

**FAST TRUNCATED MULTIPLY-DIVIDE BY POSITIVE  
CONSTANTS  
[DRAFT]**

PAUL KHUONG (PVK@PVK.CA)

1. INTRODUCTION

The transformation of unsigned integer division by constants to sequences of integer multiplication, addition and shift has been a common component of strength reduction passes in optimizing compilers for a few decades [Tera?]. Hacker folklore also includes a few additional special cases [DrDobbs] not commonly generated by compilers. This paper first unifies two common approaches to the transformation, and shows how to efficiently select and generate corresponding implementations. The transformation algorithm is efficient, and yields code sequences that are equivalent to or simpler than those generated by state of the art compilers like GCC or ICC. The second half of this paper generalizes the approaches and the transformation to overflow-less multiply-divide by constants, i.e. truncated multiplication by fractions. The techniques described in this paper have been implemented in Steel Bank Common Lisp (SBCL), an interactive optimizing native-code compiler for Common Lisp.

In the general case (the runtime variant argument is an arbitrary machine word), the code generator for division by constants described in this paper yields equivalent or marginally more efficient code than [Granlund & Montgomery]. However, modern static analyses and the prevalence of languages implemented with tagged arithmetic regularly lets optimizers exploit the restricted range of their inputs. In that case, our generator is able to derive more efficient code sequences not only than those generated by GCC or ICC<sup>1</sup>, but also than those generated by the more sophisticated algorithm proposed in [Granlund & Montgomery]. A few notes relevant to compilers for implementations that use tagged arithmetic are also included.

Multiply-divide is a generalization of integer division: rather than computing  $\lfloor x/d \rfloor$ ,  $\text{muldiv}(x, a, d)$  computes  $\lfloor ax/d \rfloor$ . Unlike integer division, multiply-divide has received next to no attention from compiler writers. In fact, even though it is difficult to express in high level programming languages such as C, very few languages expose the operator (Forth, CL, ??). Yet, it is an essential component of overflow-safe programming, and is implemented, usually in assembly language, in many common libraries or systems (Windows's `stdlib`, the Linux kernel, `plan9`, `bsd`, ...). It seems plausible that the lack of compiler support for this operations makes programmers wary of using it, rightfully fearing that it will be as slow as a machine division even in easily optimized cases. The second half of this paper presents a code generator for `muldiv` that is similar in compile-time complexity and

---

<sup>1</sup>These compilers do not seem to exploit range analyses when optimizing division by constants, even though it is available, e.g. for constant folding.

runtime efficiency to the one we propose for division, yielding code that incurs, at most, three multiplications.

## 2. RELATED WORK

Both the optimizing-compiler literature and hacker folklore reveal two main ways to quickly compute the truncated division of a non-negative integer by a positive constant divisor. The more common one [Tera, GCC, ICC ???] performs a fixed-point multiplication by an over-approximation of the reciprocal, before truncating the result to its integer part. Conversely [hackerdom, intel], it is also possible to perform a fixed-point multiplication by an under-approximation of the reciprocal, but add a small fixup constant to the temporary value before truncating to an integer.

For both approaches, we derive the ranges for which the approximation is exact with lemma 2.1; each approach represents an extremal case of that condition that maximises the correctness range. The resulting code generator is able to better exploit finer ranges of inputs than a simple bit-width, and is generalized to multiplication by fractions.

In certain situations, [GCC & ???] propose a preprocessing step to increase the correctness range of a fixed-point approximation. The simplest one, applicable when the division is even, is also used here, and generalized to muldiv, by exploiting lemma 2.2.

There are no doubt myriad ways to implement this operation, especially when conditionals or lookup-tables are considered []. However, the fixed-point approximations have the virtue of being easy to analyse and prove correct, and easy to implement efficiently: choosing a radix-two format ensures that the truncation step is a simple shift.

There seems to be very little formal literature on the topic of muldiv. [longdiv] mentions an algorithm to compute division of double-word values by constants, which would be useful when only the divisor is known. Otherwise, a few codebases [Linux] express it in terms of multiple integer division and modulo operations, enabling a few optimisations when the fraction is constant. In contrast, the code generator presented here reduces the operation to at most three multiplications.

## 3. CORRECTNESS CONDITIONS

This subsection establishes sufficient conditions for the correctness of approximations of  $\lfloor x/d \rfloor$ , when the approximations are of the form  $\lfloor mx/2^s \rfloor$  or  $\lfloor (mx + b)/2^s \rfloor$ .

