \tag{7.1}$$

whenever $|n| \le 2^N - 1$ and $0 < |d| < 2^N$, regardless of the rounding modes used to compute $q_{\rm est}$. The proof assumes that $n > 0$ and $d > 0$, by negating both sides of (7.1) if necessary (the case $n = 0$ is trivial).

Since the relative error per operation is at most 2^{1-F}, the estimated quotient $q_{\rm est}$ satisfies

$$\frac{1 + 2^{2-F}}{(1 + 2^{1-F})^2} * \frac{n}{d} \le q_{\rm est} \le \left(1 + 2^{2-F}\right) * \left(1 + 2^{1-F}\right)^2 * \frac{n}{d}.$$

Use this and the inequalities

$$\begin{aligned} 1 - 2^{-F} \le 1 - 2^{2-2F} &< \frac{1 + 2^{2-F}}{(1 + 2^{1-F})^2}, \\ (1 + 2^{2-F}) * (1 + 2^{1-F})^2 &< \frac{1}{1 - 2^{3-F}} \le \frac{1}{1 - 2^{-N}} \end{aligned}$$

to derive

$$(1 - 2^{-F}) * \frac{n}{d} < q_{\rm est} < \frac{n/d}{1 - 2^{-N}} \le \frac{n/d}{1 - \frac{1}{n+1}} = \frac{n+1}{d}.$$

Denote $q = \text{TRUNC}(n/d)$. Then $q_{\rm est} < (n+1)/d$ implies $\text{TRUNC}(q_{\rm est}) \le q$. If $q_{\rm est} < q$, then

$$(1 - 2^{-F}) * q \le (1 - 2^{-F}) * \frac{n}{d} < q_{\rm est} < q.$$

Both q and $q_{\rm est}$ are exactly representable as floating point numbers, but there are no representable numbers strictly between $(1 - 2^{-F}) * q$ and q. This contradiction shows that $q_{\rm est} \ge q$ and hence $q = \text{TRUNC}(q_{\rm est})$. ■

For quotients rounded towards $-\infty$, use (6.1). If $F = 53$ and $N \le 50$, then (7.1) can be used for N–bit integer division. The algorithm may trigger an IEEE exception for inexactness if the application program enables that condition.

Alverson [1] uses integer multiplication, but computes the multiplier using floating point arithmetic.

Baker [3] does modular multiplication using a combination of floating point and integer arithmetic.

8 Dividing udword by uword

One primitive operation for multiple–precision arithmetic [14, p. 251] is the division of a **udword** by a **uword**, obtaining **uword** quotient and remainder, where the quotient is known to be less than 2^N. We

Initialization (given **uword** d, where $0 < d < 2^N$):

int $\ell = 1 + \lfloor \log_2 d \rfloor$;	/* $2^{\ell-1} \le d < 2^\ell$ */
uword $m' = \lfloor (2^N * (2^\ell - d) - 1)/d \rfloor$;	/* $m' = \lfloor (2^{N+\ell} - 1)/d \rfloor - 2^N$ */
uword $d_{\rm norm} = {\rm SLL}(d,\ N - \ell)$;	/* Normalized divisor $d * 2^{N-\ell}$ */

For $q = \lfloor n/d \rfloor$ and $r = n - q * d$,
where d, q, r are **uword** and n is **udword**:

uword $n_2 = {\rm SLL}({\rm HIGH}(n),\ N - \ell) + {\rm SRL}({\rm LOW}(n),\ \ell)$;	/* See note about shift count. */
uword $n_{10} = {\rm SLL}({\rm LOW}(n),\ N - \ell)$;	/* $n_{10} = n_1 * 2^{N-1} + n_0 * 2^{N-\ell}$ */
	/* Ignore overflow. */
sword $-n_1 = {\rm XSIGN}(n_{10})$;	
uword $n_{\rm adj} = n_{10} + {\rm AND}(-n_1,\ d_{\rm norm} - 2^N)$;	/* $n_{10} + n_1 * (d_{\rm norm} - 2^N)$ */
	/* $= n_1 * (d_{\rm norm} - 2^{N-1}) + n_0 * 2^{N-\ell}$ */
	/* Underflow is impossible. */
uword $q_1 = n_2 + {\rm HIGH}\big(m' * (n_2 - (-n_1)) + n_{\rm adj}\big)$;	/* See Lemma 8.1. */
sdword $dr = n - 2^N * d + (2^N - 1 - q_1) * d$;	/* $dr = n - q_1 * d - d, \quad -d \le dr < d$ */
$q = {\rm HIGH}(dr) - (2^N - 1 - q_1) + 2^N$;	/* Add 1 to quotient if $dr \ge 0$. */
$r = {\rm LOW}(dr) + {\rm AND}(d - 2^N,\ {\rm HIGH}(dr))$;	/* Add d to remainder if $dr < 0$. */

Figure 8.1: Unsigned division of **udword** by run–time invariant **uword**.

describe a way to compute this quotient and remainder after some preliminary computations involving only the divisor, when the divisor is a run–time invariant expression.

