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. (FIXME: Mention win even over a single division instruction with no invariance.)**

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. 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. (But somewhat surprisingly, a well-tuned Newton reciprocal followed by multiplication and adjustments wins over the designated 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 an 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.

- Previous methods need one `umulhi` and one full `umul`. The new method uses one `umul` and one `umullo`, and `umullo` is a cheaper operation (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 algorithm has 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, computed, and used. Section III gives the main result, a new method for dividing a two-word number by a single word, together with correctness proof and analysis of the probability for the adjustment steps. 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 concludes and discusses some open problems.

A. Notation and conventions

Let ℓ denote the computer word size, and let $\beta = 2^\ell$ denote the base implied by the word size, and upper-case letters represent numbers of any size. We use the notation $X = \langle x_{n-1}, x_{n-2}, \ldots, x_0\rangle = x_{n-1}\beta^{n-1} + x_{n-2}\beta^{n-2} + \cdots + x_0$, where the n-word integer X is represented by the words x_i.

N. Möller is with **(FIXME: ???)**. Email: `nisse@lysator.liu.se`
T. Granlund is with the School of Computer Science and Communication, KTH, Stockholm. Email: `tege@nada.kth.se`

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`. **(FIXME: Comment on poor multiplication on sparc?)**

In our probability analysis, we 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.

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

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

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 will also restrict attention the case that d is a "normalised" single-word number, i.e., $\beta/2 \leq d < \beta$. This means that the word d has 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 $1/\beta < 1/d \leq 2/\beta$. We represent this using a fixed-point representation, with a single word v and an additional implicit one bit at the most significant end,

$$\frac{1}{d} \approx \frac{\beta + v}{\beta^2}. \tag{1}$$

For the border case $d = \beta/2$, we force the value to fit in this representation by using a reciprocal of $2/\beta - 1/\beta^2$ rather than the true reciprocal $2/\beta$.

We define the precomputed reciprocal of d as the integer

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

The constraints on d imply that $0 < v < \beta$, in particular, v is a single word number. The usefulness of v comes from Eq. (1) which implies

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

Since $(\beta + v)/\beta^2 < 1/d$, the expression on the right hand side is upper bounded by q and hence also by β. Since the terms on the right hand side are non-negative, the expression is still upper bounded by q if some of the terms are omitted or truncated.

```
(q, r) ← DIV2BY1PI1(⟨u_1, u_0⟩, d, v)
1  $q \leftarrow \lfloor v u_1/\beta \rfloor + u_1$          ▷ Candidate quotient (umulhi)
2  $\langle p_1, p_0 \rangle \leftarrow qd$                                ▷ umul
3  $\langle r_1, r_0 \rangle \leftarrow \langle u_1, u_0 \rangle - \langle p_1, p_0 \rangle$   ▷ Candidate remainder
4  while $r_1 > 0$ or $r_0 \geq d$     ▷ (Repeated at most 3 times)
5      do $q \leftarrow q + 1$
6          $\langle r_1, r_0 \rangle \leftarrow \langle r_1, r_0 \rangle - d$
7  return $q, r_0$
```

Algorithm 1: Simple division using a precomputed reciprocal.

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. (2).

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

$$\begin{aligned} R' &= U - q'd \\ &= u_0 + u_1\beta - \frac{u_1(\beta + v) - q_0}{\beta} d \\ &= u_0 + \frac{u_1 k + q_0 d}{\beta} \\ &< \beta + 2d. \end{aligned}$$

We see that $R' \geq 0$, which corresponds to $q' \leq q$. The other bound, $R' < \beta + 2d$, implies that $q \leq q' + 3$. 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 $u_0 \geq d$, and if so, increment q' before computing r', one gets $R' < 3d$ and $q' \leq q + 2$. The method in [1], Sec. 8, is more intricate, guaranteeing that $R' < 2d$, so that $q' \leq q + 1$. However, it still computes the full product $q'd$, so this method needs one `umul` and one `umulhi`.

