N-Bit Unsigned Division Via N-Bit Multiply-Add

Arch D. Robison
Intel Corporation
arch.robison@intel.com

Abstract

Integer division on modern processors is expensive compared to multiplication. Previous algorithms for performing unsigned division by an invariant divisor, via reciprocal approximation, suffer in the worst case from a common requirement for n+1 bit multiplication, which typically must be synthesized from n-bit multiplication and extra arithmetic operations. This paper presents, and proves, a hybrid of previous algorithms that replaces n+1 bit multiplication with a single fused multiply-add operation on n-bit operands, thus reducing any n-bit unsigned division to the upper n bits of a multiply-add, followed by a single right shift. An additional benefit is that the prerequisite calculations are simple and fast. On the Itanium® 2 processor, the technique is advantageous for as few as two quotients that share a common run-time divisor.

1. Introduction

In typical programs, integer division is relatively infrequent compared to other arithmetic operations. Combined with the complexity of directly implementing division in hardware, this has led a trend in modern processor architectures to omit direct support for integer division, and instead rely on software implementation. A case of particular interest is where a divisor is a compile-time constant [1] [7], or a run-time loop-invariant [6]. Previous work has shown that in such situations, an unsigned integer division $\lfloor x/d \rfloor$ can be profitably computed as $\lfloor (ax+b)/2^s \rfloor$, where integer a approximates the scaled reciprocal $2^s/d$, integer b compensates for rounding errors, and integer s is a right-shift count.

Previous work varies in whether the approximation a is rounded down [7] or rounded up [1] [6] from the exact scaled reciprocal. Rounding up has a slight advantage of making b=0¸ which simplifies implementation on most processors. However, for performing n-bit unsigned division, all prior schemes based on $\lfloor (ax+b)/2^s \rfloor$ require that a be rounded to n+1 bits of significance in the general case. Processors naturally implement only n-bit arithmetic, necessitating extra operations to synthesize n+1 bit arithmetic. For some divisors, the extra bit can be optimized away because it is zero [6]. But this is neither possible for all divisors nor for divisors determined at run time.

This paper contributes a method and proof for avoiding the extra bit, so that $\lfloor x/d \rfloor$ can be evaluated using $\lfloor (ax+b)/2^s \rfloor$, where a and b have only n bits of significance and s is a non-negative integer. The key is choosing whether to round the reciprocal $1/d$ up or down. Serendipitously, the value b turns out to be 0 or a. Furthermore, the choice can be made quickly enough to pay off even when as few as two unsigned quotients share a divisor.

Some 7-bit examples demonstrate how choosing the direction of rounding matters. (For shorter precisions, the "round down" approach always works given a suitable choice of b.) Consider a divisor d=11. The reciprocal 1/11 is $0.00010111010001..._2$. Rounding down to the most significant 7 bits yields 0.0001011101_2, which is $93/2^{10}$. The approximation is slightly low, but b can be chosen to avoid undershoot [7]. One possible choice is b=93; the formula $\lfloor (93x+93)/2^{10} \rfloor$ computes $\lfloor x/11 \rfloor$ for any unsigned 7-bit x. Attempts to use a rounded-up reciprocal ($94/2^{10}$) are doomed: even for b=0, the formula $\lfloor (94x+b)/2^{10} \rfloor$ overshoots $\lfloor x/11 \rfloor$ when x=109.

For a divisor d=13, the situation is reversed. Rounding up works but rounding down does not. The reciprocal 1/13 is $0.00010011101100..._2$. Rounding up to the most significant 7 bits yields 0.0001001111_2, which is $79/2^{10}$. Indeed, $\lfloor (79x+0)/2^{10} \rfloor$ can be used to compute $\lfloor x/13 \rfloor$ for any unsigned 7-bit x. However, using the rounded down reciprocal $78/2^{10}$ is hopeless. The formula $\lfloor (78x+b)/2^{10} \rfloor$ undershoots (for x=117) if b=87 and overshoots (for x=12) if b=88, and there is no other integral value of b in between.

In the two 7-bit examples, rounding the reciprocal in one particular direction works, but the other direction introduces too large an error. Previous algorithms shrink the error by resorting to 8-bit approximations. The new algorithms shrink the error by choosing a favorable rounding direction.

The rest of this paper is structured as follows. Section 2 describes the necessary architectural support. Section 3 gives the mathematical basis. Section 4 describes algorithms for choosing a and b. Section 5 evaluates the overhead and benefit of the technique for both compile-time constant divisors and hoisting loop-invariant divisors, with specifics for the Itanium® 2 processor. Section 6 discusses related work. Section 7 summarizes findings.

2. Architectural support

The algorithms to be presented depend upon an *integer* fused multiply-add instruction, denoted XMA.HU, that delivers the high n-bits of $ax+b$. Optionally, an instruction XMA.LU that returns the low n bits of $ax+b$ will be useful. Formally, for a, x, and b that are n-bit unsigned integers, these instructions are defined as:

$$\text{XMA.HU}(a,x,b) = \lfloor (ax+b)/2^n \rfloor$$
$$\text{XMA.LU}(a,x,b) = (ax+b) \bmod 2^n$$

The mnemonics are taken from Itanium® instructions. In the Itanium® architecture, multiply-add runs in the floating-point unit, hence extra transfer instructions must be executed to get integer operands to and from the floating-point unit. To simplify the presentation, the transfer instructions are omitted from the algorithms, though they are considered in the timing comparisons of Section 5.

On a machine without multiply-add, XMA.LU is trivially performed by an n-bit multiplication and n-bit addition. XMA.HU is a bit harder, because the possible carry from the low n bits to the upper n bits must be propagated. One way is to compute $ax+b$ exactly, using $2n$ bits, and take the upper n bits. For example, on x86 processors this can be done for n=32 in four instructions:

```
mov eax,a   // Set eax:=a
mul x       // Set pair (edx,eax):=x*eax
add eax,b   // Set eax:=eax+b
adc edx,0   // Propagate carry
```

Though fused floating-point multiply-add instructions are common on RISCs, the integer variant is less common. Typically, four instructions are required: two to compute both halves of the product, followed by two to perform the double precision addition.

