Fast Barrett Inverse via Structured Divide-and-Conquer Collapse

Heath P. SCHULTZ (unaffiliated)

January 19, 2025

Abstract. We present a new divide-and-conquer method for computing scaled reciprocals and quotients of large integers that eliminates one of the two recursive calls traditionally required in Barrett-style and divide-and-conquer division. The key observation is that the high half of the quotient, once computed, contains sufficient information to reconstruct the low half using only truncated multiplications and a constant and level-independent number of correction steps. This binary "collapse" reduces the per-level cost of reciprocal construction and quotient lifting while preserving constant correction bounds. We link to our complete reference implementation, including support for odd bit sizes, and empirically validate correctness across all inputs up to word scale. The technique primarily applies to any setting in which multiple divisions by a fixed denominator are performed, including cryptographic arithmetic, multiprecision libraries, and exact computation systems.

1. Introduction

Efficient division of large integers is a foundational problem in computer arithmetic. Modern high-performance implementations rarely perform division directly; instead, they compute an approximate reciprocal of the denominator and reduce division to multiplication and correction. Among the most influential of these techniques are Barrett reduction, Newton-Raphson inversion, and divide-and-conquer quotient construction. These methods form the backbone of contemporary multiprecision libraries and cryptographic arithmetic.

Despite decades of refinement, existing divide-and-conquer approaches to reciprocal construction and quotient extraction share a common structural feature: each recursion level typically requires two independent recursive calls to refine the approximation. While asymptotically optimal, this structure fixes the shape of the computation and its constant factors.

Archived Release (DOI): 10.5281/zenodo.18305611

This work shows that the second recursive call is not, in fact, necessary. We demonstrate that once the high half of the quotient has been computed, it can be used to derive the low half directly, using only truncated products and bounded correction steps. This collapses the recursion tree from two branches to one, altering the fundamental structure of the algorithm.

The result is a new divide-and-conquer collapse for scaled reciprocal and quotient computation. We present the idea, give a concrete working implementation, and discuss our other, more general implementations.

2. Problem Statement/Background

Let d be an odd positive integer of b bits, and let:

$$N = 2^{2b}$$

The goal is to compute the unique pair (q,r) such that:

$$N = qd + r, \quad 0 <= r < d$$

Equivalently, we seek a scaled reciprocal:

$$q = \left\lfloor \frac{N}{d} \right\rfloor$$

We also seek its exact remainder.

This is Barrett inversion. [1] The traditional application is Barrett reduction. Barrett reduction enables efficient reduction of multiple products modulo some fixed number. [2]

More generally, this form arises naturally in reciprocal-based division algorithms: once q is known, division of arbitrary numerators by the same denominator can be reduced to multiplication and correction. Less commonly, this method is also an approach for general integer division.

Divide-and-conquer approaches compute q by splitting both d and q into high and low halves and refining each half through two recursive calls. [4] The central question addressed here is:

> *Can the low half of the quotient be recovered using only the already-computed high half, without an additional recursive descent?*

We answer this in the affirmative by exhibiting a construction that derives the low half from the high half using truncated multiplications and bounded corrections, yielding the full quotient and remainder.

3. Algorithm Overview

This section will discuss the recursive version, and the implementation as an iteration will be discussed separately below.

3.1 Sentinel Pair

We'll first dispense with the requisite preliminary sentinel. For reasons shown below, in the general case, this is a final recursive request for a constant value, namely, the canonical Barrett inverse and remainder of one. This is the pair (4,0), that is, four remainder zero. We denote this function *Sentinel()*.

3.2 Divide-and-Conquer

The algorithm proceeds as a modified Divide-and-Conquer division applied to the numerator implicit in calculation of canonical Barrett inverses.

