Improved division by invariant integers

Niels Möller and Torbjörn Granlund

***Abstract*—This paper considers the problem of dividing a two-word integer by a single-word integer, together with a few extensions and applications. Due to lack of efficient division instructions in current processors, the division is performed as a multiplication using a precomputed single-word approximation of the reciprocal of the divisor, followed by a couple of adjustment steps. There are three common types of unsigned multiplication instructions; we define full word multiplication (`umul`) which produces the two-word product of two single-word integers, low multiplication (`umullo`) which produces only the least significant word of the product, and high multiplication (`umulhi`), which produces only the most significant word. We describe an algorithm which produces a quotient and remainder using one `umul` and one `umullo`. This is an improvement over earlier methods, since the new method uses cheaper multiplication operations. It turns out we also get some additional savings from simpler adjustment conditions. The algorithm has been implemented in version 4.3 of the GMP library. When applied to the problem of dividing a large integer by a single word, the new algorithm gives a speedup of roughly 30%, benchmarked on AMD and Intel processors in the x86_64 family.**

I. Introduction

Integer division instructions are either not present at all in current microprocessors, or if they are present, they are considerably slower than the corresponding multiplication instructions. Multiplication instructions in turn are at least a few times slower than addition instructions, both in terms of throughput and latency. The situation was similar a decade ago [1], and the trend has continued so that division *latency* is now typically 5-15 times higher than multiplication latency, and division *throughput* is up to 50 times worse than multiplication throughput. Another trend is that branches cost gradually more, except for branches that the hardware can predict correctly. But some branches are inherently unpredictable.

Division can be implemented using multiplication, by first computing an approximate reciprocal, e.g., by Newton iteration, followed by a multiplication that results in a candidate quotient. Finally, the remainder corresponding to this candidate quotient is computed, and if the remainder is too small or too large, the quotient is adjusted. This procedure is particularly attractive when the same divisor is used several times; then the reciprocal need to be computed only once. Somewhat surprisingly, a well-tuned Newton reciprocal followed by multiplication and adjustments wins over the hardware division instructions even for a single non-invariant division on modern 64-bit PC processors.

This paper considers the problem of dividing a two-word number by a single-word number, using a single-word approximate reciprocal. The main contributions are a new algorithm for division using such a reciprocal and new algorithms for computing a suitable reciprocal for 32-bit and 64-bit word size.

The key idea in our new division algorithm is to compute the candidate remainder as a single word rather than a double word, even though it does not quite fit. We then use a fraction associated with the candidate quotient to resolve the ambiguity. The new method is more efficient than previous methods for two reasons.

- It uses cheaper multiplication operations, omitting the most significant half one of the two products. Computing the least significant word of a product is a cheaper operation than computing the most significant word (e.g., on AMD Opteron, the difference in latency is one cycle, while on Intel Core 2, the difference is three cycles).
- The needed adjustment conditions are simpler.

When the division algorithms in this paper are used as building blocks for algorithms working with large numbers, our improvements typically affect the linear term of the execution time. This is of particular importance for applications using integers of size up to a few dozen words, e.g., on a 64-bit CPU, 2048-bit RSA corresponds to computations on 32-word numbers.

The new algorithms have been implemented in the GMP library [2]. As an example of the resulting speedup, for division of a large integer by a single word, the new method gives a speedup of 31% compared to earlier methods, benchmarked on AMD Opteron and Intel Core 2.

The outline of this paper is as follows. The rest of this section defines the notation we use. Section II explains how the needed reciprocal approximation is defined, and how it is useful. In Sec. III, we describe new algorithms for computing the reciprocal, and we present our main result, a new algorithm for dividing a two-word number by a single word. Analysis of the probability for the adjustment steps in the latter algorithm is provided in Appendix A. Section IV describes a couple of extensions, primarily motivated by schoolbook division, the most important one being a method for dividing a three-word number by a two-word number. In Sec. V, we consider an algorithm that can take direct advantage of the new division method: Dividing a large integer by a single-word. We describe the x86_64 implementation of this algorithm using the new method, and compare it to earlier results. Finally, Sec. VI summarises our conclusions.

A. Notation and conventions

Let ℓ denote the computer word size, and let $\beta = 2^\ell$ denote the base implied by the word size. Lower-case letters denote single-word numbers, and upper-case letters represent numbers

N. Möller is a long time member of the GMP research team. Email: `nisse@lysator.liu.se`

T. Granlund is with the Centre for Industrial and Applied Mathematics, KTH, Stockholm. Granlund's work was sponsored by the Swedish Foundation for Strategic Research. Email: `tege@nada.kth.se`

of any size. We use the notation $X = \langle x_{n-1}, \ldots, x_1, x_0\rangle = x_{n-1}\beta^{n-1} + \cdots + x_1\beta + x_0$, where the n-word integer X is represented by the words x_i, for $0 \le i < n$.

We use the following multiplication operations:

$\langle p_1, p_0\rangle \leftarrow \texttt{umul}(a,b) = ab$	Double word product
$p_0 \leftarrow \texttt{umullo}(a,b) = (ab) \bmod \beta$	Low word
$p_1 \leftarrow \texttt{umulhi}(a,b) = \left\lfloor \frac{ab}{\beta} \right\rfloor$	High word

Our algorithms depend on the existence and efficiency of these basic multiplication operations, but they do not require both `umul` and `umulhi`. These are common operations in all processors, and very few processors lack both `umul` and `umulhi` [1].

II. Division using an approximate reciprocal

Consider the problem of dividing a two-word number $U = \langle u_1, u_0\rangle$ by a single-word number d, computing the quotient and remainder

$$q = \left\lfloor \frac{U}{d} \right\rfloor \qquad\qquad r = U - qd.$$

Clearly, r is a single-word number. We assume that $u_1 < d$, to ensure that also the quotient q fits in a single word. We also restrict attention to the case that d is a "normalised" single-word number, i.e., $\beta/2 \le d < \beta$. This is equivalent to the word d having its most significant bit set. It follows that $u_0/d < 2$, and one can get a reasonable quotient approximation from u_1 alone, without considering u_0.

We have $1/\beta < 1/d \le 2/\beta$. We represent the reciprocal $1/d$ using a fixed-point representation, with a single word and an additional implicit one bit at the most significant end. We define the precomputed reciprocal of d as the integer

$$v = \left\lfloor \frac{\beta^2 - 1}{d} \right\rfloor - \beta. \tag{1}$$

The constraints on d imply that $0 < v < \beta$, in particular, v is a single word number. We have $(\beta + v)/\beta^2 \approx 1/d$, or more precisely,

$$\frac{1}{d} - \frac{1}{\beta^2} \le \frac{\beta + v}{\beta^2} < \frac{1}{d}. \tag{2}$$

For the borderline case $d = \beta/2$, we have the true reciprocal $1/d = 2/\beta$, which equals $(\beta+v)/\beta^2$ for $v = \beta$. Our definition instead gives the single-word number $v = \beta - 1$ in this case.

The usefulness of v comes from Eq. (2) which implies

$$\frac{U}{d} \approx (u_1\beta + u_0)\frac{\beta + v}{\beta^2} = u_1 + \frac{u_1 v}{\beta} + \frac{u_0}{\beta} + \frac{u_0 v}{\beta^2}. \tag{3}$$

Since $(\beta + v)/\beta^2 < 1/d$, the integer part of the right hand side is at most q, and hence a single word. Since the terms on the right hand side are non-negative, this bound is still valid if some of the terms are omitted or truncated.