DSP processors typically have a signed multiply-accumulate, but perhaps not the unsigned variant. This can be worked around by adding a correction, because the algorithms to be presented always set the high-order bit of a to 1, so the algebraic value of a, interpreted as a signed value, will be $a-2^n$. Therefore $\text{XMA.HU}(a,x,b) = x+\text{XMA.HS}(a,x,b)$, where XMA.HS denotes a multiply-add instruction that treats a and x (but not b) as signed integers.

Besides integer multiply-add, the usual n-bit unsigned and signed right-shifts, denoted SHR.U and SHR.S respectively, are presumed to exist:

$$\text{SHR.U}(x,m) = \lfloor x/2^m \rfloor \quad (x \text{ an unsigned integer})$$
$$\text{SHR.S}(x,m) = \lfloor x/2^m \rfloor \quad (x \text{ a signed integer})$$

Some of the algorithms require a *floating-point* fused multiply-add operation and the ability to extract the binary exponent and significand from a floating-point value. For floating-point values x, y, and z, the fused multiply-add operations computes $xy+z$ with a single final rounding to n bits of significance, where n includes the leading 1 bit. This operation, common on high-performance floating-point units, is denoted $(x\times y+z)_{rn}$. The exponent bias is denoted BIAS. Operations to extract the exponent and significand are denoted as EXPONENT and SIGNIFICAND. I.e., for $f\neq 0$, $f = \text{SIGNIFICAND}(f)\times 2^{\text{EXPONENT}(f)-\text{BIAS}-n+1}$. The "$-n+1$" adjusts for the fact that SIGNIFICAND(f) returns an n-bit integer that represents a significand with an explicit leading one followed by n-1 bits after the binary point.

3. Mathematical foundation

This section gives the mathematical basis for the algorithms, motivated by graphical analysis. As with other algorithms that perform division via reciprocal approximation, the notion is that for a given divisor d, the integer quotient $\lfloor x/d \rfloor$ can be approximated by computing a linear function $f(x)$ and rounding it down.

We start with Lemma 1, which provides some useful bounds.

Lemma 1 Let $c = 1/d$ for a positive integer d. For any integer x, if $cx \leq f(x) < cx+c$ then $\lfloor x/d \rfloor = \lfloor f(x) \rfloor$.

Proof: Proceed by showing that $\lfloor x/d \rfloor \leq f(x) < \lfloor x/d \rfloor + 1$. The lower bound on $f(x)$ follows from the observation $\lfloor x/d \rfloor \leq x/d = cx \leq f(x)$. The upper bound on $f(x)$ depends upon the fact that x and d are integers, and so by elementary number theory, x can be written as $dk+j$, with $k=\lfloor x/d \rfloor$ and $0 \leq j \leq d-1$. Then $f(x) < cx+c = c(dk+j)+c = (cd)k+cj+c = k+cj+c = k+c(j+1) \leq k+cd = k+1 = \lfloor x/d \rfloor+1$. □

Fig. 1 depicts the situation. For an integer x, $f(x)$ must lie on or above the lower staircase $\lfloor x/d \rfloor$ and strictly below the upper stair case $\lceil (x+1)/d \rceil$. Lemma 1's bounds on $f(x)$ delimit the gray area. The closed lower bound rests on the lower staircase. The open upper bound presses up against the upper staircase. The lemma's bounds might seem peculiar, because obviously the necessary and sufficient bounds are clearly $\lfloor x/d \rfloor \leq f(x) < \lfloor x/d \rfloor+1$ for any real x. Lemma 1's bounds are nonetheless more useful in this

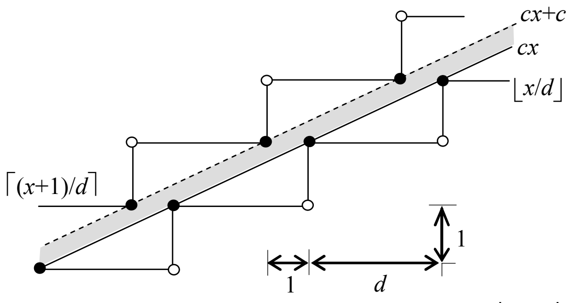

Fig. 1. Linear bounds for approximating $\lfloor x/d \rfloor$ for integral *x* and *d*.

paper because their linear form allows geometrical arguments to be employed in the proof of the key theorems below.

The next two theorems concern instances of f with the form αx and $\alpha x+\alpha$. The theorems phrase bounds on α in terms of *relative* error, instead of the absolute error typical in discussions of integer algorithms, because one version of the proposed algorithm computes α using floating-point arithmetic, where relative error is more natural. Besides, the "integer algorithms" are essentially floating-point algorithms in disguise, since they involve scaling shifts.

Theorem 2 Let $c=1/d$ for a positive integer d. Let $n\geq 1$. For real α such that $c \leq \alpha < c(1+2^{-n}+2^{-2n}+2^{-3n}+\ldots)$, then for all integers x such that $0 \leq x \leq 2^n-1$, $\lfloor x/d \rfloor = \lfloor \alpha x \rfloor$.

Proof: Examine Fig. 2. All that needs to be shown is that αx lies within the gray parallelogram, which by Lemma 1 suffices to prove the theorem. Clearly, the line αx passes through the origin, so the concern is whether αx exits through the right side of the parallelogram. From inspection, it is clear that the smallest valid α allows αx to be coincident with cx, and the largest valid α is bounded above by $h(x)$. The theorem follows from these bounds and the expansion $2^n/(2^n-1) = 1+2^{-n}+2^{-2n}+2^{-3n}+\ldots$. □

It should be clear from the diagram that the upper bound on α is not only sufficient, but also necessary.