**3.1. Lemmas.** Throughout this paper, we will exploit the two following lemmas.

The first lemma, found in [Muller, Robison] forms the basis of the correctness bounds:

**Lemma 3.1.**

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

The second lemma guarantees that we can safely implement a single truncated division (or multiply-divide) as two truncated divisions (or one multiply-divide and one truncated division).

**Lemma 3.2.** *Let  $d = d_1 d_2$ , with  $d_1, d_2 \in \mathbb{N}$ ;*

$$\forall n \in \mathbb{N}, \quad \left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{\lfloor n/d_1 \rfloor}{d_2} \right\rfloor$$

*Proof.* Let  $q = \lfloor n/d \rfloor$ , and  $r = n - qd < d$ .

We have

$$\left\lfloor \frac{n}{d_1} \right\rfloor = qd_2 + \left\lfloor \frac{r}{d_1} \right\rfloor$$

and

$$\left\lfloor \frac{r}{d_1} \right\rfloor < \frac{d}{d_1} = d_2.$$

Finally

$$\left\lfloor \frac{\lfloor n/d_1 \rfloor}{d_2} \right\rfloor = \left\lfloor q + \frac{\lfloor r/d_1 \rfloor}{d_2} \right\rfloor = q = \left\lfloor \frac{n}{d} \right\rfloor$$

□

**3.2. Correctness of the over-approximation scheme.** In the over-approximation scheme, multiplication of  $x \in \mathbb{N}$  by the reciprocal  $r = 1/d$  is approximated with a multiplication by  $\tilde{r} = \lceil 2^s/d \rceil / 2^s$ .

**Theorem 3.3.** *Let  $\delta = \tilde{r} - r$  ( $\delta \geq 0$ ).*

*If  $\delta = 0$ , the approximation is perfect and is correct for any value of  $x$ .*

*Otherwise, the approximation is correct for  $x < 1/(d\delta)$ .*

*Proof.* Lemma 2.1 gives us the following correctness condition (the other inequality is always satisfied):

$$\begin{aligned} \tilde{r}x &< \frac{x+1}{d} = rx + \frac{1}{d} \\ \Leftrightarrow \delta x &< \frac{1}{d} \\ \Leftrightarrow x &< \frac{1}{d\delta} \end{aligned}$$

□

Equivalently, for an approximation that is correct for every  $x \leq x^*$ , we need  $\delta < 1/(dx^*)$ .

Note that, for a given divisor  $2^s$ ,  $\delta$  is minimised by our choice of  $\tilde{r}$ ; it represents one extreme case of correct approximations with respect to lemma 2.1.

Finally, the same bound can be directly applied, *mutatis mutandis*, to multiply-divide.

**3.3. Correctness of the under-approximation scheme.** In this scheme, multiplication of  $x \in \mathbb{N}$  by  $r$ , the reciprocal of  $d$ , is approximated with a multiplication by  $\tilde{r} = \lfloor 2^s/d \rfloor / 2^s$ , followed by an addition of  $\tilde{r}$ .

**Theorem 3.4.** *Let  $\delta = r - \tilde{r}$  ( $\delta \geq 0$ ).*

*If  $\delta = 0$  we could instead exploit the (perfect) over-approximation scheme.*

*Otherwise, the approximation is correct for  $x \leq 1/(d\delta) - 1$ .*

*Proof.* Lemma 2.1 gives us the following condition (the remaining inequality is always satisfied):

$$\begin{aligned} \frac{x}{d} &\leq \tilde{r}x + \tilde{r} = \frac{x}{d} - \delta x + \tilde{r} \\ \Leftrightarrow 0 &\leq -\delta x + \tilde{r} \\ \Leftrightarrow \delta x &\leq \tilde{r} \\ \Leftrightarrow x &\leq \frac{\tilde{r}}{\delta} = \frac{1}{d\delta} - 1 \end{aligned}$$

□

Equivalently, for an approximation that is correct for every  $x \leq x^*$ , we need  $\delta < 1/[d(x^* + 1)]$ .

In this scheme as well, for a given divisor, our choice of  $\tilde{r}$  minimises  $\delta$ , and represents the other extreme correctness case with respect to lemma 2.1.

In the general case of truncated multiplication by a fraction, the additive constant may be different from the multiplicative constant: rather than an under-approximation of the fraction, it must be a (strict) under-approximation of  $1/d$ .

#### 4. DIVISION BY CONSTANTS

We implemented the simplification of division by constants as a two-step process. First, the constants are generated, and then the least expensive feasible code sequence emitted. In the general case, the over-approximation scheme is slightly more efficient; under-approximation is only exploited in one special case.