Lemma 8.1 *Suppose that d, m, and ℓ are nonnegative integers such that $2^{\ell-1} \le d < 2^\ell \le 2^N$ and*

$$0 < 2^{N+\ell} - m * d \le d. \tag{8.2}$$

Given n with $0 \le n \le d * 2^N - 1$, write $n = n_2 * 2^\ell + n_1 * 2^{\ell-1} + n_0$, where n_0, n_1, and n_2 are integers with $0 \le n_1 \le 1$ and $0 \le n_0 \le 2^{\ell-1} - 1$. Define integers q_1 and q_0 by

$$\begin{aligned} q_1 * 2^N + q_0 = {} & n_2 * 2^N + (n_2 + n_1) * (m - 2^N) \\ & + n_1 * \big(d * 2^{N-\ell} - 2^{N-1}\big) \\ & + n_0 * 2^{N-\ell} \end{aligned} \tag{8.3}$$

and $0 \le q_0 \le 2^N - 1$. Then $0 \le q_1 \le 2^N - 1$ and $0 \le n - q_1 * d < 2 * d$.

Proof. Define $k = 2^{N+\ell} - m * d$. Then (8.2) implies $0 < k \le d \le 2^\ell - 1$.

The bound $n \le d * 2^N - 1$ implies $n_2 \le d * 2^{N-\ell} - 1$. Equation (8.2) implies $m > 2^{N+\ell}/d > 2^N$. A corollary to (8.3) is

$$\begin{aligned} q_1 * 2^N + q_0 &= n_2 * m + n_1 * (m - 2^N) \\ &\quad + 2^{N-\ell} * \big(n_1 * (d - 2^{\ell-1}) + n_0\big) \\ &\le (d * 2^{N-\ell} - 1) * m + 1 * (m - 2^N) \\ &\quad + 2^{N-\ell} * \big(1 * (2^{\ell-1} - 1) + (2^{\ell-1} - 1)\big) \\ &= 2^{N-\ell} * (d * m - 2) < 2^{2N}. \end{aligned}$$

This proves the upper bound on the integer q_1.

A straightforward calculation using the definitions of k and q_0 and n_0 reveals that

$$\begin{aligned} n - q_1 * d = {} & \frac{(n_2 + n_1) * k + q_0 * d}{2^N} \\ & + \left(1 - \frac{d}{2^\ell}\right) * \Big(n_1 * (d - 2^{\ell-1}) + n_0\Big). \end{aligned} \tag{8.4}$$

Since $2^{\ell-1} \le d < 2^\ell$ by hypothesis, the right side of (8.4) is nonnegative. This remainder is bounded by

$$\begin{aligned} & \frac{(d * 2^{N-\ell}) * d + (2^N - 1) * d}{2^N} \\ & \quad + \left(1 - \frac{d}{2^\ell}\right) * \Big(1 * (d - 2^{\ell-1}) + (2^{\ell-1} - 1)\Big) \\ & < \left(\frac{d^2}{2^\ell} + d\right) + \left(1 - \frac{d}{2^\ell}\right) * d = 2 * d, \end{aligned}$$

completing the proof. ■

This leads to an algorithm like that in Figure 8.1 when dividing a **udword** by a run–time invariant **uword** with quotient known to be less than 2^N. Unlike the previous algorithms, this code rounds the multiplier down when computing a reciprocal. After initializations depending only on the divisor d, this algorithm requires two products (both halves of each) and 20–25 simple operations (including doubleword adds and subtracts). Five registers hold d, $d_{\rm norm}$, ℓ, m', and $N - \ell$.

Note. The shift count ℓ in the computations of m' and n_2 may equal N. If this is too large, use separate shifts by $\ell - 1$ and 1. If a doubleword shift is available, compute n_2 and n_{10} together.

9 Exact division by constants

Occasionally a language construct requires a division whose remainder is known to vanish. An example occurs in C when subtracting two pointers. Their numerical difference is divided by the object size. The object size is a compile–time constant.