Theorem 3 Let $c=1/d$ for a positive integer d. Let $n\geq 1$. For real α is such that $c(1-2^{-n}) \leq \alpha < c$, then for all integers x such that $0 \leq x \leq 2^n-1$, $\lfloor x/d \rfloor = \lfloor \alpha x+\alpha \rfloor$.

Proof: Similar to Theorem 2, all that needs to be shown is that $\alpha x+\alpha$ lies within the gray parallelogram in Fig. 2. The line $\alpha x+\alpha$ must pass through the left side of the parallelogram because $0 < c(1-2^{-n}) \leq \alpha < c$, and because $\alpha<c$, cannot get any closer to $cx+c$. So what is left to show is that it passes through the right side. If $\alpha=c(1-2^{-n})$, this condition is just barely met because then $\alpha x+\alpha$ becomes $g(x)$ in Fig. 2, which passes through the lower right corner of the

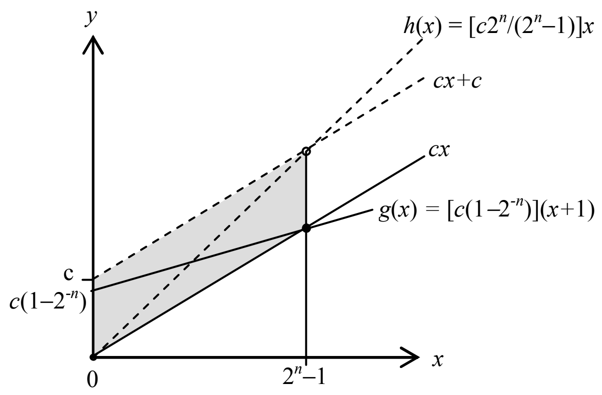

Fig. 2. Linear approximations for $\lfloor x/d \rfloor$ must lie within the gray parallogram. The upper side is open; the other three are closed.

parallelogram; higher values of α cause it to go through the side. □

Theorems 2 and 3 show that given an approximation $\alpha=(1/d)(1+\varepsilon)$, where ε is the relative error bounded by $-2^{-n} \leq \varepsilon < 2^{-n}+2^{-2n}+2^{-3n}+\ldots$, the unsigned quotient $\lfloor x/d \rfloor$ can be computed for an unsigned n-bit value x by using $\lfloor \alpha x \rfloor$ if $\varepsilon\geq 0$, or using $\lfloor \alpha x+\alpha \rfloor$ if $\varepsilon<0$.

Theorem 3 has the following corollary.

Corollary 4 If $\alpha=1-2^{-n}$, then for integers x such that $0 \leq x \leq 2^n-1$, $\lfloor x/1 \rfloor = \lfloor \alpha x+\alpha \rfloor$.

This provides a gimmick for making the XMA.HU instruction perform an identity operation. The obvious alternative formula $\lfloor 1x+0 \rfloor$ *cannot* be used to implement $\lfloor x/1 \rfloor$ using XMA.HU(a,x,b), because that would require $a=2^n$ and $b=0$, which makes a too big to represent in an n-bit word. Computing x with XMA.HU(2^n-1, x, 2^n-1) is crucial when a uniform formula must be used for all divisors.

4. Algorithms

The theorems of the previous section show that $\lfloor x/d \rfloor$ can be computed as $\lfloor \alpha x+\beta \rfloor$, where α is sufficiently close to $1/d$ and β is 0 or α, depending upon whether $a \geq 1/d$ or $a < 1/d$ respectively. The division algorithms to be presented evaluate $\lfloor \alpha x+\beta \rfloor$ by executing SHR.U(XMA.HU(a,x,b),m), where a, b, and m are integers such that

$$\alpha = a \times 2^{-m}$$
$$\beta = b \times 2^{-m}$$

Because β is always α or 0, it follows that b is always a or 0. The sections below examine two ways to compute a and decide whether $b=a$ or $b=0$.

4.1. Using integer arithmetic

Algorithm 1 employs strictly integer arithmetic to

```
Inputs: uword $d$ and $n$, with $n \ge 1$ and $1 \le d < 2^n$
int $m := \lfloor \log_2(d) \rfloor$;
uword $a$, $b$;
if $d=2^m$ then
        $a := 2^n - 1$;
        $b := 2^n - 1$;
else
        uword $t := \lfloor (2^{m+n})/d \rfloor$;
        uword $r := ((t \times d + d) \bmod 2^n$;
        if $r \le 2^m$ then
                $a := t+1$;
                $b := 0$;
        else
                $a := t$;
                $b := t$;
        endif
endif

Emit SHR.U(XMA.HU($a,x,b$),$m$)
```

Algorithm 1. Unsigned integer division via multiply-add, using integer arithmetic to pick a and b.

compute a and b. The pseudo-code notation is adapted from [6]. When $d=2^m$, Algorithm 1 reduces the multiply-add to an identity operation (per Corollary 4), and applies the obvious right shift to do the division by a power of 2. When $d \ne 2^m$, the algorithm computes the scaled and rounded-down reciprocal approximation t, and then decides whether to use the formula ax or the formula $ax+a$.

Algorithm 1 requires dividing a double word by a single word to compute t. The rest of the arithmetic is single-word arithmetic. Notably, it computes r from t and d using only n-bit unsigned arithmetic, as indicated by "mod 2^n". The low n bits suffice, because the test computes an n-bit residue described by Lemma 5 below.

Lemma 5 In Algorithm 1, $r=d(t+1)-2^{m+n}$.

Proof: Let $r' = d(t+1)-2^{m+n}$ and show that $r'=r$. Because t arises from integer division by d, it is constrained by $d > 2^{m+n}-dt \ge 0$. Multiply these expressions by -1 and add d to get transformed bounds $-d+d < dt-2^{m+n}+d \le 0+d$, which after simplification and substitution, simplifies to $0 < r' \le d$. Because $d < 2^n$, the former bounds imply $0 < r' < 2^n$. Thus $r = [d(t+1)-2^{m+n}] \bmod 2^n = d(t+1) \bmod 2^n$, the last equality arising from the fact that $2^{m+n} \bmod 2^n = 0$. □

The correctness of Algorithm 1 rests on the way it uses r to keep the relative error ε of a within the ranges required by Theorems 2 and 3, as shown respectively by Theorems 6 and 7 below.

