

Fast truncated multiply-and-divide by constants [ROUGH NON-REVIEWED DRAFT]

Paul Khuong (pvk@pvk.ca)

August 8, 2011

Abstract

Ways to efficiently implement $\lfloor n \frac{m}{d} \rfloor$, for $n \in [0, 2^w - 1]$ and an invariant fraction $f \equiv \frac{m}{d} \in \mathbb{Q}^+$, such that $\lfloor fn \rfloor \in [0, 2^w - 1]$, where w is the bit-width of machine words.

As usual, we try and do so with a base-two fixed-point approximation of f . For a practical, efficient, scheme, the essential question is how efficiently we can execute a potentially large integer multiplication followed by a right shift. This is mostly what determines the precision to which we can approximate f , and thus the ranges of inputs n over which the approximation will be exact.

Differs slightly from more traditional presentation (Alverson, Grantlund, Warren) because we don't try to determine a reasonable approximation scheme that is always precise enough. Instead, define a family of approximation schemes, and look for the simplest one that provides us with enough precision. Useful when we have tight range information on inputs.

Contribution: generalizes to multiplication by fractions, and able to exploit range info.

1 Multiply-and-shift

The goal is to compute the machine word $\lfloor \frac{\alpha n}{2^s} \rfloor$, with α and s constant integers, and n an unsigned word (or smaller).

Note that, without loss of generality, we may suppose $s \geq w$: the final result must fit in a word, thus the intermediate value $\alpha n < 2^{w+s}$, and $2^{w-s} \alpha n < 2^{2w}$. So, $\lfloor \frac{\alpha n}{2^s} \rfloor$ can be reformulated as $\lfloor \frac{2^{w-s} \alpha n}{2^w} \rfloor$, when $s < w$.

We may also assume that $\alpha < 2^s$. Otherwise, we may simply let $\alpha_h = \lfloor \frac{\alpha}{2^s} \rfloor$ and $\alpha_l = \alpha \bmod 2^s$ (i.e. $\alpha = 2^s \alpha_h + \alpha_l$); in that case, $\lfloor \frac{\alpha n}{2^s} \rfloor = \alpha_l n + \lfloor \frac{\alpha_l n}{2^s} \rfloor$.

Finally, we shall let $s \leq 2w$, as that is all the precision that we will need.

This section describes ways to implement all the cases needed in this paper, in approximately increasing order of complexity.

TODO: show how sometimes we can shift the argument instead of shifting the multiplier. Useful for $\alpha > 2^s$. Similarly, can split f in two, one part natural, other rational. Worst case, additional mul and add, but can often get away with only add, or shift and add (i.e. lea).

1.1 No multiplication at all

When α is a power of two, the operation reduces to, at most, a single shift.

1.2 Trivial case

The easiest case is when αn is known to fit in a machine word. Then, it's a straightforward (low) half-multiplication and shift.

```
multiply-and-shift-2 (a, n, s)
  t = a * n -- regular half multiplication
  return t >> s
```

1.3 High half multiplication

We assume the existence of an instruction to compute the high half of an unsigned multiplication, `mulhi` (on x86[-64], this is simply a full multiplication, with the lower half, `rAX`, ignored)). When α fits in a machine word, and $s = w + l$ ($l \geq 0$), we can use this instruction almost directly.

```
multiply-and-shift-3 (a, n, ell)
  t = mulhi(a, n) -- high half of multiplication
  return t >> ell
```

On some micro-architectures, `mulhi` is less efficient than the more common half multiplication, even in conjunction with a shift. In that case, the trivial implementation in 1.2 is preferable when applicable. However, some other micro-architectures implement both multiplications equally efficiently, and it may then be advantageous to force $s = w$ when possible to fully exploit this quick `mulhi` and eschew the final shift.

1.4 Wide multiplication

This is the general case presented in Granlund & Montgomery; it is also reminiscent of Alverson's implicit-leading-bit fixed-point approximation. Looks like Warren as well.

When $2^w < \alpha < 2^{w+1}$ and $s = w + l$ ($l \geq 0$), we can multiply separately by $\alpha_1 = \alpha - 2^w$ and by 2^w . As noted in 1.3, we can always suppose that $s \geq w$.

If $s = w$, we have

```
multiply-and-shift-4a (a1, n)
  t1 = mulhi(a1, n)
  return t1 + n
```

Otherwise, $l > 0$, and

```
multiply-and-shift-4b (a1, n, ell)
  t1 = mulhi(a1, n)
  t2 = (n - t1) >> 1 -- trick to avoid overflow
  return (t1 + t2) >> (ell - 1)
```

1.5 Very wide multiplication

If $2^{w+1} < \alpha < 2^{2w}$, and $s \geq w$, it is possible to split the multiplier α in two, $\alpha_h = \lfloor \frac{\alpha}{2^w} \rfloor$ and $\alpha_l = \alpha \bmod 2^w$, so that $\alpha = 2^w \alpha_h + \alpha_l$.