B. 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

$$x_{k+1} = x_k + x_k(1 - x_k d). \tag{3}$$

This equation implies that

$$1 - x_{k+1}d = (1 - x_k d)^2. \tag{4}$$

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. (3). 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 (4), the truncation of the result, and any truncation of the d input.

```
$v \leftarrow$ RECIPROCAL_WORD$(d)$
1  $d_9 \leftarrow \lfloor 2^{-55} d \rfloor$                         ▷ Most significant 9 bits
2  $d_{32} \leftarrow \lfloor 2^{-32} d \rfloor$                      ▷ Most significant 32 bits
3  $v_0 \leftarrow 2^6 \lfloor (2^{18} - 2^8)/d_9 \rfloor$              ▷ By table lookup
4  $v_1 \leftarrow 2^{17} v_0 - \lfloor 2^{-31} v_0^2 d_{32} \rfloor$      ▷ 2 umullo
5  $v_2 \leftarrow 2^{33} v_1 - 2\lfloor 2^{-64} v_1^2 d \rfloor$        ▷ umullo, umulhi
6  $v_3 \leftarrow (4\,v_2 - \lfloor 2^{-126} v_2^2 d \rfloor - 1) \bmod 2^{64}$     ▷ 3 umul
7  $v_4 \leftarrow (v_3 - \lfloor 2^{-64} (v_3 + 2^{64} + 1)\, d \rfloor) \bmod 2^{64}$   ▷ umul
8  return $v_4$
```

Algorithm 2: Reciprocal computation for $\beta = 2^{64}$.

Algorithm 2 gives one variant, for $\beta = 2^{64}$. Here, v_0 is represented as 16 bits, v_1 as 32 bits, v_2 as 64 bits, and v_3 and v_4 as 65-bit values where the most significant bit, which is always one, is implicit.

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:

$$e_0 = 2^{128} - 2^{49} v_0 d \qquad |e_0| < \frac{3}{4} \times 2^{120} \tag{5}$$

$$e_1 = 2^{128} - 2^{33} v_1 d \qquad |e_1| < \frac{5}{8} \times 2^{112} \tag{6}$$

$$e_2 = 2^{128} - 2 v_2 d \qquad |e_2| < \left(\frac{5}{8}\right)^2 \times 2^{96} \tag{7}$$

$$e_3 = 2^{128} - (2^{64} + v_3)\, d \qquad 0 < e_3 < 2d \tag{8}$$

$$e_4 = 2^{128} - (2^{64} + v_4)\, d \qquad 0 < e_4 \le d \tag{9}$$

(FIXME: Skip parts of the proof? Or condense further some other way?) Each step involves a truncation, and we let $0 \le \delta_k < 1$ denote the truncation error in each step. Start with (5). Let $d' = d - 2^{55} d_9$. We have

$$\begin{aligned} v_0 &= 2^6((2^{18} - 2^8)/d_9 - \delta_0) \\ e_0 &= 2^{118} + 2^{110}\delta_0 d_9 - 2^{49} v_0 d'. \end{aligned}$$

From this, we get the upper bound $(3/4) \times 2^{120}$ and lower bound $-(3/4) \times 2^{120}$.

For (6), let $d' = d - 2^{32} d_{32}$. We have

$$\begin{aligned} v_1 &= 2^{17} v_0 - 2^{-63} v_0^2 (d - d') + \delta_2 \\ e_1 &= 2^{-128} e_0^2 + 2^{-30} v_0^2 d d' - 2^{33} \delta_2 d. \end{aligned}$$

```
$v \leftarrow$ RECIPROCAL_WORD$(d)$
1  $d_9 \leftarrow \lfloor 2^{-23} d \rfloor$                        ▷ Most significant 9 bits
2  $v_0 \leftarrow 2^6 \lfloor (2^{18} - 2^8)/d_9 \rfloor$             ▷ By table lookup
3  $v_1 \leftarrow 2^{17} v_0 - 2\lfloor 2^{-32} v_0^2 d \rfloor$        ▷ umullo, umulhi
4  $v_2 \leftarrow (4\,v_1 - \lfloor 2^{-62} v_1^2 d \rfloor - 1) \bmod 2^{32}$     ▷ 3 umul
5  $v_3 \leftarrow (v_2 - \lfloor 2^{-32} (v_2 + 2^{32} + 1)\, d \rfloor) \bmod 2^{32}$   ▷ umul
6  return $v_3$
```

Algorithm 3: Reciprocal computation for $\beta = 2^{32}$.

We get the upper bound $(9217/16384) \times 2^{112} < (5/8) \times 2^{112}$ and lower bound -2^{97}.

For (7), we have

$$\begin{aligned} v_2 &= 2^{33} v_1 - 2(2^{-64} v_1^2 d - \delta_1) \\ e_2 &= 2^{-128} e_1^2 - 4\delta_2 d \end{aligned}$$

and the upper bound $(5/8)^2 \times 2^{96}$ and lower bound -2^{66}.