Theorem 6 If $r \le 2^m$ in Algorithm 1, then for all integers x such that $0 \le x \le 2^n-1$, $\lfloor x/d \rfloor = \lfloor (t+1)x/2^{m+n} \rfloor$.

Proof: Let $c = 1/d$. Define α such that $\alpha = (t+1)/2^{m+n}$ and define ε implicitly as the solution to $\alpha = c(1+\varepsilon)$. The solution is $\varepsilon = \alpha/c - 1 = \alpha d - 1 = ((t+1)/2^{m+n})d - 1$. Refactor to get $\varepsilon = [d(t+1)-2^{m+n}]/2^{m+n}$. By Lemma 5, the bracketed expression is r, thus $\varepsilon = r/2^{m+n}$, which combined with the premise $r \le 2^m$ implies $\varepsilon \le 1/2^n$. The conclusion follows from Theorem 2. □

Theorem 7 If $r \ge 2^m$ in Algorithm 1, then for all integers x such that $0 \le x < 2^n$, $\lfloor x/d \rfloor = \lfloor (tx+t)/2^{m+n} \rfloor$.

Proof: Let $c = 1/d$. Define α that $\alpha = t/2^{m+n}$ and define ε implicitly as the solution to $\alpha = c(1+\varepsilon)$. Solving for ε yields $\varepsilon = \alpha/c - 1 = \alpha d - 1 = ((t/2^{m+n})d - 1$. Refactor to get $\varepsilon = [dt-2^{m+n}]/2^{m+n}$. Use the definition $r=d(t+1)-2^{m+n}$ from Lemma 5, and substitute for r in the premise $r \ge 2^m$ to get $d(t+1)-2^{m+n} \ge 2^m$. Subtract d from both sides to get $dt-2^{m+n} \ge 2^m-d$. Because $m=\lfloor \log_2(d) \rfloor$ and $d \ne 2^m$, it follows that $2 \times 2^m > d$, and thus $2^m-d > -(2^m)$. Combine the inequalities to get $dt-2^{m+n} > -(2^m)$. Divide both sides by 2^{m+n} to get $[dt-2^{m+n}]/2^{m+n} > -1/2^n$. Thus $\varepsilon > -1/2^n$. The conclusion follows from Theorem 3. □

The obvious algebraic equality $ax+a = a(x+1)$ exposes a certain poetic symmetry in Algorithm 1: "round up the reciprocal, or round up the dividend". However, $a(x+1)$ cannot generally be used for implementation, because $x+1$ might overflow, unless particular circumstance constrains $x<2^n-1$.

The last detail is to show that $a<2^n$, otherwise a might not fit in n bits.

Theorem 8 In Algorithm 1, $a<2^n$.

Proof: The theorem is trivial when $d=2^m$, because then $a=2^n-1$. If $d \ne 2^m$, then $2^m+1 \le d < 2^n$, with the following consequences:

$2^m+1 < 2^n$		
$0 < 2^n - 2^m + 1$	Subtract 2^m+1
$2^{m+n} < 2^{m+n} + 2^n - 2^m + 1$	Add 2^{m+n}
$2^{m+n} < (2^n-1)(2^m+1)$	Factor
$2^{m+n}/(2^m+1) < 2^n-1$	Divide by (2^m+1)
$2^{m+n}/(2^m+1) + 1 < 2^n$	Add 1

Combine with the definition of t in Algorithm 1, and the prior observation that $2^m+1 \le d$, to derive $t+1 \le (2^{m+n}/d)+1 < 2^n$. Finally, a is either t or $t+1$, so by the previous inequality, $a < 2^n$. □

4.2. Using floating-point Arithmetic

Algorithm 1 is fine for compile-time constant divisors, but for loop-invariant divisors, it may be quite expensive, because it requires an integer division of a $2n$-bit dividend by an n-bit divisor. Granlund and Montgomery [6] note a similar drawback, and that the cost of the calculations outside the loop might swamp any savings inside the loop. This section describes a

```
Inputs: uword $d$, with $1 \le d < 2^{64}$
uword a, b;
real $f$
if $d=1$ then
        $f := 1-2^{-64}$
else
        $y_0 := (1/d)(1+\delta),\ |\delta| \le 2^{-8.886}$
        $e_0 := (-d \times y_0 + 1)_{rn}$
        $e_1 := (e_0 \times e_0)_{rn}$            $e_2 := (e_0 \times e_0 + e_0)_{rn}$
        $y_1 := (e_2 \times y_0 + y_0)_{rn}$      $e_3 := (e_1 \times e_1 + e_0)_{rn}$
        $y_2 := (e_3 \times y_1 + y_0)_{rn}$
        $e_4 := (-d \times y_2 + 1)_{rn}$
        $f := (e_4 \times y_2 + y_2)_{rn}$
endif
$m$ := (BIAS−1) − EXPONENT($f$)
$a$ := SIGNIFICAND($f$)
if $(-d \times f + 1)_{rn} \le 0$ then
        $b := 0$;
else
        $b := a$;
endif

Emit SHR.U(XMA.HU($a,x,b$),$m$)
```

Algorithm 2. Unsigned integer division via multiply-add, using floating-point arithmetic to pick the *a* and *b*.

variation that overcomes this objection when appropriate floating-point hardware is available.

Algorithm 2 applies floating-point arithmetic to compute *a, b,* and *m* when n=64. It is specialized to n=64 because it iteratively approximates $1/d$ with a custom iteration sequence that depends upon n and the machine architecture. The sequence, adapted from a double-extended precision division sequence [8], employs a reciprocal approximation instruction to initialize an initial estimate y_0, and fused multiply-add operations to refine that estimate. In the style of [8], potentially concurrent multiply-add operations are shown on the same line. The sequence is typical on machines that omit direct support for floating-point division. If the hardware has direct support, then $1/d$ rounded-to-nearest to n bits of significance may be used instead.