Let *D* be the input, that is, the denominator whose reciprocal and remainder that we are computing, and let *N* be the numerator implied by definition of Barrett inverses.

$$N = 4^{log2(D+1)}$$

So the Barrett inverse is a $2w$-by-$1w$ division. Divide-and-Conquer (D&C) treats this as a $4w$-by-$2w$ division and calculates it by recursively by performing a consecutive and dependent pair of $3w$-by-$2w$ divisions. Through this lens, D&C can be viewed as schoolbook division in base *B* where:

$$B = 2^{log2(D+1)}$$

Therefore the operands to our lower divisions are:

$$Numerator = N_2B^2 + N_1B + N_0$$

$$Divisor = D_1B + D_0$$

D&C is able to efficiently turn this into a recursion because:

$$Q_p \approx \frac{N_2B+N_1}{D_1} \approx \frac{N_2B^2+N_1B+N_0}{D_1B+D_0}$$

D&C lifts the $2w$-by-$1w$ approximate quotient to the full $3w$-by-$2w$ quotient by performing a single multiplication step.

For clarity, a worked numeric example in base-10 is called for. The inverse and remainder of 61 follows.

First, we need to know the implied numerator, *N*. Working in base-10, we first find the number of digits in *D*.

$$Digits = \lceil log10(D+1) \rceil = 2$$

The implied numerator in Barrett inversion is:

$$N = B^{2Digits} = 10000$$

So our result pair (*Q*,*R*) is the quotient and remainder of 10000 divided by 61.

$$\begin{array}{r l} & Q \text{ remainder } R \\ 61 & \overline{)10000} \end{array}$$

Recalling the $3w$-by-$2w$ structure of each recursive D&C step, we arrange the first division as follows:

$$\begin{array}{rl} & Q_1 \text{ remainder } R_1 \\ 61 & \overline{)1000} \end{array}$$

Calculate Q_1 and R_1

$$\begin{array}{rl} & \;\;\;16 \text{ remainder } 24 \\ 61 & \overline{)1000}0 \\ & \;-976 \\ & \;\;\;\;24 \\ \hline & \;\;\;\;24\,0 \end{array}$$

Calculate Q_0 and R:

$$\begin{array}{rl} & 163 \text{ remainder } 57 \\ 61 & \overline{)240} \\ & \underline{-183} \\ & \;\;57 \end{array}$$

Combine Q_1 and Q_0:

$$Q = Q_1B + Q_0 = 16 \times 10 + 3 = 163$$

Our result is 163 remainder 57.

Check:

$$163 \times 61 + 57 = 10000$$

We purposely glossed over a couple of issues.

First, 1000 has four digits, not three. However, it's the smallest natural number greater than the largest three digit natural number, 999.[1] We'll further defer explaining why this works until Section 5.

We also treated the $3w$-by-$2w$ calculation of the quotients as though we had access to an oracle. In order for our example to be useful in showing how a recursion is possible, we will now separately show how the $3w$-by-$2w$ to $2w$-by-$1w$ reduction is obtained.

Recall:

$$Q_p \approx \frac{N_2B+N_1}{D_1} \approx \frac{N_2B^2+N_1B+N_0}{D_1B+D_0}$$

We'll use the second quotient that we previously divined: 240 divided by 61.

Compare with an extreme example of the principle:

$$q = \frac{8175309}{903576}$$

By shifting we get an approximation. For example:

$$\frac{81}{9} \approx \frac{8175309}{903576}$$

Unsurprisingly, it turns out that floor of the quotient is exactly nine as predicted by the approximation.

Similarly, we can approximate the quotient from our example:

$$\frac{24}{6} \approx \frac{240}{61}$$

1 It's worth noting that a modified Barrett inversion can give the same results by diminishing the numerator by one and then increasing the quotient by one if the denominator is a power of two.

This example differs from our extreme example in that the approximation turns out to be inexact.

$$\left\lfloor \frac{24}{6} \right\rfloor > \left\lfloor \frac{240}{61} \right\rfloor$$

The exact quotient is three, and our approximation was four.

In order to correct this, we calculate the approximate remainder as usual.

$$R = n - QD$$

In our example:

$$R = 240 - 4 \times 61 = -4$$

The remainder being negative, we overshot the quotient. Adding the denominator once to this remainder makes the remainder valid, so the quotient is reduced to three and the remainder to increased to 57.