[1]The SPARC v9 architecture is a notable exception, making high performance arithmetic on large numbers very challenging.

A. Previous methods

The trick of using a precomputed reciprocal to replace integer division by multiplication is well-known. The simplest variant is Alg. 1, which uses a quotient approximation based on the first two terms of Eq. (3).

```
(q, r) ← DIV2BY1(⟨u1, u0⟩, d, v)
    In: β/2 ≤ d < β, u1 < d, v = ⌊(β² − 1)/d⌋ − β
1   q ← ⌊vu1/β⌋ + u1      // Candidate quotient (umulhi)
2   ⟨p1, p0⟩ ← qd                                  // umul
3   ⟨r1, r0⟩ ← ⟨u1, u0⟩ − ⟨p1, p0⟩    // Candidate remainder
4   while r1 > 0 or r0 ≥ d      // Repeated at most 3 times
5       q ← q + 1
6       ⟨r1, r0⟩ ← ⟨r1, r0⟩ − d
7   return q, r0
```

Algorithm 1: Simple division of two-word number by a single-word number, using a precomputed single-word reciprocal.

To see how it works, let $U = \langle u_1, u_0\rangle$ and let q denote the true quotient $\lfloor U/d \rfloor$. We have $(\beta + v)\, d = \beta^2 - k$, where $1 \le k \le d$. Let q' denote the candidate quotient computed at line 1, and let $q_0 = vu_1 \bmod \beta$ denote the low, ignored, half of the product. Let R' denote the corresponding candidate remainder, computed on line 3. Then

$$\begin{aligned} R' &= U - q'd \\ &= u_0 + u_1\beta - \frac{u_1(\beta + v) - q_0}{\beta}\, d \\ &= u_0 + \frac{u_1 k + q_0 d}{\beta} \end{aligned}$$

We see that $R' \ge 0$, which corresponds to $q' \le q$. Since $k \le d$, we also get the upper bound $R' < \beta + 2d \le 4d$, which implies that $q' \ge q - 3$. Since R' may be larger than β, it must be computed as a two-word number at line 3 and in the loop, at line 5, which is executed at most three times.

The problem is that in the two-word subtraction $U - q'd$, most, but not all, bits in the most significant word cancel. Hence, we must use the expensive `umul` operation rather than the cheaper `umullo`.

The quotient approximation can be improved. By checking if $u_0 \ge d$, and if so, incrementing q' before computing r', one gets $R' < 3d$ and $q' \ge q - 2$. The method in [1], Sec. 8, is more intricate, guaranteeing that $R' < 2d$, so that $q' \ge q - 1$. However, it still computes the full product $q'd$, so this method needs one `umul` and one `umulhi`.

III. New algorithms

In this section, we describe our new algorithms. We first give efficient algorithms for computing the approximate reciprocal, and we then describe our new algorithm for division of a double-word number by a single word.

A. Computing the reciprocal

From the definition of v, we have

$$v = \left\lfloor \frac{\beta^2 - 1}{d} \right\rfloor - \beta = \left\lfloor \frac{\langle \beta - 1 - d, \beta - 1\rangle}{d} \right\rfloor$$

so for architectures that provide an instruction for dividing a two-word number by a single word, that instruction can be used to compute the reciprocal straightforwardly.

If such a division instruction is lacking or if it is slow, the reciprocal can be computed using the Newton iteration

$$x_{k+1} = x_k + x_k(1 - x_k d). \tag{4}$$

This equation implies that

$$1 - x_{k+1}d = (1 - x_k d)^2. \tag{5}$$

Consider one iteration, and assume that the accuracy of x_k is roughly n bits. Then the desired accuracy of x_{k+1} is about $2n$ bits, and to achieve that, only about $2n$ bits of d are needed in Eq. (4). If x_k is represented using n bits, matching its accuracy, then the computation of the right hand side yields $4n$ bits. In a practical implementation, the result should be truncated to match the accuracy of $2n$ bits. The resulting error in x_{k+1} is the combination of the error according to Eq (5), the truncation of the result, and any truncation of the d input.

$v \leftarrow$ RECIPROCAL_WORD(d)
In: $2^{63} \le d < 2^{64}$
1 $d_0 \leftarrow d \bmod 2$ // Least significant bit
2 $d_9 \leftarrow \lfloor 2^{-55} d \rfloor$ // Most significant 9 bits
3 $d_{40} \leftarrow \lfloor 2^{-24} d \rfloor + 1$ // Most significant 40 bits
4 $d_{63} \leftarrow \lceil d/2 \rceil$ // Most significant 63 bits
5 $v_0 \leftarrow \lfloor (2^{19} - 3 \times 2^8)/d_9 \rfloor$ // By table lookup
6 $v_1 \leftarrow 2^{11} v_0 - \lfloor 2^{-40} v_0^2 d_{40} \rfloor - 1$ // 2 `umullo`
7 $v_2 \leftarrow 2^{13} v_1 + \lfloor 2^{-47} v_1 (2^{60} - v_1 d_{40}) \rfloor$ // 2 `umullo`
8 $e \leftarrow 2^{96} - v_2 d_{63} + \lfloor v_2/2 \rfloor d_0$ // `umullo`
9 $v_3 \leftarrow (2^{31} v_2 + \lfloor 2^{-65} v_2 e \rfloor) \bmod 2^{64}$ // `umulhi`
10 $v_4 \leftarrow (v_3 - \lfloor 2^{-64}(v_3 + 2^{64} + 1)\, d \rfloor) \bmod 2^{64}$ // `umul`
11 **return** v_4

Algorithm 2: Computing the reciprocal $\lfloor (\beta^2 - 1)/d \rfloor - \beta$, for 64-bit machines ($\beta = 2^{64}$).

Algorithm 2 gives one variant, for $\beta = 2^{64}$. Here, v_0 is represented as 11 bits, v_1 as 21 bits, v_2 as 34 bits, and v_3 and v_4 as 65-bit values where the most significant bit, which is always one, is implicit. Note that since d_{40} and d_{63} are rounded upwards, they may be equal to 2^{40} and 2^{63} respectively, and hence not quite fit in 40 and 63 bits.

Theorem 1 (64-bit reciprocal): With $\beta = 2^{64}$, the output v of Alg. 2 satisfies $0 < \beta^2 - (\beta + v)\, d \le d$.

Proof: We will prove that the errors in each iteration are bounded as follows:

$$e_0 = 2^{50} - v_0 d_{40} \qquad |e_0| < \frac{5}{8} \times 2^{42} \tag{6}$$

$$e_1 = 2^{60} - v_1 d_{40} \qquad 0 \le e_1 < \frac{29}{32} \times 2^{43} \tag{7}$$

$$e_2 = 2^{97} - v_2 d \qquad 0 < e_2 < \frac{873}{1024} \times 2^{63} + d \tag{8}$$

$$e_3 = 2^{128} - (2^{64} + v_3)\, d \quad 0 < e_3 < 2d \tag{9}$$

$$e_4 = 2^{128} - (2^{64} + v_4)\, d \quad 0 < e_4 \le d \tag{10}$$

Each step involves a truncation, and we let $0 \le \delta_k < 1$ denote the truncation error in each step. Start with (6). Let $d' = d_{40} - 2^{31} d_9$, then $1 \le d' \le 2^{31}$. We have

$$\begin{aligned} v_0 &= \frac{2^{19} - 3 \times 2^8}{d_9} - \delta_0 \\ e_0 &= 2^{50} - \frac{2^{19} - 3 \times 2^8}{d_9}(2^{31} d_9 + d') + \delta_0 d_{40} \\ &= 3 \times 2^{39} + \delta_0 d_{40} - \frac{2^{19} - 3 \times 2^8}{d_9}\, d' \end{aligned}$$