In order to minimise the complexity of the generated code sequence, we search for the smallest constants that are correct for the known input range. For each scheme, the multiplier is uniquely determined by, and proportional to, the shift value  $s$ . Moreover, given a correct approximation for a shift value  $s$ , it is always possible to scale the constants to yield a correct approximation with a shift value  $s' > s$ . Thus, we always seek the correct approximation with the least shift value  $s$ .

**4.1. Notation.** Throughout this section,  $w$  will denote the bit-width of machine words, and  $\text{blen}$  the binary length of an integer, i.e.

$$\text{blen } n = \lfloor \log_2 n \rfloor + 1.$$

Note that  $\text{blen}$  is such that  $2^{\text{blen } n - 1} \leq n < 2^{\text{blen } n}$ .  
mulhi: ...

**4.2. Generating over-approximation constants.** In the spirit of Granlund, we determine the smallest (or nearly) correct multiplier  $m$  and shift value  $s$  by computing a pair of values that enclose correct approximations of  $x/d$  by  $mx/2^s$  (for the given range of inputs  $x$ ) and reducing their precision as much as possible.

Let  $s$  be a tentative shift value. Per lemma 2.1, any correct over-approximation  $\tilde{r} = m/2^s$  must be greater than  $r = 1/d$ ; i.e.  $m > \lfloor 2^s/d \rfloor$ .

Let

$$m_l(s) = \left\lfloor \frac{2^s}{d} \right\rfloor.$$

```

find-over-approximation-constants(max, d)
  max_shift = blen(max) + blen(d)
  scale     = 1<<max_shift
  low      = floor(scale/d)
  k        = ceiling(scale/max)-1
  high     = floor((scale+k)/d)
  diff     = low ^ high
  delta    = blen(diff)-1
  return high>>delta, max_shift-delta

```

FIGURE 1. Algorithm to compute a multiplier and shift pair that correctly approximates  $1/d$  for values of  $x \leq \max$ .

For any  $s = s^* - \Delta_s$  ( $\Delta_s \in \mathbb{N}$ ), we can easily compute  $m_l(s)$  given  $m_l(s^*)$ : per lemma 2.2,

$$m_l(s) = \left\lfloor \frac{m_l(s^*)}{2^{\Delta_s}} \right\rfloor.$$

Moreover, per theorem 2.3,  $\delta = \tilde{r} - r$  must be such that  $0 \leq \delta < 1/(x^*d)$ ; i.e.  $m \leq \lfloor (2^s + k)/d \rfloor$ , for some  $k$  such that  $0 \leq k < 2^s/x^*$ .

Let  $k = \lceil 2^s/x^* \rceil - 1$  and

$$m_h(s) = \left\lfloor \frac{2^s + \lceil 2^s/x^* \rceil - 1}{d} \right\rfloor;$$

in order to satisfy the condition in theorem 2.3, it suffices that  $m/2^s \leq m_h(s')/2^{s'}$  for some  $s'$ .

Thus, for any  $s = s^* - \Delta_s$  ( $\Delta_s \in \mathbb{N}$ ), the condition may be strengthened into <sup>2</sup>

$$m \leq \left\lfloor \frac{m_h(s^*)}{2^{\Delta_s}} \right\rfloor.$$

In order to minimize  $s$ , we wish to maximize  $\Delta_s$  such that

$$\left\lfloor \frac{m_l(s^*)}{2^{\Delta_s}} \right\rfloor \neq \left\lfloor \frac{m_h(s^*)}{2^{\Delta_s}} \right\rfloor.$$

As truncated division by a power of two is equivalent to a bitwise right shift on binary computed,  $\Delta_s$  can be easily computed by finding the most significant bit in which  $m_l(s^*)$  and  $m_h(s^*)$  differ. Of course, this depends on having an  $s^*$  such that  $m_l(s^*) \neq m_h(s^*)$ . Letting  $s^* = \text{blen } x^* + \text{blen } d$  suffices, and guarantees that  $s^* \leq 2w$ . This yields the algorithm in figure 1.

4.2.1. *Generating under-approximation constants.* We only attempt to use the under-approximation when  $x^* < 2^w - 1$ , and the over-approximation fails to yield a  $w$ -bit wide multiplier. We use  $s = \text{blen } x^* + \text{blen } d - 1$ , to ensure that the corresponding multiplier  $\lceil 2^s/d \rceil$  will be word-sized. Since the over-approximation scheme failed, we may assume that  $d$  is not a power of two. In fact, when the over-approximation fails to yield a  $w$ -bit multiplier, the equality condition below always stands (otherwise the over-approximation would be precise enough).