Checking:

$$QD + R = N = 3 \times 61 + 57 = 240$$

We converted our $3w$-by-$2w$ division into a a $2w$-by-$1w$ division at the cost of one multiplication. In practice, our result is obtained by treating each $2w$-by-$1w$ division as a *smaller* $4w$-by-$2w$ division, the result of which we can obtain recursively, as this was the form of the input to our function.

The recursion proceeds until the width of the denominator is one digit, and then Sentinel()[2] is called, and the recursion terminates.

3.3 Our Collapse Algorithm

Our method is based on the observation that Q_1 *is* the Barrett inverse of the truncated denominator used for both of the divisions in D&C. As such, *by definition* it can be used to perform the second division of D&C by Barrett inversion without making the second recursive call. Therefore, the binary tree implied by two recursive calls collapses. The number of multiplications *per level* increases slightly, but the *total* number is reduced exponentially.

Although the above description is technically complete, it's treated more fully below in Section 4. For now, we'll refer back to our previous example to demonstrate our restructuring.

First, we look at how far we were able to get in our inversion using only one recursive call to D&C.

We were able to obtain the high half of the quotient and remainder modulo D.

2 In base-10, an alternative Sentinel() would be needed because the quotient and remainder are only constant in base-2.

$$\begin{array}{r l}
 & 16 \;\text{remainder } 24 \\
61 & \overline{)10000}\,0 \\
 & -976 \\
 & 24 \\
\hline
 & 24\,0
\end{array}$$

Making further progress requires that we perform the second division, namely 240 divided by 61.

Previously, we prepared for the second recursive division by approximating our operands in anticipation of our ability to lift the result to a full quotient. That is, we observed that:

$$\frac{24}{6} \approx \frac{240}{61}$$

Because our approximation caused our division operation to have $2w$-by-$1w$ operand sizes, we were able to obtain the result recursively.

But our recursive call wasn't special. We can substitute the second recursive call with any method that computes a $2w$-by-$1w$ quotient and remainder.

The observation of our D&C collapse algorithm is that we do have such an alternative way to obtain this quotient. When we calculated the high half, we obtained the Barrett reciprocal of the high half of D. The high half of D is what we are trying to divide by.

It follows immediately from this that we can obtain our quotient and remainder by multiplying Q_1 with our residual. However, recall that Barrett inverses are scaled representations of an inverse, and accordingly our product must be scaled. Our implicit numerator was 10000, so the half scaling factor is 100.

Therefore, our second quotient approximation is obtained as follows:

$$Q = \left\lfloor \frac{16 \times 24}{100} \right\rfloor = \left\lfloor \frac{384}{100} \right\rfloor = 3$$

This type of scaling doesn't involve an actual division by 100. It is, by definition, a truncation of the digits. In base-2, it amounts to the right shift operation.

Having obtained the second quotient approximation by alternative means, the D&C algorithm has all of the information that it needs to finish forming the full quotient and remainder.

4. Detailed Analysis

We review our method in detail by closely examining our basic recursive implementation.

As a preliminary, a standard correction function is supplied.

Algorithm 1 qr_cor — Correct quo./rem.

```
Require: $d, b, q, r, m, p$
Ensure: $(q, r) = (\lfloor 2^{2\cdot b}/d \rfloor, 2^{2\cdot b} - q \cdot d)$
1: for $i = 1$ to m do
2:     $t \leftarrow r < p$                ▷ Avoid signed compare.
3:     $q \leftarrow q - t$                ▷ Decrement $q$.
4:     $r \leftarrow r + d \cdot t$        ▷ Add $d$ to $r$.
5: end for
```

Our main collapse algorithm follows:

Algorithm 2 inv — D&C Collapse

```
Require: $d, b = \lceil log2(d) \rceil$     ▷ d = denom.
Ensure: $(q, r) = (\lfloor 2^{2\cdot b}/d \rfloor, 2^{2\cdot b} - q \cdot d)$
1: if $b = 1$ then
2:     $(q, r) \leftarrow (4, 0)$                ▷ sentinel
3:     return
4: end if
5: if $b \mod 2 = 1$ then                 ▷ Handle odd sizes.
6:     $d \leftarrow 2 \cdot d$
7:     INV$(d, b + 1, q, r)$               ▷ recursive call
8:     $r \leftarrow r + d \cdot (q \mod 2)$
9:     $d \leftarrow \lfloor d/2 \rfloor$
10:    $(q, r) = (\lfloor q/2 \rfloor, \lfloor r/4 \rfloor)$
11:    return
12: end if
13: $h \leftarrow \lfloor b/2 \rfloor$
14: $d_{hi} \leftarrow \lfloor d/2^h \rfloor$
15: INV$(d_{hi}, h, q, r)$
16: $d_{lo} \leftarrow d - d_{hi} \cdot 2^h$
17: $r \leftarrow r \cdot 2^h$
18: $p \leftarrow q \cdot d_{lo}$                  ▷ Mul 1
19: QR_COR$(r, q, d, p, 4)$                ▷ Correction 1
20: $r \leftarrow r - p$
21: $q0_1 \leftarrow \lfloor (q \cdot r)/2^b \rfloor$        ▷ Muls 2/3
22: $p \leftarrow q0_1 \cdot d_{hi}$                ▷ Mul 4
23: $r \leftarrow r - p$
24: for $i = 0$ to 2 do                    ▷ Correction 2
25:     $t = r \geq d_{hi}$
26:     $q \leftarrow q + t$
27:     $r \leftarrow r - d_{hi} \cdot t$
28: end for
29: $r \leftarrow r \cdot 2^h$
30: $p \leftarrow q0_1 \cdot d_{lo}$                ▷ Mul 5
31: QR_COR$(r, q0_1, d, p, 2)$             ▷ Correction 3
32: $q \leftarrow (q \cdot 2^h) + q0_1$
33: $(q, r) \leftarrow (q, r - p)$
```

5. Correctness

We give a brief argument that the collapse algorithm returns the correct scaled quotient and remainder.

5.1 Invariant

At every call inv(d,b,q,r), the algorithm maintains the invariant:

$$2^{2b} = qd + r, \quad 0 \leq d,$$

where d is a positive integer of b bits.

The base cases $b < \varepsilon$ are computed directly and satisfy the invariant by construction.

5.2. Even bit sizes

Assume b is even and write $h = b/2$. Let:

$$d = d_{hi}2^h + d_{lo}, \quad 0 < d_{lo} < 2^h$$

The recursive call computes (q_1, r_1) such that:

$$2^{2h} = q_1 d_{hi} + r_1, \quad 0 \leq r_1 < d_{hi}$$

Multiplying by 2^h gives:

$$2^{3h} = q_1 d_{hi} 2^h + r_1 2^h$$

The algorithm forms:

$$r' = r_1 2^h - q_1 d_{lo}.$$

Since:

$$d = d_{hi}2^h + d_{lo},$$

this yields:

$$2^{3h} = q_1 d + r'.$$

The subsequent bounded correction loop adjusts (q_1,r') so that:

$$2^{3h} = q_1' d + r_2, \quad 0 \leq r_2 < d.$$

This establishes a correct high-half quotient.

5.3 *Low-half reconstruction (collapse)*

The algorithm then computes:

$$q_0 \approx \left\lfloor \frac{q_1' r_2}{2^{2h}} \right\rfloor.$$

Using truncated products and shifts, this produces an approximation to the low half of the quotient. Because $r_2 < d$ and q_1' is already a correct high-half quotient, standard Barrett analysis shows that:

$$\left| q_0 - \left\lfloor \frac{r_2 2^h}{d} \right\rfloor \right| \leq C$$

for a small constant C.

The algorithm subtracts $q_0 d_{hi}$ from r_2, lifts the residue, and subtracts $q_0 d_{lo}$. After each subtraction, a bounded correction loop is applied. These loops terminate after a fixed number of steps and restore:

$$2^{2b} = (q_1' 2^h + q_0')d + r, \quad 0 \leq r < d.$$

Thus the low half of the quotient is reconstructed without an additional recursive call.

5.4 *Odd bit sizes*

When b is odd, the algorithm reduces to the even case by computing the exact scaled problem:

$$2^{2(b+1)} = 2dQ + R$$

via a recursive call.