Algorithm 2 presumes that the exponent range is large enough to prevent exponent underflow or overflow. The value f approximates $1/d$ with a relative error ε of at most 2^{-64} [8]. The test $(-d \times f+1)_{rn} \le 0$ determines whether this error is positive or negative; the particular form of the test allows it to be performed by a fused multiply-add. The correctness of Algorithm 2 follows from Theorems 2 and 3.

```
Inputs: uword $d$, with $1 \le d < 2^{64}$
uword a, b;
real $f$
if $d=1$ then
        $f := 1-2^{-64}$
else
        $y_0 := (1/d)(1+\delta),\ |\delta| \le 2^{-8.886}$
        $e_0 := (-d \times y_0 + 1)_{rn}$
        $y_1 := (e_0 \times y_0 + y_0)_{rn}$      $e_1 := (e_0 \times e_0)_{rn}$
        $y_2 := (e_1 \times y_1 + y_1)_{rn}$
        $e_2 := (-d \times y_2 + 1)_{rn}$
        $y_3 := (e_2 \times y_2 + y_2)_{rn}$
        $e_3 := (-d \times y_3 + 1)_{rn}$
        $f := (e_3 \times y_3 + y_3)_{rn}$
endif
$m$ := (BIAS−1) −EXPONENT($f$)
$a$ := SIGNIFICAND($f$)
$b := \lfloor a/2 \rfloor$

Emit SHR.S ($x$+XMA.HS($a,x,b$), $m$)
```

Algorithm 3. Computing $\lfloor x/d \rfloor$ for a 64-bit signed *x* and unsigned *d.*

4.3. Signed division

There does not appear to be any benefit to using the multiply-add approach for signed division that rounds towards zero. Constructing a diagram similar to Fig. 1 for signed division reveals that the approximating function must be piecewise linear, with separate pieces for positive and negative x. Prior methods employing multiplication by a rounded up reciprocal suffice because each piece spans a range of 2^{n-1}, and thus an n-bit multiplier suffices [6].

However, in applications that require rounding the quotient towards $-\infty$, a linear approximation $\lfloor (ax+b)/2^m \rfloor$ does offer an improvement when the divisor is unsigned. Algorithm 3 shows such an algorithm. Remarkably, no conditional logic is necessary to compute b from a. This simplification is offset by a complication: the computation $ax+b$ needs to be performed with an unsigned a and signed x, which is not in the assumed instruction repertoire. Fortunately, because the most significant bit of a is always 1, a two's complement reinterpretation a' of a has the algebraic value $a' = a-2^n$, hence XMA.HS(a,x,b) $= \lfloor ((a-2^n)x+b)/2^n \rfloor = \lfloor (ax+b)/2^n \rfloor - x$. The algorithm corrects for the "$-x$" error by adding x back.

Algorithm 3 is similar to Algorithm 2, but has a different iteration sequence that delivers a more accurate reciprocal approximation f that is $1/d$ rounded

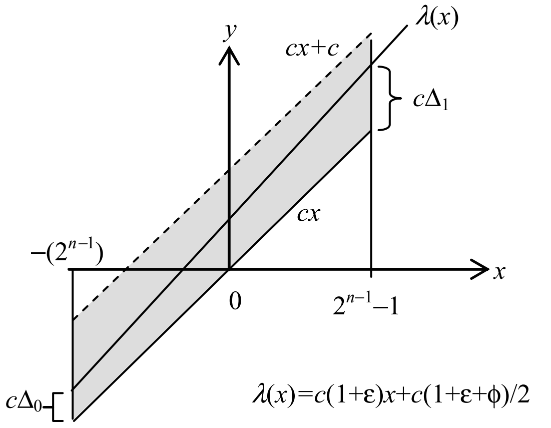

Fig. 3. Construction used in proof of Algorithm 3.

to nearest, except when $d=2^n-1$ [4]. Increased accuracy is used because the computation $b:=\lfloor a/2\rfloor$ introduces further truncation error that must be accounted for quite carefully. Algorithm 3 computes the integer quotient $\lfloor x/d\rfloor$ via the formula $\lfloor\lambda(x)\rfloor$, where $\lambda(x) = c(1+\varepsilon)x + (c/2)(1+\varepsilon+\phi)$. The relative error ε results from approximating the exact reciprocal $c=1/d$ with $a2^{-m}$. The relative error ϕ results from approximating $c/2$ with $\lfloor a/2\rfloor 2^{-m}$. The correctness of Algorithm 3 follows from Lemma 1 if in Fig. 3, the line $\lambda(x)$ stays within the gray parallelogram over the range $2^{n-1} \le x \le 2^{n-1}-1$. The theorems below bound ε and ϕ, and then apply those bounds to prove $0\le\Delta_0<1$ and $0\le\Delta_1<1$, which implies that $\lambda(x)$ is constrained as necessary.

Though Algorithm 3 is specific to $n=64$ because of the particular iterations used to approximate $1/d$, the proofs below depend only on the assumption that $n\ge1$ and f is an n-bit round-to-nearest approximation of $1/d$, with extra latitude granted in two cases. When $d=1$, the "one off" approximation $f=2^n-1$ is allowed, or when $d=2^n-1$, f may be rounded down instead of to nearest. Thus Algorithm 3 can be adapted to other values of n simply by computing f suitably.

The rest of this section details the necessary theorems.

Theorem 9 Define ε implicitly by the equation $f=(1/d)(1+\varepsilon)$, where f and d are from Algorithm 3. Then $-(2^{-n}) \le \varepsilon \le 2^{-n}-2^{-2n+1}$.