Again, let $s = w + l$, $l \geq 0$. If the intermediate multiplied value is guaranteed not to overflow two words, we can use the following

```
multiply-and-shift-5a (ah, al, n, ell)
  t1 = ah * n -- implicit high half.
  t2 = mulhi(al, n)
  return (t1 + t2) >> ell -- assume no overflow here
```

If there could be an overflow we can instead use (if overflow is possible, $l > 0$) the following. The sequence for overflow-safe averaging is different from 4b: we don't know which argument is larger.

```

multiply-and-shift-5b (ah, al, n, ell)
  t1 = ah * n
  t2 = mulhi(al, n)
  t3 = (t1 & t2) + ((t1 ^ t2) >> 1) -- credit aggregate/hakmem[?] for avg
  return t3 >> (ell-1)

```

avg: hakmem 23 (Schroeppel)

Some ISAs have a way to shift the carry flag into the result of an addition (e.g. RCR on x86[-64]). This may constitute a more efficient manner to compute t3.

Finally, if $2^{2w} < \alpha < 2^{2w+1}$, the multiplier can be divided in three: 2^{2w} , $\alpha_h = \lfloor \frac{\alpha - 2^{2w}}{2^w} \rfloor$, and $\alpha_l = \alpha \bmod 2^w$. For such an α , $s = 2w + l$, and we have, for $l = 0$:

```

multiply-and-shift-5c (ah, al, n)
  t1 = ah * n
  t2 = mulhi(al, n)
  t3 = (t1 & t2) + ((t1 ^ t2) >> 1)
  return n + (t3 >> (w-1))

```

or, for $l > 0$ (if $\alpha < 2^{2w}$, the faster overflow-avoidance trick may be used for the last sum, t4)

```

multiply-and-shift-5d (ah, al, n, ell)
  t1 = ah * n
  t2 = mulhi(al, n)
  t3 = ((t1 & t2) + ((t1 ^ t2) >> 1)) >> (w-1)
  t4 = (n & t3) + ((n ^ t3) >> 1)
  return t4 >> (ell-1)

```

The routines in this subsection are probably fairly expensive, even compared to a division. However, they are only used for fractions in which the numerator $m > 1$. In those cases, they replace not a division, but a full multiplication and a 2 word by 1 word division (an operation that is rarely available in hardware). In fact, the last routine is only included for completeness's sake, and is not used by the approximation scheme described in this paper.

1.6 Very high shift count

When s is so high that the result can only take very few values, it may be simpler to implement the operation with comparisons than to try and use any of the above.

2 Error bounds for fixed-point truncated multiplication

Given an imperfect upper-approximation $\tilde{f} = f + \delta$ ($\delta > 0$) of $f = \frac{m}{d}$ ($m, d \in \mathbb{N}$), what is the least $n \in \mathbb{N}$ for which $\lfloor fn \rfloor \neq \lfloor \tilde{f}n \rfloor$?

Lemma:

$$\forall a, d \in \mathbb{N}, \quad \frac{a}{d} \leq \rho < \frac{a+1}{d} \Rightarrow \lfloor \rho \rfloor = \left\lfloor \frac{a}{d} \right\rfloor$$

(ST100, Muller & Tisserand, no proof, Robison05, more specific case, but only to show correctness of worst-case approximations).

The floor operator is monotone, so $\lfloor \frac{a}{d} \rfloor \leq \lfloor \rho \rfloor$.

If $a \bmod d < d - 1$, the conclusion is straightforward, as $\lfloor \frac{a}{d} \rfloor = \lfloor \frac{a+1}{d} \rfloor$.

If, otherwise $a \bmod d = d - 1$, $\lfloor \frac{a+1}{d} \rfloor = \frac{a+1}{d} = \lfloor \frac{a}{d} \rfloor + 1$, and we find

$$\left\lfloor \frac{a}{d} \right\rfloor \leq \rho < \frac{a+1}{d} = \left\lfloor \frac{a}{d} \right\rfloor + 1$$

It is then straightforward to see that, given $\tilde{f} = f + \delta$ ($\delta > 0$), $f = \frac{m}{d}$, and $n \in \mathbb{N}$

$$\begin{aligned} \frac{mn}{d} &\leq \tilde{f}n < \frac{mn+1}{d} \\ \Leftrightarrow 0 &\leq \delta n < \frac{1}{d} \\ \Leftrightarrow n &< \frac{1}{d\delta} \end{aligned}$$

The resulting bound $n < (d\delta)^{-1}$ (or $n < \lceil \frac{1}{d\delta} \rceil - 1$) is also found (in a less general form) in Cavagnino & Werbroeck. Note that, for a given shift s , the upper-approximation of f that minimises δ is $\tilde{f} = \frac{\lceil f2^s \rceil}{2^s}$, with $\delta < 2^{-s}$.