Because $2^{2(b+1)}$ is divisible by 4 and $2dQ$ is divisible by 2, the remainder R is always congruent to either 0 or 2 mod 4. If Q is odd, the algorithm adds $2d$ to R, making it divisible by 4. It then sets:

$$q = \lfloor Q/2 \rfloor, r = R/4.$$

A direct calculation shows that this transformation preserves the invariant:

$$2^{2b} = qd + r, 0 \leq r < d.$$

Thus odd sizes are reduced to the even case without altering the core algorithm.

5.5 *Termination and correctness*

Each recursive call reduces the bit size by approximately one half (or by one after the odd bridge), so the algorithm terminates at the base cases.

At every level:

- algebraic reconstruction maintains the invariant,
- correction loops enforce:

$$0 \le r < d,$$

- and the quotient is assembled from its high and low halves.

By induction on b, the algorithm returns the unique correct quotient and remainder satisfying:

$$2^{2b} = qd + r, 0 \le r < d.$$

---

6. Complexity

We analyze the asymptotic cost of the collapse algorithm under the standard multiplication-time model. [3] Let $M(n)$ denote the cost of multiplying two n-bit integers using a given multiplication algorithm.

Classical divide-and-conquer reciprocal and quotient construction proceeds by splitting the denominator into two halves and performing *two independent recursive calls per level*, followed by a constant number of multiplications and corrections. At bit width n, this gives a recurrence of the form:

$$T_{divcon}(n) = 2T_{divcon}(n/2) + O(M(n/2))$$

Under any superlinear multiplication model, this solves to:

$$T_{divcon}(n) = O(M(n)).$$

See Cormen, et al. [5]

Our collapse algorithm modifies only the structure of the recursion. Instead of two recursive calls, it performs *a single recursive call* to compute the high half of the quotient, then reconstructs the low half using truncated multiplications and bounded correction steps.

At each level, the algorithm performs:

- one recursive call on size $n/2$,
- a constant number of truncated products involving operands of size at most $n/2$ and n,
- and a bounded number of additions, subtractions, and shifts.

Therefore the recurrence becomes:

$$T_{col}(n) = T_{col}(n/2) + CM(n/2) + O(n)$$

for a small constant C determined by the number of truncated products.

Solving this recurrence yields:

$$T_{col}(n) = O(M(n))$$

Thus, like classical divide-and-conquer and Newton-based reciprocal methods, the collapse algorithm achieves the optimal asymptotic complexity dictated by multiplication.

The improvement lies not in the asymptotic class, but in the *structure of the recursion and the associated constants*. Classical divide-and-conquer methods must pay the cost of two recursive descents per level. The collapse eliminates the second descent entirely, replacing it with a fixed pattern of truncated multiplications and bounded corrections. This reduces the total number of large multiplications required to construct the reciprocal and quotient and removes an entire dependency chain from the recursion tree.

Because the collapse operates exclusively through truncated products and constant correction passes, its per-level work integrates naturally with modern multiplication backends. In particular, its total cost remains $O(M(n))$ under schoolbook, Karatsuba, Toom-Cook, or FFT-based multiplication, while strictly reducing the number of large-operand multiplications relative to classical divide-and-conquer division.

Finally, because the collapse derives the low half of the quotient directly from the high half, the leaf-level cost influences the total runtime less strongly than in two-branch recursion schemes. This makes the algorithm especially well-suited to settings where large divisions by a fixed denominator are performed repeatedly and where reciprocal construction dominates total cost.

7. Implementation Notes

7.1. Reference implementation

Reference implementation is available at: https://github.io/meathportes/dc-collapse

7.2. Constant time

The reference implementation *doesn't* run in constant time. The methods for obtaining constant run times are well-known, and support for constant time will be added.

7.3 Diminished representation

It turns out that implementing our algorithm benefits from an alternative representation of the quotient. Our reference implementation uses this representation. Accordingly, the invariant preserved is different. Instead of ensuring that every level of the algorithm preserves (q,r), representing q as q-2^b-1 while leaving r unchanged guarantees that q never occupies more bits than d. Otherwise, carry bits need to be tracked across levels. To convert from this *diminished* representation, q may be obtained by calculating $q = q'+2^b+1$.