Suppose we want code for $q = n/d$, where d is a nonzero constant and n is an expression known to be divisible by d. Write $d = 2^e * d_{\text{odd}}$ where d_{odd} is odd. Find d_{inv} such that $1 \le d_{\text{inv}} \le 2^N - 1$ and

$$d_{\text{inv}} * d_{\text{odd}} \equiv 1 \pmod{2^N}. \tag{9.1}$$

Then

$$\begin{aligned} 2^e * q = 2^e * \frac{n}{d} &= \frac{n}{d_{\text{odd}}} \\ &\equiv (d_{\text{inv}} * d_{\text{odd}}) * \frac{n}{d_{\text{odd}}} = d_{\text{inv}} * n \pmod{2^N}, \end{aligned}$$

as in [2]. Hence $2^e * q \equiv d_{\text{inv}} * n \pmod{2^N}$. Since $n/d_{\text{odd}} = 2^e * q$ fits in N bits, it must equal the lower half of the product $d_{\text{inv}} * n$, namely $\text{MULL}(d_{\text{inv}}, n)$. An SRA (for signed division) or SRL (for unsigned division) produces the quotient q.

The multiplicative inverse d_{inv} of d_{odd} modulo 2^N can be found by the extended Euclidean GCD algorithm [14, p. 325]. Another algorithm observes that (9.1) holds modulo 2^3 if $d_{\text{inv}} = d_{\text{odd}}$. Each Newton iteration

$$d_{\text{inv}} \leftarrow d_{\text{inv}} * (2 - d_{\text{inv}} * d_{\text{odd}}) \textbf{ mod } 2^N \tag{9.2}$$

doubles the known exponent by which (9.1) holds, so $\lceil \log_2(N/3) \rceil$ iterations of (9.2) suffice.

If $d_{\text{odd}} = \pm 1$, then $d_{\text{inv}} = d_{\text{odd}}$ so the multiplication by d_{inv} is trivial or a negation. If d is odd, then $e = 0$ and the shift disappears.

A variation tests whether an integer n is exactly divisible by a nonzero constant d without computing the remainder. If d is a power of 2 (or the negative thereof, in the signed case), then check the lower bits of n to test whether d divides n. Otherwise compute d_{inv} and e as above. Let $q_0 = \text{MULL}(d_{\text{inv}}, n)$. If $n = q * d$ for some q, then $q_0 = 2^e * q$ must be a multiple of 2^e. The original division is exact (no remainder) precisely when

(i) q_0 is a multiple of 2^e, and

(ii) q_0 is sufficiently small that $q_0 * d_{\text{odd}}$ is representable by the original data type.

For unsigned division check that

$$0 \le q_0 \le 2^e * \left\lfloor \frac{2^N - 1}{d} \right\rfloor$$

and that the bottom e bits of q_0 (or of n) are zero. When $e > 0$, these tests can be combined if the architecture has a rotate (i.e., circular shift) instruction, or by expanding this rotate into

$$\text{OR}(\text{SRL}(q_0,\, e),\ \text{SLL}(q_0,\, N - e)) \le \left\lfloor \frac{2^N - 1}{d} \right\rfloor.$$

For signed division check that

$$-2^e * \left\lfloor \frac{2^{N-1}}{d} \right\rfloor \le q_0 \le 2^e * \left\lfloor \frac{2^{N-1} - 1}{d} \right\rfloor$$

and that the bottom e bits of q_0 are zero; the interval check can be done with an add and one signed or unsigned compare. Relatedly, to test whether n **rem** $d = r$, where d and r are constants with $1 \le r < d$ and where n is signed, check whether $\text{MULL}(d_{\text{inv}}, n - r)$ is a nonnegative multiple of 2^e not exceeding $2^e * \lfloor (2^{N-1} - 1 - r)/d \rfloor$.

EXAMPLE. To test whether a signed 32–bit value i is divisible by 100, let $d_{\text{inv}} = (19 * 2^{32} + 1)/25$. Compute **sword** $q_0 = \text{MULL}(d_{\text{inv}}, i)$. Next check whether q_0 is a multiple of 4 in the interval $[-q_{\max}, q_{\max}]$, where $q_{\max} = (2^{31} - 48)/25$.

Since these algorithms require only the lower half of a product, other optimizations for integer multiplication apply here too. For example, applying strength reduction to the C loop

```
signed long i, imax;
for (i = 0; i < imax; i++) {
   if ((i % 100) == 0) {
      ...
   }
}
```

might yield (** denotes exponentiation)

```
const unsigned long dinv = (19*2**32 + 1)/25;
const unsigned long qmax = (2**31 - 48)/25;
unsigned long test = qmax;
       /* test = dinv*i + qmax   mod 2**32 */
for (i = 0;  i < imax;  i++,  test += dinv) {
   if (test <= 2*qmax  &&  (test & 3) == 0) {
      ...
   }
}
```

No explicit multiplication or division remains.