From this, we get

$$\begin{aligned} e_0 &\le 3 \times 2^{39} + \delta_0 d_{40} \\ &< 3 \times 2^{39} + 2^{40} = 5 \times 2^{39} \\ e_0 &\ge 3 \times 2^{39} - \frac{2^{19} - 3 \times 2^8}{d_9} d' \\ &> 3 \times 2^{39} - 2^{42} = -5 \times 2^{39} \end{aligned}$$

For (7), we get

$$\begin{aligned} v_1 &= 2^{11} v_0 - 2^{-40} v_0^2 d_{40} - (1 - \delta_1) \\ e_1 &= 2^{60} - (2^{11} v_0 - 2^{-40} v_0^2 d_{40})\, d_{40} + (1 - \delta_1) d_{40} \\ &= 2^{-40} e_0^2 + (1 - \delta_1) d_{40} \end{aligned}$$

It follows that $e_1 > 0$ and that

$$e_1 < \left(\frac{5}{8}\right)^2 \times 2^{44} + 2^{40} = \frac{29}{32} \times 2^{43}$$

For (8), we first note that the product $v_1(2^{60} - v_1 d_{40})$ fits in 64 bits, since the first factor is 21 bits and the second factor is e_1, which fits in 43 bits. Let $d' = 2^{24} d_{40} - d$, then $1 \le d' \le 2^{24}$. We get

$$\begin{aligned} v_2 &= 2^{13} v_1 + 2^{-47} v_1 (2^{60} - v_1 d_{40}) - \delta_2 \\ e_2 &= 2^{97} - v_2 (2^{24} d_{40} - d') \\ &= 2^{97} - 2^{24}(2^{13} v_1 + 2^{-47} v_1 (2^{60} - v_1 d_{40})) d_{40} + v_2 d' + \delta_2 d \\ &= 2^{-23} e_1^2 + v_2 d' + \delta_2 d \end{aligned}$$

It follows that $e_2 > 0$ and that

$$e_2 < \left(\frac{29}{32}\right)^2 \times 2^{63} + 2^{58} + d = \frac{873}{1024} \times 2^{63} + d$$

For (9), first note that the value e, computed at line 8, equals $\lfloor e_2/2 \rfloor$. Then (8) implies that this value fits in 64 bits. Let ϵ denote the least significant bit of e_2, so that $e = (e_2 - \epsilon)/2$. Define

$$\begin{aligned} v_3' &= 2^{31} v_2 + \lfloor 2^{-66} v_2 (e_2 - \epsilon) \rfloor \\ e_3' &= 2^{128} - v_3' d \end{aligned}$$

(We will see in a moment that $v_3' = 2^{64} + v_3$, and hence also $e_3' = e_3$). We get

$$\begin{aligned} e_3' &= 2^{128} - (2^{31} v_2 + 2^{-66} v_2 (2^{97} - v_2 d - \epsilon))\, d + \delta_3 d \\ &= 2^{-66} e_2^2 + (2^{-66} v_2 \epsilon + \delta_3)\, d \end{aligned}$$

It follows that $e_3' > 0$ and that

$$e_3' < \left(\frac{873}{1024}\right)^2 \times 2^{60} + \left(\frac{873}{4096} + \frac{1}{4} + \frac{1}{2^{32}} + 1\right) d < 2d$$

$v \leftarrow$ RECIPROCAL_WORD(d)
In: $2^{31} \le d < 2^{32}$
1 $d_0 \leftarrow d \bmod 2$
2 $d_{10} \leftarrow \lfloor 2^{-22} d \rfloor$ // Most significant 10 bits
3 $d_{21} \leftarrow \lfloor 2^{-11} d \rfloor + 1$ // Most significant 21 bits
4 $d_{31} \leftarrow \lceil d/2 \rceil$ // Most significant 31 bits
5 $v_0 \leftarrow \lfloor (2^{24} - 2^{14} + 2^9)/d_{10} \rfloor$ // By table lookup
6 $v_1 \leftarrow 2^4 v_0 - \lfloor 2^{-32} v_0^2 d_{21} \rfloor - 1$ // `umullo` + `umulhi`
7 $e \leftarrow (2^{48} - v_1 d_{31} + \lfloor v_1/2 \rfloor d_0)$ // `umullo`
8 $v_2 \leftarrow 2^{15} v_1 + \lfloor 2^{-33} v_1 e \rfloor$ // `umulhi`
9 $v_3 \leftarrow (v_2 - \lfloor 2^{-32}(v_2 + 2^{32} + 1)\, d \rfloor) \bmod 2^{32}$ // `umul`
10 **return** v_3

Algorithm 3: Computing the reciprocal $\lfloor (\beta^2 - 1)/d \rfloor - \beta$, for 32-bit machines ($\beta = 2^{32}$).

It remains to show that $2^{64} \le v_3' < 2 \times 2^{64}$. The upper bound follows from $e_3' > 0$. For the borderline case $d = 2^{64} - 1$, one can verify that $v_3' = 2^{64}$, and for $d \le 2^{64} - 2$, we get

$$\begin{aligned} v_3' &= \frac{2^{128} - e_3'}{d} \ge \frac{2^{128} - e_3'}{2^{64} - 2} \\ &= 2^{64} + \frac{2 \times 2^{64} - e_3}{2^{64} - 2} > 2^{64}. \end{aligned}$$

For the final adjustment step, we have

$$\begin{aligned} \lfloor 2^{-64}(v_3 + 2^{64} + 1)d \rfloor &= \lfloor 2^{-64}(2^{128} - e_3 + d) \rfloor \\ &= 2^{64} + \lfloor 2^{-64}(d - e_3) \rfloor \\ &= \begin{cases} 2^{64} & e_3 \le d \\ 2^{64} - 1 & e_3 > d \end{cases} \end{aligned}$$

Hence, the effect of the adjustment is to increment the reciprocal approximation if and only if $e_3 > d$. The desired bound, Eq. (10), follows. ∎

Algorithm 3 is a similar algorithm for $\beta = 2^{32}$. In this algorithm, v_0 is represented as 15 bits, v_1 as 18 bits, and v_2 and v_3 as 33-bit values where the most significant bit, always one, is implicit. The correctness proof is analogous, with the following error bounds:

$$\begin{aligned} e_0 &= 2^{35} - v_0 d_{21} & |e_0| &< \frac{33}{64} \times 2^{26} \\ e_1 &= 2^{49} - v_1 d & 0 < e_1 &< \frac{2113}{4096} \times 2^{31} + d \\ e_2 &= 2^{64} - (2^{32} + v_2) d & 0 < e_2 &< 2d \\ e_3 &= 2^{64} - (2^{32} + v_3) d & 0 < e_2 &\le d \end{aligned}$$

Remarks:

- The final step in the algorithm is not a Newton iteration, but an adjustment step which adds zero or one to the reciprocal approximation.
- We gain precision in the first Newton iteration by choosing the initial value v_0 so that the range for the error e_0 is symmetric around zero.
- In the Newton iteration $x + x(1 - xd)$, there is cancellation in the subtraction $(1 - xd)$, since xd is close to 1. In Alg. 2 and 3 we arrange so that the errors e_k, for $k \ge 1$, are non-negative, and exploit that a certain number of the high bits of $v_k d$ are know a-priori to be all ones.
- The execution time of Alg. 2 is roughly 48 cycles on AMD Opteron, and 70 cycles on Intel Core 2.