<sup>2</sup>When  $x^*$  is not a power of two,  $\lceil 2^s/x^* \rceil - 1 = \lfloor 2^s/x^* \rfloor$ , and, per lemma 2.2, this is strictly equivalent.

**Theorem 4.1.** *Let  $s = \text{blen } x^* + \text{blen } d - 1$ .*

*When*

$$m = \left\lfloor \frac{2^s}{d} \right\rfloor = \text{round} \left( \frac{2^s}{d} \right),$$

*the strict underapproximation  $\tilde{r} = m/2^s$  is correct  $\forall x \leq x^* < 2^w$ .*

*Proof.* Since the approximation is equal to the rounded value,

$$\delta \leq 1/2^{\text{blen } x^* + \text{blen } d - 1 + 1}.$$

Finally, it is straightforward to see that

$$\begin{aligned} \delta &\leq 1/2^{\text{blen } x^*} \cdot 1/2^{\text{blen } d} \\ &\leq 1/[(x^* + 1)d] \end{aligned}$$

and thus the condition in theorem 2.4 is satisfied.  $\square$

4.2.2. *Generating code.* The code generator used in SBCL only recognizes a small number of code patterns (c.f. annex). It is never worse than the code generator proposed in Granlund & Montgomery, and in many cases slightly better.

- (1) Simpler strength reductions (identity, conversion to shift or constant folding) is applied if possible.
- (2) An over-approximation is generated from algorithm 1:
  - (a) When all the intermediate values are word-sized, a direct implementation is generated.
  - (b) When the multiplier is word-sized, a mulhi and a shift are used. For this to be possible, the shift must be at least word-long. If that is not the case, both the multiplier and the shift can be scaled up equally:  $m/2^s < 1$  so the multiplier is at most  $s$ -bit wide.
  - (c) When the divisor is even, or the value of  $x$  at most  $2^w - 2$ , different approximations can be used; otherwise, generate a multiplication by a  $w + 1$ -bit constant.
- (3) If no code has been generated, and the divisor  $d$  is even, lemma 2.2 lets us perform the division in two steps. Let  $p$  be the largest integer such that  $d|2^p$ . Granlund & Montgomery generate code to shift the bottom  $p$  bits, and then perform the rest of the truncation. Instead, we generate code to mask away the bottom  $p$  bits of the argument  $x$ , and then to divide by  $d/2^p$ , but with the shift amount increase by  $p$ . This is equivalent to shifting the temporary shifted value left then right by  $p$ .<sup>3</sup>
- (4) If no code has been generated and  $x$  is at most  $2^w - 2$ , generate an under-approximation sequence. Note that the increment and the multiplier are the same. Rather than multiply by  $\tilde{r}$  and increment the result by  $\tilde{r}$  again, we multiply  $x + 1$  by  $\tilde{r}$ .

When  $d$  is even, or  $x < 2^w - 1$ , the sequences consist, at most, of a mask or an increment, a multiplication and a shift, and may actually reduce to a single multiplication. Otherwise, the worst case corresponds to that found in Granlund & Montgomery (one multiplication, one subtraction, one addition and two shifts).

---

<sup>3</sup>Correctness can be derived directly by noting that lemma 2.1 works with fractions, so can reduce common factors out and increase precision proportionally “for free.”

4.2.3. *Considerations for tagged arithmetic.* The most important simplification for tagged arithmetic is probably to take into account the more restricted range of inputs when computing correct multipliers. The algorithm in Granlund is able to exploit the input range, but this is not exploited in the implementations found in GCC or ICC.

When a lowtag scheme is used, it is possible to mask away the low bits instead of shifting them out (to the right): the shift is a division by a power of two, and can be included in the divisor. However, it's often also possible to only mask away the tag bits, without shifting them: the result of the division will already be left-shifted by the right value, and the tag bits can be directly deposited. This is clearly particularly interesting when the fixnum tag bits are zero, a common design choice.

## 5. MULTIPLY-DIVIDE

Multiply-divide,  $\text{muldiv}(x, a, d)$ , computes  $\lfloor ax/d \rfloor$  without any intermediate overflow or loss of precision when the final result (and the three arguments) are word-sized. It is a useful operation to perform non-trivial conversions between word-sized values, e.g. unit conversion on fixed-point values. Given the lack of widespread compiler support, it is often implemented at the assembly level, making it opaque to the optimizer. In some cases, it is implemented as a sequence of many division, modulo and multiplication operations that are at best translucent to the optimizer. The approach presented in this section always generates better code than either of the latter two.

5.1. **Correctness.** Both the over-approximation and the under-approximation schemes can be generalized to truncated multiplication by fractions.