Proof: Suppose f is the round-to-nearest floating-point approximation of $1/d$ with an n-bit significand. Cornea-Hasegan, Golliver, and Markstein show that an integer quotient of two n-bit integers cannot be arbitrarily close to the midpoint between two n-bit significands, and the closest-to-midpoint case occurs for $1/d$ when $d=2^n-1$ [4]. The rounded-up n-bit floating-point approximation f of this value is $2^{-n}+2^{-2n+1}$. Since $\varepsilon = fd-1$, this implies the bound $|\varepsilon| \le (2^{-n}+2^{-2n+1})(2^n-1)-1 = (1-2^{-n}+2^{-n+1}-2^{-2n+1})-1 = 2^{-n}-2^{-2n+1}$.

The extra latitude allowed for the cases $d=1$ and $d=2^n-1$ results in a negative relative error bounded by $\varepsilon \ge -(2^{-n})$. □

Theorem 10 Define ϕ such that $b=(1/d)(1+\varepsilon+\phi)/2$, where b and d are from Algorithm 3, and ε is defined as in Theorem 9. Then $-(2^{-n+1}) < \phi \le 0$. Furthermore, if $\varepsilon < -(2^{-n}-2^{-2n+1})$ then $\phi \ge -(2^{-n})$.

Proof: The computation $\lfloor a/2\rfloor$ is equivalent to zeroing the least significant bit in a, which introduces an absolute error of 0 or -1, and then halving that result exactly. Thus the relative error ϕ introduced cannot be positive. Since a has n bits with a leading bit of 1, and a must be odd for the zeroing to have any effect, the smallest value of a for which ε is non-zero is $a=2^{n-1}+1$. Thus the relative error is bounded from below by $\phi \ge -1/(2^{n-1}+1) > -(2^{-n+1})$.

The "furthermore" portion of the theorem follows from the observation that $\varepsilon < -2^{-n}+2^{-2n+1}$ occurs only in two situations: when $d=1$, or when $d=2^n-1$ and f is $1/d$ rounded down instead of to nearest. When $d=1$, the value f is $1-2^{-n}$. The least significant bit of f is a one, which when zeroed introduces an absolute error of -2^{-n}. Converting this to a relative error (relative to 1) yields $\phi = -2^{-n}$. When $d=2^n-1$ and f is $1/d$ rounded down, then $f=2^{-n}$, which has a zero in the least significant bit, and thus $\phi=0$ in that case. □

Theorem 11: In Fig. 3, $0\le\Delta_0<1$.

Proof: From consideration of the geometry in Fig. 3, the following algebraic manipulations yield an equation for Δ_0.

$$c\Delta_0 = \lambda(x)-cx \text{ where } x= -(2^{n-1})$$
$$c\Delta_0 = \lambda(-(2^{n-1}))-c(-(2^{n-1}))$$
$$c\Delta_0 = c(1+\varepsilon)(-(2^{n-1})) + (c/2)(1+\varepsilon+\phi) - c(-(2^{n-1}))$$
$$\Delta_0 = -(2^{n-1}) - \varepsilon(2^{n-1}) + 2^{-1} + \varepsilon 2^{-1} + \phi 2^{-1} + (2^{n-1})$$
$$\Delta_0 = \varepsilon(-2^{n-1}+2^{-1}) + \phi 2^{-1} + 2^{-1}$$

A lower bound on Δ_0 is implied by the upper bound $\varepsilon \le 2^{-n}-2^{-2n+1}$ and lower bound $\phi > -(2^{-n+1})$. Plugging these bounds into the equation for Δ_0 yields $\Delta_0 > (2^{-n}-2^{-2n+1})(-2^{n-1}+2^{-1}) + (-2^{-n+1})2^{-1} + 2^{-1} = (-2^{-1}+2^{-n-1}+2^{-n}-2^{-2n}) -2^{-n} +2^{-1} = 2^{-n-1}-2^{-2n}$. The final right side is non-negative for $n\ge0$.

An upper bound on Δ_0 is implied by the lower bound $\varepsilon \ge -(2^{-n})$ and upper bound $\phi \le 0$. Plugging these inequalities into the equation for Δ_0 derives the following: $\Delta_0 \le (-(2^{-n}))(-2^{n-1}+2^{-1}) + (0)2^{-1} +2^{-1} = (2^{-1} - 2^{-n-1}) + (2^{-1}) = 1-2^{-n-1} < 1$. □

Theorem 12: In Fig. 3, $0\le\Delta_1<1$.

Proof: From consideration of the geometry in Fig. 3, the following algebraic manipulations yield an equation for Δ_1.

$$c\Delta_1 = \lambda(x)-cx \text{ where } x=2^{n-1}-1$$

$$c\Delta_1 = \lambda(2^{n-1}-1)-c(2^{n-1}-1)$$
$$c\Delta_1 = c(1+\varepsilon)(2^{n-1}-1) + (c/2)(1+\varepsilon+\phi) - c(2^{n-1}-1)$$
$$\Delta_1 = (2^{n-1}-1) + \varepsilon 2^{n-1} - \varepsilon + 2^{-1} + \varepsilon 2^{-1} + \phi 2^{-1} - (2^{n-1}-1)$$
$$\Delta_1 = \varepsilon(2^{n-1}-2^{-1}) + \phi 2^{-1} + 2^{-1}$$

A lower bound on Δ_1 is implied by the lower bounds on ε and ϕ. The going gets tricky here, because the special case where $\varepsilon < -(2^{-n}-2^{-2n+1})$ needs the special bound on ϕ from Theorem 10.

For the usual case, the lower bounds are $\varepsilon \geq -(2^{-n})+2^{-2n+1}$ and $\phi \geq -(2^{-n+1})$. Plugging these inequalities into the equation for Δ_1 yields $\Delta_1 > (-(2^{-n})+2^{-2n+1})(2^{n-1}-2^{-1}) + (-(2^{-n+1}))2^{-1} + 2^{-1} = (-(2^{-1})+2^{-n-1}+2^{-n}-2^{-2n}) -2^{-n} + 2^{-1} = 2^{-n-1}-2^{-2n}$. Thus $\Delta_1 > 2^{-n-1}-2^{-2n}$. The right side is non-negative for $n \geq 0$.