$(q, r) \leftarrow$ DIV2BY1$(\langle u_1, u_0 \rangle, d, v)$
In: $\beta/2 \le d < \beta$, $u_1 < d$, $v = \lfloor (\beta^2 - 1)/d \rfloor - \beta$
1 $\langle q_1, q_0 \rangle \leftarrow v u_1$ // `umul`
2 $\langle q_1, q_0 \rangle \leftarrow \langle q_1, q_0 \rangle + \langle u_1, u_0 \rangle$
3 $q_1 \leftarrow (q_1 + 1) \bmod \beta$
4 $r \leftarrow (u_0 - q_1 d) \bmod \beta$ // `umullo`
5 **if** $r > q_0$ // Unpredictable condition
6 $q_1 \leftarrow (q_1 - 1) \bmod \beta$
7 $r \leftarrow (r + d) \bmod \beta$
8 **if** $r \ge d$ // Unlikely condition
9 $q_1 \leftarrow q_1 + 1$
10 $r \leftarrow r - d$
11 **return** q_1, r

Algorithm 4: New algorithm for dividing a two-word number by a single-word number, using a precomputed single-word reciprocal.

B. Dividing a two-word number by a single word

To improve performance of division, it would be nice if we could get away with using `umullo` for the multiplication $q'd$ in Alg. 1 (line 2), rather than a full `umul`. Then the candidate remainder $U - q'd$ will be computed only modulo β, even though the full range of possible values is too large to be represented by a single word. We will need some additional information to be able to make a correct adjustment. It turns out that this is possible, if we take the fractional part of the quotient approximation into account. Intuitively, we expect the candidate remainder to be roughly proportional to the quotient fraction.

Our new and improved method is given in Alg. 4. It is based on the following theorem.

Theorem 2: Assume $\beta/2 \le d < \beta$, $0 \le u_1 < d$, and $0 \le u_0 < \beta$. Put $v = \lfloor (\beta^2 - 1)/d \rfloor - \beta$. Form the two-word number

$$\langle q_1, q_0 \rangle = (\beta + v)\, u_1 + u_0.$$

Form the candidate quotient and remainder

$$\begin{aligned} \widetilde{q} &= q_1 + 1 \\ \widetilde{r} &= \langle u_1, u_0 \rangle - \widetilde{q} d. \end{aligned}$$

Then $\widetilde{r}$ satisfies

$$\max(\beta - d, q_0 + 1) - \beta \le \widetilde{r} < \max(\beta - d, q_0)$$

Hence $\widetilde{r}$ is uniquely determined given $\widetilde{r} \bmod \beta$, d and q_0.

Proof: We have $(\beta + v)\, d = \beta^2 - k$, where $1 \le k \le d$. Substitution in the expression for $\widetilde{r}$ gives

$$\widetilde{r} = u_1 \beta + u_0 - q_1 d - d = \frac{u_1 k + u_0(\beta - d) + q_0 d}{\beta} - d.$$

For the lower bound, we clearly have

$$\widetilde{r} \geq \frac{q_0 d}{\beta} - d.$$

This bound implies that both these inequalities hold:

$$\widetilde{r} \geq -d$$
$$\widetilde{r} \geq (q_0 - \beta)\frac{d}{\beta} > q_0 - \beta.$$

The desired lower bound on $\widetilde{r}$ now follows.

For the upper bound, we have

$$\begin{aligned}\widetilde{r} &< \frac{d^2 + \beta(\beta - d) + q_0 d}{\beta} - d \\ &= \frac{\beta - d}{\beta}(\beta - d) + \frac{d}{\beta} q_0 \leq \max(\beta - d, q_0)\end{aligned}$$

where the final inequality follows from recognising the expression as a convex combination. ∎

Remark: The lower bound for $\tilde{r}$ is attained if and only if $u_0 = u_1 = 0$. Then $q_1 = q_0 = 0$, and $\widetilde{r} = -d$. The upper bound is attained if and only if $u_0 = u_1 = \beta - 1$ and $d = \beta/2$. Then $v = \beta - 1$, $q_1 = \beta - 2$, $q_0 = \beta/2$, and $\widetilde{r} = \beta/2 - 1$.

In Alg. 4, denote the value computed at line 4 by r'. Then $r' = \widetilde{r} \bmod \beta$. A straightforward application of Theorem 2 would compare this value to $\max(\beta - d, q_0)$. In Alg. 4, we instead compare r' to q_0. To see why this gives the correct result, consider two cases:

- Assume $\widetilde{r} \geq 0$. Then $r' = \widetilde{r} < \max(\beta - d, q_0)$. Hence, whenever the condition at line 5 is true, we have $r' < \beta - d$, so that the addition at the next line does not overflow. The second adjustment condition, at line 8, reduces the remainder to the proper range $0 \leq r < d$.
- Otherwise, $\widetilde{r} < 0$. Then $r' = \widetilde{r} + \beta \geq \max(\beta - d, q_0 + 1)$. Since $r' > q_0$, the condition at line 5 is true, and since $r' \geq \beta - d$, the addition $(r' + d) \bmod \beta = r' + d - \beta = \widetilde{r} + d$ yields a correct remainder in the proper range. The condition at line 8 is false.

Of the two adjustment conditions, the first one is inherently unpredictable, with a non-negligible probability for either outcome. This means that branch prediction will not be effective. For good performance, the first adjustment must be implemented in a branch-free fashion, e.g., using a conditional move instructions. The second condition, $r' \geq d$, is true with very low probability (see Appendix A for analysis of this probability), and can be handled by a predicated branch or using conditional move.

IV. Extensions for schoolbook division

The key idea in Alg. 4 can be applied to other small divisions, not just two-word divided by single word (which we call a "2/1" division). This leads to a family of algorithms, all which compute a quotient approximation by multiplication by a precomputed reciprocal, then omit computing the high, almost cancelling, part of the corresponding candidate remainder, and finally, they perform an adjustment step using a fraction associated with the quotient approximation.

We will focus on extensions that are useful for schoolbook division with a large divisor. The most important extension is 3/2-division, i.e., dividing a three-word number by a two-word number. This is described next. Later on in this section, we will also look into variations that produce more than one quotient word.

$(q, \langle r_1, r_0\rangle) \leftarrow$ DIV3BY2$(\langle u_2, u_1, u_0\rangle, \langle d_1, d_0\rangle, v)$
In: $\beta/2 \leq d_1 < \beta$, $\langle u_2, u_1\rangle < \langle d_1, d_0\rangle$, $v = \lfloor(\beta^2 - 1)/d\rfloor - \beta$

1 $\langle q_1, q_0\rangle \leftarrow v u_2$ // umul
2 $\langle q_1, q_0\rangle \leftarrow \langle q_1, q_0\rangle + \langle u_2, u_1\rangle$
3 $r_1 \leftarrow (u_1 - q_1 d_1) \bmod \beta$ // umullo
4 $\langle t_1, t_0\rangle \leftarrow d_0 q_1$ // umul
5 $\langle r_1, r_0\rangle \leftarrow (\langle r_1, u_0\rangle - \langle t_1, t_0\rangle - \langle d_1, d_0\rangle) \bmod \beta^2$
6 $q_1 \leftarrow (q_1 + 1) \bmod \beta$
7 **if** $r_1 \geq q_0$
8 $\quad q_1 \leftarrow (q_1 - 1) \bmod \beta$
9 $\quad \langle r_1, r_0\rangle \leftarrow (\langle r_1, r_0\rangle + \langle d_1, d_0\rangle) \bmod \beta^2$
10 **if** $\langle r_1, r_0\rangle \geq \langle d_1, d_0\rangle$ // Unlikely condition
11 $\quad q_1 \leftarrow q_1 + 1$
12 $\quad \langle r_1, r_0\rangle \leftarrow \langle r_1, r_0\rangle - \langle d_1, d_0\rangle$
13 **return** $q_1, \langle r_1, r_0\rangle$

Algorithm 5: Dividing a three-word number by a two-word number, using a precomputed single-word reciprocal.

