Multiprecision division on an 8-bit processor

Eric Rice Richard Hughey
Department of Computer Engineering
University of California, Santa Cruz, CA 95064
E-mail: elrice,rph@cse.ucsc.edu

Abstract

Small processors can be especially useful in massively parallel architectures. This paper considers multiprecision division algorithms on an 8-bit processor (the Kestrel processor, currently in fabrication) that includes a small amount of memory and an 8-bit multiplier. We evaluate several variations of the Newton-Raphson reciprocal approximation methods for use with division. Our final single-precision algorithm requires 41 cycles to divide two 24-bit numbers to produce a 26-bit result. The double-precision version requires 98 cycles to divide two 53-bit numbers to produce a 55-bit result. This low cycle count is the result of several techniques including low-precision arithmetic, early introduction of dividends, and simple yet good initial reciprocal estimates.

1. Introduction

This paper presents a study of division on an 8-bit processor. It is motivated by the Kestrel architecture, an 8-bit parallel processor tuned to sequence analysis [8]. The word size is a natural choice for sequence analysis applications: characters may require 2 (DNA and RNA), 5 (protein), or 8 (text) bits, and the dynamic programming calculation at the core of many analysis methods requires a mix of precisions including 8 bits for character costs and 24 to 32 bits for accumulated global costs. From one perspective, falling midway between word-parallel and bit-serial designs, the byte-parallel processor gains some of the advantages of both word-parallel architectures (fast arithmetic units, array multipliers, low-latency operations, and the like) and bit-serial architectures (memory efficiency from adjustable word size, small circuits, faster clock speeds, lower pin requirements, and the like). Of course, it also inherits some of the disadvantages of each class as well.

The Kestrel processing element includes several separate, and often independently usable, parts. A few of its features most important to multiprecision division include a single-cycle 3-operand add-min instructions ($r = \min(a + b, c)$), single-cycle signed and unsigned 8×8 multiplication (in fact, we implement $ab + c + d$ to aid in multiprecision multiplication), single-cycle one-bit shifts, 32 8-bit registers and a locally-addressable memory of 256 bytes.

Because of the availability of a multiplier, and a lack of memory for any but the smallest tables, we took a close look at multiplicative algorithms (which typically double the precision on each iteration) as methods of efficiently implementing division, rather than additive algorithms such as digit-recurrence methods [3] and SRT division. Tuning algorithms to byte calculations is frequently not discussed in the literature [1, 9]. Typically methods are discussed in terms of the number of operations, not size of operations, though there are exceptions [12, 2, 10].

Since the analysis found was lacking for our purposes, we evaluated various suggested forms of multiplicative division. The basic principles of each were examined in terms of how accuracy needs to be maintained throughout their processes; on this basis we determined their desirability for Kestrel. We soon found the various Newton-Raphson iterations to be most promising. In Newton-Raphson methods, a recurrence, such as $r_{i+1} = r_i[2 - br_i]$, is evaluated from an initial reciprocal estimate r_0 to form a series of increasingly exact reciprocal estimates to $1/b$ [5]. These estimates converge quadratically or better (depending on the equation used), this being the prime advantage of Newton-Raphson over digit-at-a-time methods for moderately-sized numbers. Its disadvantage is the complexity of the steps, which require multiprecision multiplication.

There were several issues we considered in our search for an efficient implementation of Newton-Raphson. Most basic was to compare the several possible NR iterations available. Within each of these, the basic processes involved needed to be fine-tuned for maximal efficiency. For example, approximate calculations, while speeding up an iteration, may increase the total number of iterations needed. A similar trade-off occurs in choosing the accuracy of our initial reciprocal estimate r_0, where the cost of obtaining the estimate must be balanced with the savings in iterations that a good estimate can produce.

We evaluate methods according to two primary targets:

1063-6889/97 $10.00 © 1997 IEEE

single precision and double precision floating-point numbers. The significands of interest are thus 24 bits and 53 bits [6]. In both cases, as shall be seen, division must be carried out to additional precision to enable proper rounding.