For (8), put

$$\begin{aligned} v_3' &= 4 v_2 - \lfloor 2^{-126} v_2^2 d \rfloor + \delta_3 - 1 \\ e_3' &= 2^{128} - v_3' d = 2^{-128} e_2^2 + (1 - \delta_3) d. \end{aligned}$$

(We will see in a moment that $v_3' = 2^{64} + v_3$, and hence also $e_3' = e_3$). Then $0 < e_3' < (5/8)^4 \times 2^{64} + d < 2^{62} + d \le 2d$. It remains to show that $2^{64} \le v_3' < 2 \times 2^{64}$. The upper bound follows from $e_3' > 0$. For the border 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

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

Hence, the effect of the adjustment is to increment the reciprocal approximation if and only if $e_3 > d$. The desired bound, Eq. (9), follows. ■

Algorithm 3 is the corresponding algorithm for $\beta = 2^{32}$. The correctness proof is analogous.

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.
- To ensure that the needed adjustment is at most one, the choice of initial value is crucial. The 64-bit algorithm works because $|e_0| < (3/4) \times 2^{120}$, and $(3/4)^8 < 1/2$ (with some margin to accommodate for truncation errors). The correctness of the 32-bit algorithm is tighter, with one squaring less. It works because also $(3/4)^4 < 1/2$ with a sufficient margin.[1]

[1]And unlike the 64-bit variant, exhaustively testing the 32-bit algorithm does not require extreme patience or computing resources.

```
$(q, r) \leftarrow$ DIV2BY1PI1($\langle u_1, u_0\rangle, d, v$)
 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 $r \geq q_0$                                ▷ Unpredictable condition
 6     then $q_1 \leftarrow (q_1 - 1) \bmod \beta$
 7          $r \leftarrow (r + d) \bmod \beta$
 8  if $r \geq d$                                       ▷ Unlikely condition
 9     then $q_1 \leftarrow q_1 + 1$
10          $r \leftarrow r - d$
11  return $q_1, r$
```

Algorithm 4: New division method.

- The algorithm can most likely be improved further. In the Newton iteration $x + x(1 - xd)$, there is cancellation in the subtraction $(1 - xd)$, since the upper part of xd are all one bits. But Alg. 2 computes each iteration as $2x - x^2 d$, and can therefore not exploit this cancellation.
- The execution time of Alg. 2 is roughly 50 cycles on AMD Opteron, and 72 cycles on Intel Core 2.

III. 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 keep the fractional part of the quotient approximation around. Intuitively, we expect the candidate remainder to be roughly proportional to the quotient fraction.

A. A new division algorithm

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

Theorem 2: Assume $\beta/2 \leq d < \beta$, $0 \leq u_1 < d$, and $0 \leq u_0 < \beta$. 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

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

Then $\widetilde{r}$ satisfies

$$c - \beta \leq \widetilde{r} < c$$

with

$$c = \max(\beta - d, q_0).$$

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 $1 \leq k \leq d$. 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 $\widetilde{r} \geq -d$. We also have

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

It follows that

$$\widetilde{r} \geq \max(-d, q_0 - \beta) = c - \beta.$$

For the upper bound, we have

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

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

Remarks:

- The lower bound $\widetilde{r} = c - \beta$ is attained if and only if $U = 0$. Then $q_1 = q_0 = 0$, $c = \beta - d$, and $\widetilde{r} = -d$.
- The upper bound $\widetilde{r} = c - 1$ is attained if and only if $d = \beta/2$, $u_1 = \beta/2 - 1$, and $u_0 = \beta - 1$. Then $v = \beta - 1$, $q_1 = \beta - 2$, $q_0 = c = \beta/2$, 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 c. In Alg. 4, we avoid computing c explicitly, and instead compare r' to q_0. To see why this still gives the correct result, consider two cases:

- Assume $\widetilde{r} \geq 0$. Then $r' = \widetilde{r} < c$. 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 $r' = \widetilde{r} + \beta \geq c$. Since $r' \geq q_0$, the condition at line 5 is true, and since $r' \geq \beta - d$, the addition overflows. After the first adjustment, we have a remainder in the proper range. The condition at line 8 is false.