A. Dividing a three-word number by a two-word number

For schoolbook division with a large divisor, the simplest method is to compute one quotient word at a time by dividing the most significant two words of the dividend by the single most significant word of the divisor, which is a direct application of Alg. 4. Assuming the divisor is normalised, the resulting quotient approximation is at most two units too large. Next, the corresponding remainder candidate is computed and adjusted if necessary. A drawback with this method is that the probability of adjustment is significant, and that each adjustment has to do an addition or a subtraction of large numbers. To improve performance, it is preferable to compute a quotient approximation based on one more word of both dividend and divisor, three words divided by two words. With a normalised divisor, the quotient approximation is at most one off, and the probability of error is small. For more details on the schoolbook division algorithm, see [3, Sec. 4.3.1, Alg. D] and [4].

We therefore consider the following problem: Divide $\langle u_2, u_1, u_0\rangle$ by $\langle d_1, d_0\rangle$, computing the quotient q and remainder $\langle r_1, r_0\rangle$. To ensure that q fits in a single word, we assume that $\langle u_2, u_1\rangle < \langle d_1, d_0\rangle$, and like for 2/1 division, we also assume that the divisor is normalised, $d_1 \geq \beta/2$.

Algorithm 5 is a new algorithm for 3/2 division. The adjustment condition at line 7 is inherently unpredictable, and should therefore be implemented in a branch-free fashion, while the second one, at line 10, is true with very low probability. The algorithm is similar in spirit to Alg. 4. The correctness of the algorithm follows from the following theorem.

Theorem 3: Consider the division of the three-word number $U = \langle u_2, u_1, u_0\rangle$ by the two-word number $D = \langle d_1, d_0\rangle$.

Assume that $\beta/2 \le d_1 < \beta$ and $\langle u_2, u_1\rangle < \langle d_1, d_0\rangle$ Put

$$v = \left\lfloor \frac{\beta^3 - 1}{D} \right\rfloor - \beta$$

which is in the range $0 \le v < \beta$. Form the two-word number

$$\langle q_1, q_0\rangle = (\beta + v)u_2 + u_1.$$

Form the candidate quotient and remainder

$$\widetilde{q} = q_1 + 1$$
$$\widetilde{r} = \langle u_2, u_1, u_0\rangle - \widetilde{q}\,\langle d_1, d_0\rangle.$$

Then $\widetilde{r}$ satisfies

$$c - \beta^2 \le \widetilde{r} < c$$

with

$$c = \max(\beta^2 - D, q_0\beta).$$

Proof: We have $(\beta + v)\, D = \beta^3 - K$, for some K in the range $1 \le K \le D$. Substitution gives

$$\begin{aligned}\widetilde{r} &= U - \widetilde{q}D \\ &= \frac{u_2K + u_1(\beta^2 - D) + u_0\beta + q_0D}{\beta} - D.\end{aligned}$$

The lower bounds $\widetilde{r} \ge -D$ and $\widetilde{r} > q_0\beta - \beta^2$ follow in the same way as in the proof of Theorem 2, proving the lower bound $\widetilde{r} \ge c - \beta^2$. For the upper bound, the borderline cases make the proof more involved. We need to consider several cases.

- If $u_2 \le d_1 - 1$, then
$$\begin{aligned}\widetilde{r} &< \frac{(d_1 - 1)D + (\beta - 1)(\beta^2 - D) + \beta^2 - \beta D + q_0D}{\beta} \\ &= \frac{(\beta^2 - D)^2 + q_0\beta D - d_0D}{\beta^2} \\ &= \frac{\beta^2 - D}{\beta^2}(\beta^2 - D) + \frac{D}{\beta^2}q_0\beta - \frac{d_0D}{\beta^2} \\ &\le c.\end{aligned}$$
- If $u_2 = d_1$, then $u_1 \le d_0 - 1$, by assumption. In this case, we get
$$\begin{aligned}\widetilde{r} &< \frac{d_1D + (d_0 - 1)(\beta^2 - D) + \beta^2 - \beta D + q_0D}{\beta} \\ &= \frac{\beta^2 - D}{\beta^2}(\beta^2 - D) + \frac{D}{\beta^2}q_0\beta \\ &\quad + \frac{(\beta - d_0)\left((\beta + 1)D - \beta^3\right)}{\beta^2} \\ &\le c + \frac{(\beta - d_0)\left((\beta + 1)D - \beta^3\right)}{\beta^2}.\end{aligned}$$
Under the additional assumption that $D \le \beta(\beta - 1)$, we get $(\beta + 1)D - \beta^3 \le -\beta < 0$, and it follows that $\widetilde{r} < c$.
- Finally, the remaining borderline case is $u_2 = d_1$ and $D > \beta(\beta - 1)$. We then have $u_2 = d_1 = \beta - 1$, $0 \le u_1 < d_0$, and $v = 0$ since $(\beta^3 - 1)/D - \beta < 1$. It follows that $q_1 = u_2 = \beta - 1$. We get
$$\widetilde{r} = u - \beta D = \beta\,(u_1 - d_0) + u_0 < 0 < c.$$
Hence the upper bound $\widetilde{r} < c$ is valid in all cases.

■

$v \leftarrow$ RECIPROCAL_WORD_3BY2($\langle d_1, d_0\rangle$)
In: $\beta/2 \le d_1 < \beta$
1 $v \leftarrow$ RECIPROCAL_WORD(d_1)
// We have $\beta^2 - d_1 \le (\beta + v)\, d_1 < \beta^2$.
2 $p \leftarrow d_1 v \bmod \beta$ // umullo
3 $p \leftarrow (p + d_0) \bmod \beta$
4 **if** $p < d_0$ // Equivalent to carry out
5 $v \leftarrow v - 1$
6 **if** $p \ge d_1$
7 $v \leftarrow v - 1$
8 $p \leftarrow p - d_1$
9 $p \leftarrow (p - d_1) \bmod \beta$
// We have $\beta^2 - d_1 \le (\beta + v)\, d_1 + d_0 < \beta^2$.
10 $\langle t_1, t_0\rangle \leftarrow vd_0$ // umul
11 $p \leftarrow (p + t_1) \bmod \beta$
12 **if** $p < t_1$ // Equivalent to carry out
13 $v \leftarrow v - 1$
14 **if** $\langle p, t_0\rangle \ge \langle d_1, d_0\rangle$
15 $v \leftarrow v - 1$
16 **return** v

Algorithm 6: Computing the reciprocal which DIV3BY2 expects, $v = \lfloor(\beta^3 - 1)/\langle d_1, d_0\rangle\rfloor - \beta$. This is a single word reciprocal based on a two-word divisor.