Our results are presented as follows. After a brief discussion of savings available during calculations we turn to a detailed examination of the world of Newton-Raphson. Perhaps new here is a more efficient way of implementing the 'extended' Newton-Raphson equation as well as an extremely useful way of employing such to boost the accuracy of a result.

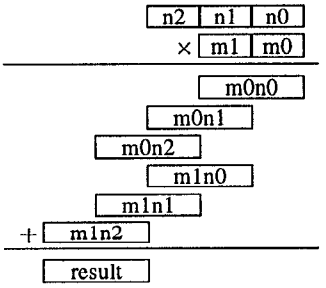
We then turn to the specifics of our algorithm, beginning with a unique way of obtaining an initial reciprocal estimate, and then a brief discussion of target accuracy in terms of leading to efficient rounding modes. Finally, the details of our algorithms for single precision and double precision division are given.

2. Considerations on 8-bit Processors

There are several situations where important savings can be made when using an 8-bit processor to perform multiple Newton-Raphson (NR) iterations; it is useful to discuss them briefly before turning to the NR algorithm itself.

Because multiprecision multiplication is the most costly part of an iteration, it is especially important to perform these as efficiently as possible. Most important in this regard is to minimize the number of bytes in the operands. This can be done to varying degrees with the different forms of the NR algorithm.

Secondly, in a multiprecision multiplication often only the most significant bytes of the result represent useful information. In such cases many of the partial products can be omitted without significantly affecting the convergence rate of the iterations. For example, suppose in a 2×3-byte multiplication we only need a close approximation of the two most significant bytes of result. The situation is pictured below, where each box in the multiplicand m and multiplier n represents a byte. This is a representation we will use throughout this paper, each small box representing a digit in base 256. The implied radix point is to the left of the boxes, except in the case of an explicit specification of the radix point with a [1.] symbol.



Here, the partial product m0n0 is of little significance and can be omitted. Less apparent is that although the partial products m0n1 and m1n0 can each impact the result by nearly 1 least significant bit (lsb), it is generally better to omit these partial products as well, as explained later. Such uncomputed partial products will be indicated with the ⊠ symbol.

Finally, also representing about 1 lsb of result each are the low-byte results of m0n2 and m1n1. In some machines, including values such as these in the result would cost one or more instructions, and might well be best omitted. This is not an issue in any architecture with a multiply-accumulate instruction, however, since these values can then be included at no cost—at most only 1 lsb of these values will be lost during the truncation to two bytes. These truncated bytes are indicated as, for example, [(75)].

When a multiprecision subtraction result does not need high precision, one can avoid performing the less significant partial subtractions. However, in any subtraction in an NR iteration, care must be taken to ensure that the result of an approximate subtraction is less than the exact result. This is because the result of the subtraction is used to produce a new estimate of a reciprocal, which we want to assume is less than the true reciprocal. Thus, in approximate subtraction, a borrow of 1 or more must be forced, especially if the number to be subtracted has been under-estimated, such as after an abbreviated multiplication.

Each of the preceding suggestions increases the speed of iterations while reducing the accuracy of calculations. Accuracy can be recovered with additional iterations or other techniques, and even accounting for these corrections there is a net savings in time. This is accentuated by the fact that once a result has a certain amount of inaccuracy, the relative impact of further losses is less significant. For example, if our least significant byte has only 5 accurate bits—is ± 8 lsb—and we introduce additional error of 4 lsb—we are now ± 12 lsb—we still have $8 - \log_2 12 = 4.4$ bits of accuracy.

A last situation in which unnecessary work can be avoided during NR iterations is when we know that the most significant byte(s) of a subtraction are going to be 0 (common in the second and subsequent iterations). In this case, we can save on both the multiplication that produces the term to be subtracted, and on the subtraction itself.