10 Implementation in GCC

We have implemented the algorithms for constant divisors in the freely available GCC compiler [21], by extending its machine– and language–independent internal code generation. We also made minor machine–dependent modifications to some of the *machine descriptor*, or *md* files to get optimal code. All languages and almost all processors supported by GCC benefit. Our changes are scheduled for inclusion in GCC 2.6.

To generate code for division of N–bit quantities, the CHOOSE_MULTIPLIER function needs to perform $(2N)$–bit arithmetic. This makes that procedure more complex than it might appear in Figure 6.2.

Optimal selection of instructions depending on the bitsize of the operation is a tricky problem that we spent quite some time on. For some architectures, it is important to select a multiplication instruction that has the smallest available precision. On other architectures, the multiplication can be performed faster using a sequence of additions, subtractions, and shifts.

We have not implemented any algorithm for run–time invariant divisors. Only a few architectures (AMD 29050, Intel x86, Motorola 68k & 88110, and to some extent IBM POWER) have adequate hardware support to make such an implementation viable, i.e., an instruction that can be used for integer logarithm computation, and a $(2N)$–bit/N–bit divide instruction. Even with hardware support, one must be careful that the transformation really improves the code; e.g., a loop might need to be executed many times before the faster loop body outweighs the cost of the multiplier computation in the loop header.

11 Results

Figure 11.1 has an example with compile–time constant divisor that gets drastically faster on all recent processor implementations. The program converts a binary number to a decimal string. It calculates one quotient and one remainder per output digit.

Table 11.1 shows the generated assembler codes for Alpha, MIPS, POWER, and SPARC. There is no explicit division. Although initially computed separately, the quotient and remainder calculations have been combined (by GCC's common subexpression elimination pass).

The `unsigned int` data type has 32 bits on all four architectures, but Alpha is a 64–bit architecture. The Alpha code is longer than the others because it multiplies $(2^{34}+1)/5$ by x using

$$4*\left[(2^{16}+1)*(2^8+1)*\left(4*[4*(4*x-x)+x]-x\right)\right]+x$$

instead of the slower, 23–cycle, `mulq`. This illustrates that the multiplications needed by these algorithms can sometimes be computed quickly using a sequence of shifts, adds, and subtracts [5], since multipliers for small constant divisors have regular binary patterns.

Table 11.2 compares the timing on some processor implementations for the radix conversion routine, with and without the division elimination algorithms. The number converted was a full 32–bit number, sufficiently large to hide procedure calling overhead from the measurements.

We also ran the integer benchmarks from SPEC'92. The improvement was negligible for most of the programs; the best improvement seen was only about 3%. Some benchmarks that involve hashing show improvements up to about 30%. We anticipate significant improvements on some number theoretic codes.

References

[1] Robert Alverson. Integer division using reciprocals. In Peter Kornerup and David W. Matula, editors, *Proceedings 10th Symposium on Computer Arithmetic*, pages 186–190, Grenoble, France, June 1991.

[2] Ehud Artzy, James A. Hinds, and Harry J. Saal. A fast division technique for constant divisors. *CACM*, 19(2):98–101, February 1976.

[3] Henry G. Baker. Computing A*B (mod N) efficiently in ANSI C. *ACM SIGPLAN Notices*, 27(1):95–98, January 1992.

[4] H.B. Bakoglu, G.F. Grohoski, and R. K. Montoye. The IBM RISC system/6000 processor: Hardware overview. *IBM Journal of Research and Development*, 34(1):12–22, January 1990.

[5] Robert Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[6] Raymond T. Boute. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992.

[7] A.P. Chang. A note on the modulo operation. *SIGPLAN Notices*, 20(4):19–23, April 1985.

[8] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor, Hardware Reference Manual*, 1st edition, October 1992.

[9] Intel Corporation, Santa Clara, CA. *386 DX Microprocessor Programmer's Reference Manual*, 1990.

[10] Intel Corporation, Santa Clara, CA. *Intel486 Microprocessor Family Programmer's Reference Manual*, 1992.

[11] David H. Jacobsohn. A combinatoric division algorithm for fixed-integer divisors. *IEEE Trans. Comp.*, C–22(6):608–610, June 1973.