Add note: when $(n, d) > 1$, we have better bounds. At extreme, $(n, d) = d$, and we have $n < \frac{1}{\delta}$. For $n < 2^w$, we only need $\tilde{f} = \frac{\lceil 2^w f \rceil}{2^w}$. Of course, the euclidean algorithm gives us an even better solution, given that $(d, 2) = 1$.

2.1 Correctness when $m = 1$, and $d < 2^w$

When $m = 1$ (i.e. the classic case of division-by-multiplication), it is always possible to find a close approximation \tilde{f} of $f = \frac{1}{d}$ such that the constant multiplier uses at most one more bit than a machine word, and the resulting approximation is exact for every machine word.

Let w be the width of machine words and \lg denote the base-2 logarithm. For a given shift amount s , we have $\alpha = \lceil \frac{2^s}{d} \rceil$. $s = w + \lceil \lg d \rceil$ yields

$$\begin{aligned} \alpha &= \left\lceil \frac{2^{w+\lceil \lg d \rceil}}{d} \right\rceil \\ &\leq \left\lceil 2^w \frac{2^{\lceil \lg d \rceil}}{d} \right\rceil \\ &\leq \lceil 2^w \cdot 2 \rceil \\ &= 2^{w+1} \end{aligned}$$

With such an approximation, $\delta < 2^{-(w+\lceil \lg d \rceil)} \leq (2^w d)^{-1}$, and $n^0 \geq (d\delta)^{-1} > 2^w$.

2.2 Correctness when $1 < m < d < 2^w$

If we, again, fix $s = w + \lceil \lg d \rceil$, the approximation is correct for machine words. However, α may take significantly more bits: $\alpha \leq m2^{w+1} < 2^{2w+1}$. Given that $s = w + \lceil \lg d \rceil \leq 2w$, in the worst case, `multiply-and-shift-5c` will be used.

2.3 Correctness when $m > d$

$m > d$ presents more complications, as an appropriate multiply-and-shift routine may not always be available. However, it is always possible to split m in m_q and m_r such that $m = dm_q + m_r$. We then have $\lfloor \frac{mn}{d} \rfloor = m_q n + \lfloor \frac{m_r n}{d} \rfloor$.

2.4 Increasing the correct range with an unconditional addition

A different approximation is sometimes useful. Rather than approximating $f = \frac{m}{d}$ with $f + \delta$, we approximate fn by $\underline{f}n + \beta$, with $\underline{f} = f - \delta$ ($\delta > 0$) and $\beta = \frac{1}{d} - \epsilon$, $\epsilon > 0$.

In that case, we clearly have

$$\underline{fn} + \beta < fn + \frac{1}{d}$$

It remains to see for which values of n we can guarantee that $fn \leq \underline{fn} + \beta$

$$\begin{aligned} fn &\leq \underline{fn} + \beta \\ \Leftrightarrow \delta n &\leq \beta \\ \Leftrightarrow n &\leq \frac{\beta}{\delta} \\ \Leftrightarrow n &\leq \frac{1}{d\delta} - \frac{\epsilon}{\delta} \end{aligned}$$

Finally, when $\frac{\epsilon}{\delta} \leq 1$ (this is always the case when $m = 1 \Rightarrow \underline{f} = \beta$), we can simplify the condition:

$$\begin{aligned} n &\leq \frac{1}{d\delta} - \frac{\epsilon}{\delta} \\ \Leftrightarrow n &\leq \left\lfloor \frac{1}{d\delta} \right\rfloor - 1 \\ \Leftrightarrow n &< \left\lceil \frac{1}{d\delta} \right\rceil - 1 \end{aligned}$$

When the intermediate result of the multiplication is larger than a word, it's probably simpler to use one more bit with the usual over-approximation. Similarly, when the multiplication fits in a word, but not the addition, the cost of an overflow-free average doesn't seem worth the trouble.

However, when the result of the intermediate multiplication and addition fit in a word, the additional range offered by the additive correction may let us avoid a more complex code sequence. The range improvement over the correction-less sequence is usually fairly small, and thus only useful for upper bounds that are smaller than powers of two. In these cases, the operation can usually be implemented by incrementing the input before multiplying.

Conjecture: for $f = \frac{1}{d}$ and power of two input range, it's never useful.

3 Code generation