7.4 Code paths

In progress is a fully optimized version of the algorithm implementing each of the natural code paths which arise from Divide-and-Conquer Collapse.

(a) **Multi-limb**. Our reference implementation doesn't yet contain a multi-limb version of the algorithm. It's expected that the full machine word version will easily adapt to multiple limbs because the same overflow concerns which may be expected to apply to a multi-limb implementation have already been worked out.

(b) **Machine word**. Our reference implementation contains an implementation of the algorithm for denominators which are as wide as the full machine word. Correctness has been validated empirically by means of fuzzer testing.

The machine word implementation also handle the case where the number of bits in the denominator is one less than the machine word width.

(c) **Sub-machine word high**. When $b > word\_bits \times 2 / 3$, an intermediate between the full overflow aware machine-word implementation and a naive method suffices. Specifically, it must properly handle the second and third multiplications (the 2x1 truncated multiplication).

This is currently not implemented.

(c) **Sub-machine word low**. For the lowest width denominators larger than the sentinel maximum, a fully overflow naive method suffices.

This is currently not implemented.

(d) **Sentinel**. Our sentinel function directly calculates the diminished quotient and the remainder for denominators less than four bits.

Correctness is confirmed by exhaustively checking all inputs.

7.5 Iteration

Some of the above described code paths lend themselves naturally to iterative implementation. Specifically, all of those paths which would never be called recursively more than once are subject to this optimization. Working iterative versions have been made but are not quite ready for release.

8. Conclusion

We have shown that the traditional two-branch divide-and-conquer structure used in reciprocal-based division is not intrinsic. The high half of the quotient contains enough information to recover the low half using truncated multiplications and a small, constant number of correction steps. Exploiting this fact collapses the recursion tree to a single branch per level.

We presented a complete reference implementation at sub machine-word scale, including support for odd bit sizes, and validated correctness across all tested inputs. The resulting method reduces the per-level multiplication work required to construct scaled reciprocals and full quotients, while remaining compatible with standard truncated-product and correction techniques.

This collapse opens a new point in the design space of large-integer division algorithms. It suggests that further structural reductions may be possible in other reciprocal-based methods, and it provides a new primitive for high-performance arithmetic in cryptography, multiprecision libraries, and exact computation systems. Future work includes lifting the method to full multi-limb arithmetic, integrating it with existing multiplication backends and developing formal correctness proofs and optimized implementations.

9. Contributors

Heath P. SCHULTZ
<meathportes@gmail.com>

ANONYMOUS (`EURnyhCmKagvIwhr`)[3]

Special thank you (in alphabetical order):

Bootsie D.

Bubba J.

Cyrus A.

Delilah C.

Elizabeth S.

Jenna R.

John S.

Natasha O.

3 The attribution tag is generated by a fixed procedure. Take the UTF-8 encoding of an ASCII string (M) and compute its SHA-256 digest (FIPS 180-4). Take the first 128 bits of that digest in big-endian order and write them in hexadecimal. Remove any leading zero hex digits, decode the remaining hex string to bytes, and base-64 encode the result using the standard RFC 4648 alphabet without padding. The resulting base-64 string is the attribution tag. Verification consists of recomputing this transform and checking equality with the published tag. The security of the tag relies on the preimage resistance of SHA-256.

Works Cited

[1] Barrett, Paul. *Implementing the Rivest–Shamir–Adleman Public-Key Encryption Algorithm on a Standard Digital Signal Processor*. Advances in Cryptology — CRYPTO ’86, Springer, 1987, pp. 311–323.

[2] Bernstein, Daniel J. *Multidigit Modular Arithmetic*. 2001.

[3] Brent, Richard P., and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[4] Burnikel, Christoph, and Joachim Ziegler. *Fast Recursive Division*. Journal of Symbolic Computation, vol. 42, no. 1–2, 2007, pp. 98–113.

[5] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.