For the special case $\varepsilon < -(2^{-n})+2^{-2n+1}$, Theorem 10 says that $\varepsilon \geq -(2^{-n})$ and $\phi \geq -(2^{-n})$. Plugging these inequalities into the equation for Δ_1 yields $\Delta_1 \geq (-2^{-n})(2^{n-1}-2^{-1}) + (-(2^{-n}))2^{-1} + 2^{-1} = (-2^{-1}+2^{-n-1}) -2^{-n-1} + 2^{-1} = 0$. Notice that there is no slack to spare.

An upper bound on Δ_1 is implied by the upper bounds $\varepsilon \leq 2^{-n}-2^{-2n+1}$ and $\phi \leq 0$. Plugging these inequalities into the equation for Δ_1 yields $\Delta_1 \leq (2^{-n}-2^{-2n+1})(2^{n-1}-2^{-1}) + (0)2^{-1} + 2^{-1} = (2^{-1}-2^{-n-1}-2^{-n}+2^{-2n}) + 2^{-1} = 1-[3(2^{-n-1})-2^{-2n}]$. The expression $[3(2^{-n-1}) - 2^{-2n}]$ is non-negative for positive n, thus $\Delta_1 < 1$. □

5. Performance analysis

For hardware that supports the necessary operations, the multiply-add approach to unsigned division permits shorter and faster instruction sequences for compile-time divisors than approaches that depend exclusively on the "round up" or "round down" methods, because it never has to synthesize n+1 bit arithmetic. Furthermore, it is fast enough to make hoisting of run-time loop-invariant divisors practical on machines with the necessary support. This section analyzes these advantages in detail when implemented on the Itanium® 2 processor.

Latencies of various instruction sequences can vary depending upon assumptions, so the following uniform assumptions are made when comparing latencies. Typically, formation of constants can be anticipated and hoisted to earlier program points, and thus do not contribute to critical paths in a program. Latencies will be measured between when the dividend x is known and the quotient y is produced. The Itanium® architecture has the further idiosyncrasy of performing general integer multiplication in the floating-point unit, thus necessitating extra instructions to move integer operands to and from the floating-point unit. The latencies measured below assume that x starts out and y ends up in an integer register.

The explicitly parallel Itanium® architecture allows the instructions for checking for divide-by-zero to be issued in parallel with the rest of the instruction sequence. Hence, such checking adds no latency.

For divisors known at compilation time, Algorithms 1 and 2 deliver instruction sequences that are at least as fast, and sometimes faster, than the sequences delivered by exclusive use of the round-up or round-down methods, because those methods sometimes require synthesis of n+1 bit multiplication [6] [7]. Fig. 4 lists some sequences for a 64-bit divide by 7. The notation ";;" delimits groups of instructions that can be issued simultaneously. Here, the round-up method needs a 65-bit reciprocal approximation. The multiplication instruction handles 64 bits, and the remaining sub/shr/add sequence synthesizes the extra bit. The sequence's latency from x to y is 19 clocks. In contrast, round-down method delivers a sequence requiring only 16 clocks, a 15.8% reduction. For some other divisors, the situation is reversed, and it is the round-down algorithm that needs 65 bits and thus a longer sequence to synthesize the missing bit. Algorithm 1 or 2 always delivers the shorter 16-clock sequence.

// Code emitted by round-up method
```
movl r10=0x2492492492492493 ;;
setf.sig f8=r10
setf.sig f7=x        ;;
xma.hu   f6=f8,f7,f0 ;;
getf.sig r9=f6       ;;
sub      r8=x,r9     ;;
shr.u    r3=r8,1     ;;
add      r2=r9,r3    ;;
shr.u    y=r2,2
```

// Code emitted by round-down method.
```
movl r10= 0x9249249249249248 ;;
setf.sig f8=r10
setf.sig f7=x        ;;
xma.hu   f6=f8,f7,f8 ;;
getf.sig r9=f6       ;;
shr.u    y=r9,2
```

Fig. 4. 64-bit computation of y=$\lfloor x/7 \rfloor$ on Itanium® architecture

```
// Adaptation of maximum throughput
// sequence from [3].
setf.sig f6=x                  ;;
fnorm.s1 f6=f6                 ;;
fma.s1   f9=f8,f6,f0           ;;
fnma.s1  f7=f7,f9,f6           ;;
fma.s1   f8=f7,f8,f9           ;;
fcvt.fxu.trunc.s1 f8=f8        ;;
getf.sig y=f8
```

```
// Sequence from Algorithm 1 or 2.
setf.sig f7=x          ;;
xma.hu   f6=a,f7,b     ;;
getf.sig r9=f6         ;;
shr.u    y=r9,m
```

Fig. 5. Loop-variant portions of instruction sequences for 64-bit unsigned division.

Algorithms 2 is particularly powerful because not only is the loop-variant sequence shorter than the worst-case sequence required by the round-up or round-down method, but the portion that depends only upon the divisor is relatively simple and fast. Consider an *i*-iteration loop with a loop-invariant divisor. The minimum latency instruction sequence for 64-bit unsigned integer division on an Itanium® 2 processor takes 43 clocks [3]. However, this sequence is not ideal for hoisting loop-invariant divisors because it uses the dividend early. A better approach is to use the minimum throughput sequence [3], which though it takes 47 clocks, uses the dividend relatively late. This sequence splits into a 26-clock loop-invariant (top of Fig. 5) and 31-clock loop-variant portion. For a loop with *i* iterations, the total latency is 26+31*i* clocks. In contrast, Algorithm 2 splits such that the total latency is 45+16*i*. (The loop-variant portion is shown at the bottom of Fig. 5.) Thus for as few as 2 loop iterations, Algorithm 2 comes out ahead (77 clocks vs. 88 clocks for i=2), and is only 7% slower even for a one-trip loop (61 clocks vs. 57 clocks for i=1). Furthermore, zero-trip loops are not ordinarily an issue, because when hoisting invariants, compilers typically insert a zero-trip guard that branches around computation of loop invariants, so the approach loses only for one-trip loops.