B. Computing the reciprocal for 3/2 division

The reciprocal needed by Alg. 5, even though still a single word, is slightly different from the reciprocal that is needed by Alg 4. One can use Alg. 2 or Alg. 3 (depending on word size) to compute the reciprocal of the most significant word d_1, followed by a couple of adjustment steps to take into account the least significant word d_0. We suggest the following strategy:

Start with the initial reciprocal v, based on d_1 only, and the corresponding product $(\beta + v)\, d_1\beta$, where only the middle word is represented explicitly (the high word is $\beta - 1$, and the low word is zero). We then add first βd_0 and then vd_0 to this product. For each addition, if we get a carry out, we cancel that carry by appropriate subtractions of d_1 and d_0 to get an underflow. The details are given in Alg. 6.

Remark: The product $d_1 v \bmod \beta$, computed in line 2, may be available cheaply, without multiplication, from the intermediate values used in the final adjustment step of RECIPROCAL_WORD (Alg. 2 or Alg. 3).

C. Larger quotients

The basic algorithms for 2/1 division and 3/2 division can easily be extended in two ways.

- One can substitute double-words or other fixed-size units for the single words in Alg. 4 and Alg. 5. This way, one can construct efficient algorithms that produce quotients of two or more words. E.g., with double-word units, we get algorithms for division of sizes 4/2 and 6/4.
- In any of the algorithms constructed as above, one can fix one or more of the least significant words of both

$(Q, r) \leftarrow$ DIV_NBY1(U, d)
  In: $U = \langle u_{n-1} \dots u_0 \rangle$, $\beta/2 \le d < \beta$
  Out: $Q = \langle q_{n-1} \dots q_0 \rangle$
1 $v \leftarrow$ RECIPROCAL_WORD(d)
2 $r \leftarrow 0$
3 **for** $j = n-1, \dots, 0$
4 $(q_j, r) \leftarrow$ DIV2BY1$(\langle r, u_j \rangle, d, v)$
5 **return** Q, r

Algorithm 7: Dividing a large integer $U = \langle u_{n-1} \dots u_0 \rangle$ by a normalised single-word integer d.

dividend and divisor to zero. This gives us algorithms for division of sizes such as 3/1 and 5/3 (and applying this procedure to 3/2 would recover the good old 2/1 division).

Details and applications for some of these variants are described in [4].

V. Case study: x86_64 implementation of $n/1$ division

Schoolbook division is the main application of 3/2 division, as was described briefly in the previous section. We now turn to a more direct application of 2/1 division using Alg. 4.

In this section, we describe our implementation of DIV_NBY1, dividing a large number by a single word number, for current processors in the x86_64 family. We use conditional move (`cmov`) to avoid branches that are difficult to handle efficiently by branch-prediction. Besides `cmov`, the most crucial instructions used are `mul`, `imul`, `add`, `adc`, `sub` and `lea`. Detailed latency and throughput measurements of these instructions, for 32-bit and 64-bit processors in the x86 family, are given in [5].

We discuss the timing only for AMD Opteron ("K8/K9") and Intel Core 2 (65 nm "Conroe") in this section. The AMD Opteron results are valid also for processors with the brand names Athlon and Phenom[2]. Other recent Intel processors give results slightly different from the 65 nm Core 2 results we describe[3].

Our results focus mainly on AMD chips since they are better optimised for scientific integer operations, i.e., the ones we depend on. If we don't specify host architecture, we are talking about AMD Opteron.

A. Dividing a large integer by a single word

Consider division of an n-word number U by a single word number d. The result of the division is an n-word quotient and a single-word remainder. This can be implemented by repeatedly replacing the two most significant words of U by their single-word remainder modulo d, and recording the corresponding quotient word [3, Sec. 4.3.1, exercise 16]. The variant shown in Alg. 7 computes a reciprocal of d (and hence requires that d is normalised), and applies our new 2/1 division algorithm in each step.

To use Alg. 7 directly, d must be normalised. To also handle unnormalised divisors, we select a shift count k such that $\beta/2 \le 2^k d < \beta$. Alg. 7 can then be applied to the shifted operands $2^k U$ and $2^k d$. The quotient is unchanged by this transformation, while the resulting remainder has to be shifted k bits right at the end. Shifting of U can be done on the fly in the main loop. In the code examples, register `cl` holds the normalisation shift count k.

```
loop:  mov  (np, un, 8), %rax
       div  d
       mov  %rax, (qp, un, 8)
       dec  un
       jnz  loop
```

Example 1: Basic division loop using the `div` instruction, running at 71 cycles per iteration on AMD Opteron, and 116 cycles on Intel Core 2. Note that `rax` and `rdx` are implicit input and output arguments to the `div` instruction.

B. Naïve implementation

The main loop of an implementation in x86_64 assembler is shown in Example. 1. Note that the `div` instruction in the x86 family appear to be tailor-made for this loop: This instructions takes a divisor as the explicit argument. The two-word input dividend is placed with the most significant word in the `rdx` register and the least significant word in the `rax` register. The output quotient is produced in `rax` and the remainder in `rdx`. No other instruction in the loop need to touch `rdx` as the remainder is produced by each iteration and consumed in the next.

However, the dependency between iterations, via the remainder in `rdx`, means that the execution time is lower bounded by the latency of the `div` instruction, which is 71 cycles on AMD Opteron [5] (and even longer, 116 cycles, on Intel Core 2). Thanks to parallelism and out-of-order execution, the rest of the instructions are executed while waiting for the result from the division. This loop is more than an order of magnitude slower than the loop for multiplying a large number by a single-word number.

C. Old division method

The earlier division method from [1] can be implemented with the main loop in Example 2. The dependency between operations, via the `rax` register, is still crucial to understand the performance. Consider the sequence of dependent instructions in the loop, from the first use of `rax` until the output value of the iteration is produced. This is what we call the *recurrency chain* of the loop. The assembler listing is annotated with cycle numbers, for AMD Opteron and Intel Core 2. We let cycle 0 be the cycle when the first instructions on the recurrency chain starts executing, and the following instructions in the chain are annotated with the cycle number of the earliest cycle the

[2] Phenom has the same multiplication latencies, but slightly higher(!) latency for division.

[3] The 45 nm Core 2 has somewhat lower division latency, and the same multiplication latencies. The Core ix processors ($x = 3, 5, 7, 9$) have lower division latency, and for `umul`, they have lower latency for the low product word, but higher(!) latency for the high product word.

```
                loop:  mov     (up,un,8), %rdx
                       shld    %cl, %rdx, %r14
                       lea     (d,%r14), %r12
                       bt      $63, %r14
                       cmovnc  %r14, %r12
   0      0            mov     %rax, %r10
   0      0            adc     $0, %rax
   1      2            mul     dinv
   5     10            add     %r12, %rax
                       mov     d, %rax
   6     11            adc     %r10, %rdx
   7     13            not     %rdx
   8     14            mov     %rdx, %r12
   8     14            mul     %rdx
  12     22            add     %rax, %r14
  13     23            adc     %rdx, %r10
  14     25            sub     d, %r10
  13     23            lea     (d,%r14), %rax
  14     26            cmovnc  %r14, %rax
 AMD   Intel           sub     %r12, %r10
                       mov     (up,un,8), %r14
                       mov     %r10, 8(qp,un,8)
                       dec     un
                       jnz     loop
```

Example 2: Previous method using a precomputed reciprocal, running at 17 cycles per iteration on AMD Opteron, and 32 cycles on Intel Core 2.

```
                loop:  nop
                       mov     (up,un,8), %r10
                       lea     1(%rax), %r11
   0      0            shld    %cl, %r10, %rbp
                       mul     dinv
   0      0            add     %rbp, %rax
   4      8            adc     %r11, %rdx
   5      9            mov     %rax, %r11
                       mov     %rdx, %r13
                       imul    d, %rdx
   6     11            sub     %rdx, %rbp
  10     16            mov     d, %rax
                       add     %rbp, %rax
  11     17            cmp     %r11, %rbp
  11     17            cmovb   %rbp, %rax
  12     18            adc     $-1, %r13
 AMD   Intel           cmp     d, %rax
                       jae     fix
                ok:    mov     %r13, (qp)
                       sub     $8, qp
                       dec     un
                       mov     %r10, %rbp
                       jnz     loop
                       jmp     done

                fix:   sub     d, %rax
                       inc     %r13
                       jmp     ok
                done:
```

Example 3: Division code (from GMP-4.3) with the new division method, based on Alg. 4. Running at 13 cycles per iteration on AMD Opteron, and 25 cycles on Intel Core 2.

instruction can start executing, taking its input dependencies into account.