Theorem 2.3 is straightforward to generalize: rather than approximating  $1/d$ , we wish to approximate  $a/d$ . As lemma 2.1 is mainly concerned with the divisor  $d$ , the same bound  $x < 1/(d\delta)$  stands, except that  $\delta$  is the approximation error on  $a/d$ , not  $1/d$ .

When generalizing theorem 2.4, care must also be taken that only the multiplier approximates the fraction  $a/d$ : the additive constant (and the dividend in the correctness bound) strictly under-approximates  $1/d$ .

5.2. **Implementation.** Again, the optimizer is implemented as a two-step process. First correct constants are computed. Then, the code is generated, or the operation is simplified, and new constants computed. The code sequences to implement the under-approximation scheme is more complex for arbitrary fractions; thus, only over-approximations are considered.

5.2.1. *Generating over-approximation constants.* The constant generator is easily extended. Rather than approximating  $1/d$ , we wish to approximate  $a/d$ . Thus, we have

$$m \leq \left\lfloor \frac{a2^s + k}{d} \right\rfloor,$$

$$m_l(s) = \left\lfloor \frac{a2^s}{d} \right\rfloor,$$

and

$$m_h(s) = \left\lfloor \frac{m2^s + \lceil 2^s/x^* \rceil - 1}{d} \right\rfloor.$$

5.2.2. *Generating code for  $1 < a < d$ .* When a simple enough sequence suffices, the constant generated in 2.2.1 are used directly. However, when  $d$  is even, it is sometimes possible to reduce the case to a division.

- (1) When  $m$  is at most  $w + 1$  bit wide, the corresponding code is generated; the code is always comparable to that for a division.
- (2) When there exists a  $p \leq w$  such that  $d|2^p$  and  $a < 2^p$ , use lemma 2.2 and first generate a truncated multiplication by  $a/2^p$  as a single mulhi. Then, generate a division by  $d/2^p$ : since both  $d$  and the input range have been reduced, the remaining code will need a smaller multiplier, often  $w$  bit (or less) wide.
- (3) Otherwise, emit the full general case with a double-word multiplier, and a shift by two words.

In the worst case, the code contains two multiplications.

5.2.3. *Generating code for  $a > d$ .* Again, the operation is reduced to a simpler one (in this case,  $a < d$ ) when needed.

- (1) Although the (modified) algorithm presented in 2.2.1 always returns a correct approximation, the code patterns available in SBCL (c.f. annex) do not always suffice: we only implement multiplication by, at most, a double-word and a shift by  $2w$ . If they do, the corresponding code is generated.
- (2) Otherwise, multiplication  $a/d$  is simplified into a multiplication by its integer part  $q = \lfloor a/d \rfloor$  and the remainder  $r/d = a/d - q$ . Multiplication by  $q$  is an unsigned multiplication, while multiplication by  $r/d$  is generated according to the previous section ( $r < d$ ). Finally, both values are added together.

In the worst case, the code contains three multiplications. It would sometimes be possible to subtract only a power of two from  $a/d$ , yielding instead two multiplications (and an additional shift). However, considering the amount of independent work in the  $a < d$  code, the saving is likely marginal, especially on a superscalar out-of-order microarchitecture.

## 6. CONCLUSION

Modern compilers now regularly include range analyses or target runtime systems with tagged integers. This situation makes it more likely to be useful for strength reduction algorithms to exploit more precise static information than only “constant word” or “arbitrary word”. The code generator presented in this paper does so and thus manages to generate code that is often simpler (and never worse) than state of the art compilers like GCC or ICC, or even than the adaptive-precision algorithm described in Granlund & Montgomery.

The generator is also practical, even for an interactive compiler: section 2.3.1 shows how the constants can be determined efficiently. The compile-time cost of algorithm 1 is comparable to that of the algorithm described in G&M and used in GCC, but is better able to exploit more precise range information than simple bit-width.

Interestingly, our use of lemma 2.1 means that the correctness range can implicitly benefit from static information regarding the exact divisors of runtime variant

values. Classically, this information has only been used to optimize pointer subtraction (which includes an implicit division) and bitwise operations. However, it may be fruitful to also exploit it when optimizing integer division.

Finally, we have shown how overflow-free multiply-divide (muldiv) by constants can be optimized, at little additional cost compared to division by constants. muldiv is an essential component of overflow-safe programs; it is our hope that this line of work will increase the number of language implementations that support and optimize this operation. This would in turn enable more programmers to use the operation when they would otherwise believe it too expensive, and hopefully lead to safer better-performing programs.