Of the two adjustment conditions, the first one is inherently unpredictable. For random inputs, the probability is roughly 50% 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 conditional move instructions. The second condition, $r' \geq d$, is true with very low probability, and can be handled by a predicated branch or using conditional move.

B. Probability of second adjustment step

In this section, 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 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 $\widetilde{r} < c$ and $d \geq \beta - d$ imply that $\widetilde{r} < q_0$. The first adjustment is skipped, the second is done.
- If $\widetilde{r} \geq q_0$, then $\widetilde{r} < c$ 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.[2]

Theorem 3: 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

$$\begin{aligned}
&\mathrm{P}[\widetilde{r} \geq \xi\beta \text{ or } \widetilde{r} \geq q_0] \\
&\quad = \frac{(2-1/\xi)^3}{6(1-\xi)^2} \log\frac{2-1/\xi}{\xi} + \frac{1}{6} \\
&\quad + (1-\xi)\left(-\frac{1}{18} + \frac{1}{2\xi} - \frac{11}{12\xi^2} + \frac{11}{36\xi^3}\right). \qquad (10)
\end{aligned}$$

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

$$\frac{1}{1-\xi}\int_0^{\xi+1/\xi-2} \mathrm{P}[R \geq 1 + s]\,\mathrm{d}s. \qquad (11)$$

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

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

where the latter equation is valid only for s in the interval of interest. Substituting in Eq. (11) and integrating yields Eq. (10), which completes the proof. ∎

In Fig. 1, the adjustment probability of Eq. 10 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 border case $d =$

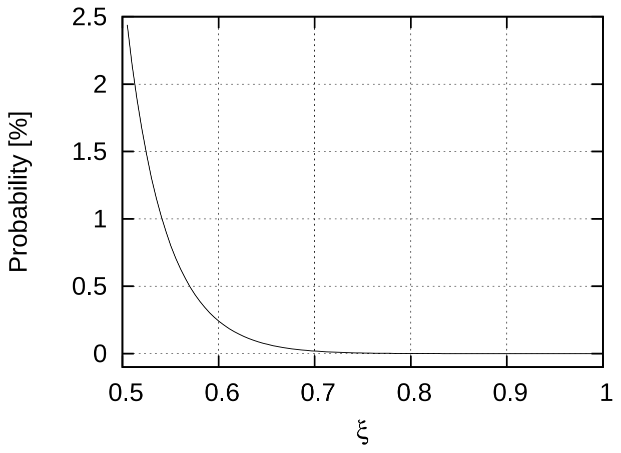

Fig. 1. Probability of the unlikely adjustment step, as a function of the ratio $\xi = d/\beta$.

[2]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.

$\beta/2$ actually gives an adjustment probability of zero, so it is not the worst case.

(FIXME: Comment on empirical data? Maybe add to plot?)

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 (denoted “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.

A. Dividing a 3-word number by a 2-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.

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 4: 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

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

```
$q, \langle r_1, r_0\rangle \leftarrow$ DIV3BY2PI1($\langle u_2, u_1, u_0\rangle, \langle d_1, d_0\rangle, v$)
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 $r_1 \ge q_0$
8      then $q_1 \leftarrow (q_1 - 1) \bmod \beta$
9           $\langle r_1, r_0\rangle \leftarrow (\langle r_1, r_0\rangle + \langle d_1, d_0\rangle) \bmod \beta^2$
10 if $\langle r_1, r_0\rangle \ge \langle d_1, d_0\rangle$                          ▷ Unlikely condition
11     then $q_1 \leftarrow q_1 + 1$
12          $\langle r_1, r_0\rangle \leftarrow \langle r_1, r_0\rangle - \langle d_1, d_0\rangle$
13 return $q_1, \langle r_1, r_0\rangle$
```

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

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

$$\begin{aligned}\widetilde{r} &= U - \widetilde{q}D \\ &= \frac{u_2 K + u_1(\beta^2 - D) + u_0\beta + q_0 D}{\beta} - D.\end{aligned}$$

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 border cases makes the proof more involved. We need to consider several cases.