[12] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1989.

```
#define BUFSIZE 50
char *decimal (unsigned int x)
{
    static char buf[BUFSIZE];
    char *bp = buf + BUFSIZE - 1;
    *bp = 0;
    do {
        *--bp = '0' + x % 10;
        x /= 10;
    } while (x != 0);
    return bp;   /* Return pointer to first digit */
}
```

Figure 11.1: Radix conversion code

Alpha		MIPS		POWER		SPARC	
	lda $2,buf		la $5,buf+49		l 10,LC..0(2)		sethi %hi(buf+49),%g2
	ldq_u $1,49($2)		sb $0,0($5)		cau 11,0,0xcccc		or %g2,%lo(buf+49),%o1
	addq $2,49,$0		li $6,0xcccc0000		oril 11,11,0xcccd		stb %g0,[%o1]
	mskbl $1,$0,$1		ori $6,$6,0xcccd		cal 0,0(0)		sethi %hi(0xcccccccd),%g2
	stq_u $1,49($2)	L1:	multu $4,$6		stb 0,0(10)		or %g2,0xcd,%o2
L1:	zapnot $16,15,$3		mfhi $3	L1:	mul 9,3,11	L1:	add %o1,-1,%o1
	s4subq $3,$3,$2		subu $5,$5,1		srai 0,3,31		umul %o0,%o2,%g0
	s4addq $2,$3,$2		srl $3,$3,3		and 0,0,11		rd %y,%g3
	s4subq $2,$3,$2		sll $2,$3,2		a 9,9,0		srl %g3,3,%g3
	sll $2,8,$1		addu $2,$2,$3		a 9,9,3		sll %g3,2,%g2
	subq $0,1,$0		sll $2,$2,1		sri 9,9,3		add %g2,%g3,%g2
	addq $2,$1,$2		subu $2,$4,$2		muli 0,9,10		sll %g2,1,%g2
	sll $2,16,$1		addu $2,$2,48		sf 0,0,3		sub %o0,%g2,%g2
	ldq_u $4,0($0)		move $4,$3		ai. 3,9,0		add %g2,48,%g2
	addq $2,$1,$2		bne $4,$0,L1		ai 0,0,48		orcc %g3,%g0,%o0
	s4addq $2,$3,$2		sb $2,0($5)		stbu 0,-1(10)		bne L1
	srl $2,35,$2		j $31		bc 4,2,L1		stb %g2,[%o1]
	mskbl $4,$0,$4		move $2,$5		ai 3,10,0		retl
	s4addl $2,$2,$1				br		mov %o1,%o0
	addq $1,$1,$1						
	subl $16,$1,$1						
	addl $1,48,$1						
	insbl $1,$0,$1						
	bis $2,$2,$16						
	bis $1,$4,$1						
	stq_u $1,0($0)						
	bne $16,L1						
	ret $31,($26),1						

Table 11.1: Code generated by our GCC for radix conversion

Architecture/Implementation	MHz	Time with division performed	Time with division eliminated	Speedup ratio
Motorola MC68020 [18, pp. 9–22]	25	39	33	1.2
Motorola MC68040	25	19	14	1.4
SPARC Viking [20]	40	6.4	3.2	2.0
HP PA 7000	99	9.7	2.1	4.6
MIPS R3000 [12]	40	12	7.3	1.7
MIPS R4000 [17]	100	8.3	2.4	3.4
POWER/RIOS I [4, 22]	50	5.0	3.5	1.4
DEC Alpha 21064 [8]	133	22	1.8	12*

*This time difference is artificial. The Alpha architecture has no integer divide instruction, and the DEC library functions for division are slow.

Table 11.2: Timing (microseconds) for radix conversion with and without division elimination

[13] Donald E. Knuth. An empirical study of FORTRAN programs. Technical Report CS–186, Computer Science Department, Stanford University, 1970. Stanford artificial intelligence project memo AIM–137.

[14] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[15] Shuo-Yen Robert Li. Fast constant division routines. *IEEE Trans. Comp.*, C–34(9):866–869, September 1985.

[16] Daniel J. Magenheimer, Liz Peters, Karl Pettis, and Dan Zuras. Integer multiplication and division on the HP Precision Architecture. In *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*. ACM, 1987. Published as *SIGPLAN Notices*, Volume 22, No. 10, October, 1987.

[17] MIPS Computer Systems, Inc, Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, 1991.

[18] Motorola, Inc. *MC68020 32–Bit Microprocessor User's Manual*, 2nd edition, 1985.

[19] Motorola, Inc. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[20] SPARC International, Inc., Menlo Park, CA. *The SPARC Architecture Manual, Version 8*, 1992.

[21] Richard M. Stallman. *Using and Porting GCC*. The Free Software Foundation, Cambridge, MA, 1993.

[22] Henry Warren. *Predicting Execution Time on the IBM RISC System/6000*. IBM, 1991. Preliminary Version.