To create the annotations, one needs to know the latencies of the instructions. Most arithmetic instructions, including `cmov` and `lea` have a latency of one cycle. The crucial `mul` instruction has a latency of four cycles until the low word of the product is available in `rax`, and one more cycle until the high word is available in `rdx`. The `imul` instructions, which produces the low half only, also has a latency of four cycles. These numbers are for AMD, the latencies are slightly longer on Intel Core 2 (2 cycles for `adc` and `cmov`, 5 cycles for `imul` and 8 for `mul`). See [5] for extensive empirical timing data.

Using these latency figures, we find that the latency of the recurrency chain in Example 2 is 15 cycles. This is a lower bound on the execution time. It turns out that the loop runs in 17 cycles per iteration; the instructions not on the recurrency chain are mostly scheduled for execution in parallel with the recurrency instructions, and there's plenty of time, 8 cycles, when the CPU is otherwise just waiting for the results from the multiplication unit. This is a four time speedup compared to the 71-cycle loop based on the `div` instruction. For Intel Core 2, the latency of the recurrency chain is 28 cycles, while the actual running time is 32 cycles per iteration.

D. New division method

The main loop of an implementation of the new division method is given in Example 3. Annotating the listing with cycle numbers in the same way, we see that the latency of the recurrency chain is 13 cycles. Note that the rarely taken branch does not belong to the recurrency chain. The loop actually also runs at 13 cycles per iteration; all the remaining instructions are scheduled for execution in parallel with the recurrency chain[4]. For Intel Core 2, the latency of the recurrency chain is 20 cycles, with an actual running time of 25 cycles per iteration.

Comparing the old and the new method, first make the assumption (which is conservative in the Opteron case) that all the loops can be tuned to get their running times down to the respective latency bounds. We then get a speedup of 15% on AMD Opteron and 40% on Intel Core 2. If we instead compare actual cycle counts, we see a speedup of 31% on both Opteron and Core 2. On Opteron, we gain one cycle from replacing one of the `mul` instructions by the faster `imul`, the other cycle shaved off the recurrency chain are due to the simpler adjustment conditions.

In this application, the code runs slower on Intel Core 2 than on AMD Opteron. The Intel CPU loses some cycles due

[4] It's curious that if the `nop` instruction at the top of the loop is removed, the loop runs one cycle slower. It seems likely that similar random changes to the instruction sequence in Example 2 can reduce its running time by one or even two cycles, to reach the lower bound of 15 cycles.

Implementation	Recurrency chain latency and real cycle counts			
	AMD Opteron		Intel Core 2	
Naïve `div` loop (Ex. 1)	71	71	116	116
Old method (Ex. 2)	15	17	28	32
New method (Ex. 3)	13	13	20	25

TABLE I
SUMMARY OF THE LATENCY OF THE RECURRENCY CHAIN, AND ACTUAL CYCLE COUNTS, FOR TWO X86_64 PROCESSORS. THE LATENCY NUMBERS ARE LOWER BOUNDS FOR THE ACTUAL CYCLE COUNTS.

to higher latencies for multiplication and carry propagation, resulting in a higher overall latency of the recurrency chain. And then it loses some additional cycles due to the fact that the code was written and scheduled with Opteron in mind.

VI. CONCLUSIONS

We have described and analysed a new algorithm for dividing a two-word number by a single-word number ("2/1" division). The key idea is that when computing a candidate remainder where the most significant word almost cancels, we omit computing the most significant word. To enable correct adjustment of the quotient and the remainder, we work with a slightly more precise quotient approximation than in previous algorithms, and an associated fractional word.

Like previous methods, we compute the quotient via an approximate reciprocal of the divisor. We describe new, more efficient, algorithms for computing this reciprocal for the most common cases of a word size of 32 or 64 bits.

The new algorithm for 2/1 division directly gives a speedup of roughly 30% on current processors in the x86_64 family, for the application of dividing a large integer by a single word. It is curious that on these processors, the combination of our reciprocal algorithm (Alg. 2) and division algorithm (Alg. 4) is significantly faster than the built in assembler instruction for 2/1 division. This indicates that the algorithms may be of interest for implementation in CPU microcode.

We have also described a couple of extensions of the basic algorithm, primarily to enable more efficient schoolbook division with a large divisor.

Most of the algorithms we describe have been implemented in the GMP library [2].

ACKNOWLEDGEMENTS

The authors wish to thank Stephan Tolksdorf, Björn Terelius, David Harvey and Johan Håstad for valuable feedback on draft versions of this paper. As always, the responsibility for any remaining errors stays with the authors.

REFERENCES

[1] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," in *Proceedings of the SIGPLAN PLDI'94 Conference*, June 1994.
[2] T. Granlund, "GNU multiple precision arithmetic library, version 4.3," May 2009, http://gmplib.org/.
[3] D. E. Knuth, *Seminumerical Algorithms*, 3rd ed., ser. The Art of Computer Programming. Reading, Massachusetts: Addison-Wesley, 1998, vol. 2.
[4] T. Granlund and N. Möller, "Division of integers large and small," August 2009, to appear.
[5] T. Granlund, "Instruction latencies and throughput for AMD and Intel x86 processors," 2009, http://gmplib.org/~tege/x86-timing.pdf.

APPENDIX A
PROBABILITY OF THE SECOND ADJUSTMENT STEP

In this appendix, we analyse the probability of the second adjustment step (line 8 in Alg. 4), and substantiate our claim that the second adjustment is unlikely. We use the notation from Sec. III-B. We also use the notation that $\mathrm{P}[\text{event}]$ is the probability of a given event, and $\mathrm{E}[X]$ is the expected value of a random variable X.

We will treat $\widetilde{r}$ as a random variable, but we first need to investigate for which values of $\widetilde{r}$ that the second adjustment step is done. There are two cases:

- If $\widetilde{r} \geq d$, then $\widetilde{r} < \max(\beta - d, q_0)$ and $d \geq \beta - d$ imply that $\widetilde{r} < q_0$. The first adjustment is skipped, the second is done.
- If $\widetilde{r} > q_0$, then $\widetilde{r} < \max(\beta - d, q_0)$ implies that $\widetilde{r} < \beta - d$ and $d \leq \widetilde{r} + d < \beta$. The first adjustment is done, then undone by the second adjustment.

The inequalities $\widetilde{r} \geq d$ and $\widetilde{r} \geq q_0$ are thus mutually exclusive, the former possible only when $q_0 > d$ and the latter possible only when $q_0 < \beta - d$.