- If $u_2 \le d_1 - 1$, then

$$\begin{aligned}\widetilde{r} &< \frac{(d_1 - 1)D + (\beta - 1)(\beta^2 - D) + \beta^2 - \beta D + q_0 D}{\beta} \\ &= \frac{(\beta^2 - D)^2 + q_0\beta D - d_0 D}{\beta^2} \\ &= \frac{\beta^2 - D}{\beta^2}(\beta^2 - D) + \frac{D}{\beta^2} q_0\beta - \frac{d_0 D}{\beta^2} \\ &\le c.\end{aligned}$$

```
$v \leftarrow$ RECIPROCAL_WORD_3BY2($\langle d_1, d_0 \rangle$)
 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     then $v \leftarrow v - 1$
 6          if $p \ge d_1$
 7             then $v \leftarrow v - 1$
 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  $\langle t_1, t_0 \rangle \leftarrow v d_0$                         ▷ umul
11  $p \leftarrow (p + t_1) \bmod \beta$
12  if $p < t_1$                          ▷ Equivalent to carry out
13     then $v \leftarrow v - 1$
14          if $\langle p, t_0 \rangle \ge \langle d_1, d_0 \rangle$
15             then $v \leftarrow v - 1$
16  return $v$
```

Algorithm 6: Computing the reciprocal needed for DIV3BY2PI1; a single word reciprocal based on a two-word divisor.

- If $u_2 = d_1$, then $u_1 \le d_0 - 1$, by assumption. In this case, we get

$$\begin{aligned}\widetilde{r} &< \frac{d_1 D + (d_0 - 1)(\beta^2 - D) + \beta^2 - \beta D + q_0 D}{\beta} \\ &= \frac{\beta^2 - D}{\beta^2}(\beta^2 - D) + \frac{D}{\beta^2} q_0 \beta \\ &\quad + \frac{(\beta - d_0)\big((\beta + 1)D - \beta^3\big)}{\beta^2} \\ &\le c + \frac{(\beta - d_0)\big((\beta + 1)D - \beta^3\big)}{\beta^2}.\end{aligned}$$

  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 border 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.

(FIXME: Try to simplify, by doing the borderline cases first?) ■

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. To compute the needed reciprocal, 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 $v d_0$ 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 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]. When discussing the details of instruction timing, we focus primarily on AMD Opteron. Results are provided for both AMD Opteron and Intel Core 2.

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.

```
$(Q, r) \leftarrow$ DIV_NBY1$(U, d)$
    In: $U = \langle u_{n-1} \dots u_0 \rangle$
    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       do $(q_j, r) \leftarrow$ DIV2BY1PI1$(\langle r, u_j \rangle, d, v)$
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.

```
loop:
        mov  (np, un, 8), %rax
        div  d
        mov  %rax, (qp, n, 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.

To use Alg. 7 directly, d must be normalised. To also handle unnormalised divisors, we select a shift count k such that $\beta/2 \leq 2^k d < \beta$. 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.

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 op-

```
                 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      8            add     %r12, %rax
                        mov     d, %rax
    6     10            adc     %r10, %rdx
    7     12            not     %rdx
    8     13            mov     %rdx, %r12
    8     13            mul     %rdx
   12     21            add     %rax, %r14
   13     23            adc     %rdx, %r10
   14     25            sub     d, %r10
   13     22            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.

erations, 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 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 half of the product is available in `rax`, and one more cycle until the high half 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 (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

```
                 loop:  nop
                        mov    (up,un,8), %r10
  0     0               lea    1(%rax), %r11
                        shld   %cl, %r10, %rbp
  0     0               mul    dinv
  4     8               add    %rbp, %rax
  5    10               adc    %r11, %rdx
                        mov    %rax, %r11
                        mov    %rdx, %r13
  6    12               imul   d, %rdx
 10    20               sub    %rdx, %rbp
                        mov    d, %rax
 11    21               add    %rbp, %rax
 11    21               cmp    %r11, %rbp
 12    22               cmovb  %rbp, %rax
AMD  Intel              adc    $-1, %r13
                        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 24.5 cycles on Intel Core 2.

to the 71-cycle loop based on the `div` instruction. For Intel Core 2, the latency if 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. [3] For Intel Core 2, the latency of the recurrency chain is 24 cycles, with an actual running time of 24.5 cycles per iteration.

Comparing the old and the new method, first make the conservative assumption 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 17% 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.

[3]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	24	24.5

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.

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

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 was 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 and an associated fractional word.

Like previous methods, we compute the quotient via an approximate reciprocal of the remainder. We describe new, more efficient, algorithms for computing this reciprocal, for the most common cases of a word size of 32 or 64 bits. These algorithms are pretty good, but they can most likely be improved further.

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

(FIXME: Thanks to Stephan Tolksdorf, Björn Terelius, any other?)

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.