Because Algorithm 2 comes out ahead for even as few as two quotients that share a divisor, it can be profitable in situations besides loops. For instance, it can be applied where two subexpressions share a common divisor. Another application is hash tables, where hash functions of the form "... mod *d*" occur, where *d* is the size of the table. Algorithm 2 can quickly recompute *a, b,* and *m* when *d* changes, thus allowing table lookups to apply the concise multiply-add formula.

6. Related work

Algorithm 1 is similar to previous work, but there is literally "a bit" (but a crucial bit) of difference. Previous work [1][6] based on rounding up the reciprocal approximation requires that $\log_2(d)$ be rounded up, thus requiring up to n+1 bits to represent the multiplier *a*. Previous work [7] based on rounding down the reciprocal approximation also requires an n+1 bit multiplier in the worst case. Viewed from the perspective of Fig. 2, both methods try to draw a line from the left side to the right side of the gray parallelogram. But "round up" methods are constrained to make the line's slope at least *c*, and "round down" methods require the line's slope to be no more than *c*. In either case, *n*-bit multipliers quantize the slope of the line too coarsely to always keep it inside the parallelogram. By choosing between "round up" and "round down", the present work requires only an *n*-bit multiplier. The "missing" bit is the choice.

Magenheimer et al [7] give bounds on *b* when using a general formula $\lfloor (ax+b)/z \rfloor$, and recommend using $b = a+(z \bmod d)-1$ when *d* does not divide exactly into *z*. Algorithm 1, for which $z=2^n$, uses the same value for *b* when ($z \bmod d$) = 1, but otherwise uses a smaller value, namely *a,* that is smaller than $a+(z \bmod d)-1$. Using $b=a$ has an advantage when the divisor is a compile-time constant: reusing *a* rather than creating a distinct constant *b* avoids further raising register pressure. Of course if ($z \bmod d$) ≥ $d/2$, Algorithm 1 uses the "round up" approach instead, in which case b=0.

Similar work is found in [2], which also suggests using a formula of the form $\lfloor ax+b/2^m \rfloor$, where b=0 or $b=a$. However, that work has a much slower algorithm for choosing which formula to use, and no proof of correctness is given. The algorithm tries the round up method [6], and if the resulting *a* does not fit in *n* bits, tries to produce an *n*-bit *a* using the round down method [7]. If the round-down method fails, then as a last resort it sets *a* to a rounded up scaled reciprocal approximation and *b* is set to *a.* The tests for deciding which formula to use are much more complicated than the current work, making it an impractical approach for loop-invariant divisors. Furthermore, its last resort formula is always wrong for the case $x=d-1$. (The approximation will compute $\lfloor (a(d-1)+a)/2^m \rfloor$, which simplifies to $\lfloor (ad)/2^m \rfloor$. Because *a* was rounded up, this formula is doomed to deliver an answer of 1, not the correct quotient of 0.) The probable reason that the erroneous formula goes undetected is that it is used only if *both* the round-up algorithm and round-down algorithm fail to find an n-bit a, which as the present work makes clear, can never occur, and thus the "last resort" in the method is simply unnecessary. (As a sanity check of the proofs, the algorithm in [2] for computing a, b, and m for 32-bit divisors was subjected to exhaustive testing, and indeed the erroneous formula is never used.) A more recent edition (March, 2004) of [2] removes the "last resort", but without any proof that the remaining cases cover all situations correctly.

A quite different approach is to eliminate division via strength reduction [9]. That approach is limited to situations where the operands are constants or linear functions of loop indices. However, combining strength reduction with Algorithm 1 might yield further improvement by reducing the multiply-add computation $ax+b$ to an induction variable when the numerator x is itself an induction variable.

7. Summary

This paper proves that unsigned integer division by a compile-time or loop-invariant divisor can be accomplished by an n-bit integer multiply-add (that yields a $2n$-bit result) followed by a right shift. This is an improvement over previous methods that required $n+1$ bit arithmetic in the worst case. The necessary operands that depend on the divisor can be calculated quickly, making the technique economical for hoisting loop-invariant divisors even for short loops and quotients with shared divisors. Fused floating-point multiply-add instructions have a multitude of uses. Now there is one more reason for supporting its lesser-known integer cousin.

8. Acknowledgements

Marius A. Cornea-Hasegan enlightened me on some mysteries of division on the Itanium® architecture. Clay Breshears picked nits from an earlier draft. An anonymous referee suggested using numerical examples for demonstration, and pointed out how to tighten the iteration sequence in an earlier version of Algorithm 2.

9. References

[1] R. Alverson, "Integer Division Using Reciprocals," *Proceedings of the 10th IEEE Symposium on Computer Arithmetic* (Grenoble, France), June, pp. 186-190.

[2] AMD, *Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors*, September 2003, pp. 186-189.

[3] M. Cornea, J. Harrison, C. Iordache, B. Norin, and S. Story, *Division, Square Root and Remainder Algorithms for the Intel® Itanium™ Architecture,* November 2003.

[4] M. Cornea-Hasegan, R. Golliver, and P. Markstein, "Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms," *IEEE Symposium on Computer Arithmetic* (Adelaide, Australia), 1999 pp. 96-105.

[5] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys* 23(1), March 1992, pp. 5-48.

[6] T. Granlund and P. Montgomery, "Division by Invariant Integers Using Multiplication," *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (Orlando Florida), pp. 61-72.

[7] D. Magenheimer, L. Peters, K. Pettis, and D. Zuras, "Integer multiplication and division on the HP Precision Architecture," *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, pp. 90-99.

[8] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision,* Prentice-Hall, 2000, p. 123.

[9] J. Sheldon, W. Lee, B. Greenwald, and S. Amarasinghe, "Strength Reduction of Integer Division and Modulo Operations," *Languages and Compilers for Parallel Computing, 14th International Workshop* (Cumberland Falls, KY, August 1-3, 2001), pp. 254-273.