One example of each kind, for $\beta = 2^5 = 32$:

U	d	q	r	v	k	$\widetilde{q}$	q_0	$\widetilde{r}$
414	18	23	0	24	16	22	30	18
504	18	28	0	24	16	28	0	0

To find the probabilities, in this section, we treat $\widetilde{r}$ as a random variable. Consider the expression for $\widetilde{r}$,

$$\widetilde{r} = \frac{u_1 k + u_0(\beta - d) + q_0 d}{\beta} - d.$$

We assume we have a fixed $d = \xi\beta$, with $1/2 \leq \xi < 1$, and consider u_1 and u_0 as independent uniformly distributed random variables in the ranges $0 \leq u_1 < d$ and $0 \leq u_0 < \beta$. We also make the simplifying assumptions that k and q_0 are independent and uniformly distributed, in the ranges $0 < k \leq d$ and $0 \leq q_0 < \beta$, and that all these variables are *continuous* rather than integer-valued.[5]

Lemma 4: Assume that $1/2 \leq \xi < 1$, that u_1, u_0, k and q_0 are independent random variables, continuously and uniformly distributed with ranges $0 \leq u_1, k \leq \xi\beta$, $0 \leq u_0, q_0 \leq \beta$. Let

$$\widetilde{r} = \frac{u_1 k + u_0(1 - \xi)\beta + q_0 \xi \beta}{\beta} - \xi\beta.$$

Then

$$\begin{aligned}\mathrm{P}[\widetilde{r} \geq \xi\beta \text{ or } \widetilde{r} \geq q_0] &= \frac{(2 - 1/\xi)^3}{6(1-\xi)^2} \log \frac{2 - 1/\xi}{\xi} + \frac{1}{6} \\ &\quad + (1 - \xi)\left(-\frac{1}{18} + \frac{1}{2\xi} - \frac{11}{12\xi^2} + \frac{11}{36\xi^3}\right) \quad (11)\end{aligned}$$

[5]These assumptions are justified for large word-size. Strictly speaking, with fixed d, the variable k is of course not random at all. To make this argument strict, we would have to treat d as a random variable with values in a small range around $\xi\beta$, e.g., uniformly distributed in the range $\xi\beta \pm \beta^{3/4}$, and consider the limit as $\beta \to \infty$. Then the modulo operations involved in q_0 and k make these variables behave as almost independent and uniformly distributed.

Furthermore, if we define

$$f(\xi) = 1 - \frac{297}{64}(1-\xi) + \frac{15}{2}(1-\xi)^2 - \frac{17}{4}(1-\xi)^3$$

then

$$\mathrm{P}[\widetilde{r} \geq \xi\beta \text{ or } \widetilde{r} \geq q_0] \approx \frac{(1-\xi)^6}{24 f(\xi)} \tag{12}$$

with an absolute error less than 0.01 percentage points, and a relative error less than 5%.

Proof: Define the stochastic variables

$$X = \frac{u_1 k}{\xi\beta^2} \qquad R = \frac{u_1 k + u_0(1-\xi)\beta}{\xi\beta^2} \qquad Q = \frac{q_0}{\beta}.$$

Now,

$$\frac{\widetilde{r}}{\xi\beta} = R + Q - 1.$$

By assumption, Q is uniformly distributed, while R has a more complicated distribution. Conditioning on $Q = s$, we get the probabilities

$$\begin{aligned}
\mathrm{P}[\widetilde{r} \geq \xi\beta] &= \int_{3-\xi-1/\xi}^{1} \mathrm{P}[R \geq 2 - s]\,\mathrm{d}s \\
&= \int_{0}^{\xi+1/\xi-2} \mathrm{P}[R \geq 1 + s]\,\mathrm{d}s \\
\mathrm{P}[\widetilde{r} \geq q_0] &= \int_{0}^{1-\xi} \mathrm{P}[R \geq 1 + (1/\xi - 1)s]\,\mathrm{d}s \\
&= \frac{1}{1/\xi - 1} \int_{0}^{\xi+1/\xi-2} \mathrm{P}[R \geq 1 + s]\,\mathrm{d}s.
\end{aligned}$$

Adding the probabilities (recall that the events are mutually exclusive), we get the probability of adjustment as

$$\frac{1}{1-\xi} \int_{0}^{\xi+1/\xi-2} \mathrm{P}[R \geq 1 + s]\,\mathrm{d}s. \tag{13}$$

We next need the probabilities $\mathrm{P}[R \geq s]$ for $1 \leq s \leq \xi + 1/\xi - 1$. By somewhat tedious calculations, we find

$$\begin{aligned}
\mathrm{P}\left[X \leq s\right] &= \frac{\beta s}{d}\left(1 - \log\frac{\beta s}{d}\right) \\
\mathrm{P}[R \geq s] &= \frac{\xi}{1-\xi}\,\mathrm{E}[\max(0, X - (s - (1/\xi - 1)))] \\
&= -\frac{(s + 1 - 1/\xi)^2}{2(1-\xi)} \log\frac{s + 1 - 1/\xi}{\xi} \\
&\quad + \frac{\xi^2 - 4(s + 1 - 1/\xi) + 3(s + 1 - 1/\xi)^2}{4(1-\xi)},
\end{aligned}$$

where the latter equation is valid only for s in the interval of interest. Substituting in Eq. (13) and integrating yields Eq. (11). To approximate this complicated expression, we first derive its asymptotics:

$$(1-\xi)^6/24 + O\left((1-\xi)^7\right)$$

for ξ close to 1, and

$$\begin{aligned}
1/36 - 13/18(\xi - 1/2) + 34/3(\xi - 1/2)^2 \\
+ O\left((\xi - 1/2)^3 \log(\xi - 1/2)\right)
\end{aligned}$$

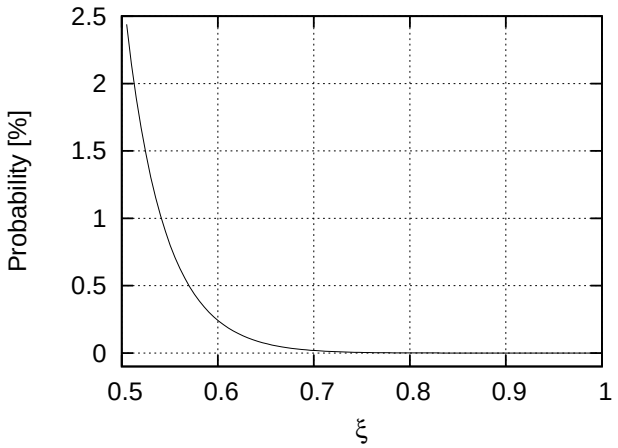

Fig. 1. Probability of the unlikely adjustment step, as a function of the ratio $\xi = d/\beta$.

for ξ close to $1/2$. The coefficients of f are chosen to give the same asymptotics. The error bounds for Eq. (12) are found numerically. ■

In Fig. 1, the adjustment probability of Eq. (11) is plotted as a function of the ratio $\xi = d/\beta$. This is a rapidly decreasing function, with maximum value for $\xi = 1/2$, which gives the worst case probability of $1/36$ for d close to $\beta/2$. This curve is based on the assumptions on continuity and independence of the random variables. For a fixed d and word size, the adjustment probability for random u_1 and u_0 will deviate some from this continuous curve. In particular, the borderline case $d = \beta/2$ actually gives an adjustment probability of zero, so it is not the worst case.