3. Observations on Newton-Raphson

Consider the problem of computing $Q = a/b$ using successive approximation to $r = 1/b$ using the Newton-Raphson method. After obtaining an approximation r_0 for $1/b$ by some means, successive estimates are typically obtained using the equation:

$$\begin{aligned} r_{i+1} &= r_i\,[1 + (1 - br_i)] \\ &= r_i\,[2 - br_i]\,, \end{aligned} \qquad (1)$$

where successive estimates converge quadratically toward an accurate value. It has also been pointed out that Newton-Raphson can be extended to:

$$r_{i+1} = r_i \left[1 + (1 - br_i) + (1 - br_i)^2 + \ldots + (1 - br_i)^n\right], \quad (2)$$

which is a more general equation, with equation 1 corresponding to $n = 1$ [4]. Looking more closely at equation 2, an inner iteration is suggested to obtain an increasingly accurate value by which the previous r_i should be multiplied, each such inner iteration requiring a multiplication by $(1 - br_i)$ and an addition:

$$1 + (1 - br_i) + (1 - br_i)^2 + \ldots + (1 - br_i)^n. \quad (3)$$

In terms of convergence, each such iteration linearly extends the accuracy of the estimate; that is, if r_i is accurate to one byte, $r_i[1 + (1 - br_i)]$ will be accurate to two bytes; $r_i[1 + (1 - br_i) + (1 - br_i)^2]$ will be accurate to three bytes, and so on.

One can thus obtain any degree of accuracy by calculating just one iteration of equation 2 provided enough inner iterations have been performed.

In fact, one obtains (theoretically) exactly the same results in this manner as by iterating repeatedly over equation 1. For example, two iterations of equation 1 lead to:

$$\begin{aligned} r_{i+1} &= r_i\,[2 - br_i] \\ r_{i+2} &= r_{i+1}\,[2 - br_{i+1}] \\ &= [r_i(2 - br_i)] \times [2 - b\,[r_i(2 - br_i)]] \\ &= r_i\left[1 + (1 - br_i) + (1 - br_i)^2 + (1 - br_i)^3\right], \end{aligned}$$

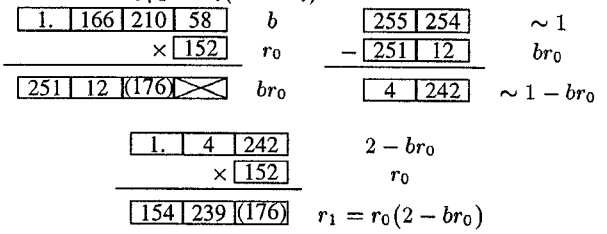
and we have equation 2 with $n = 3$.

However, although mathematically equivalent in this way, there are big differences between these two methods in practice. The following takes a closer look at each of these methods to understand some of the subtleties of each in terms of how they can be best implemented for our purposes.

3.1. Successive NR Iterations

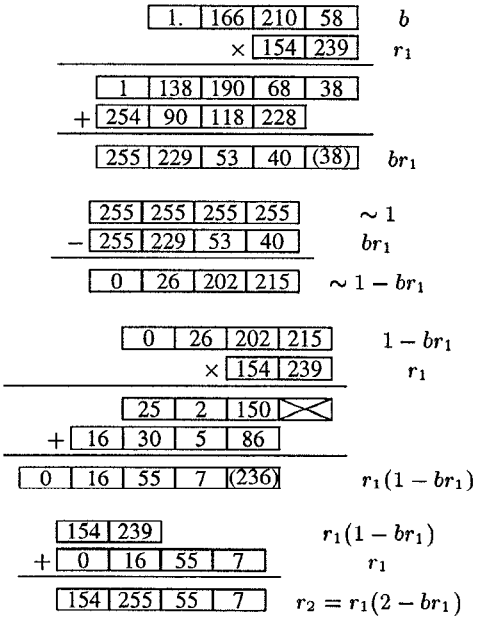
One of the key differences between successive iterations of NR equation 1 and inner iterations of equation 2 is the way in which accuracy must be dealt with to maintain the integrity of the process. Because of the potential savings of liberal rounding techniques, we would like to be able to introduce a certain amount of error along the way without fatally affecting things. Successive NR iterations are very forgiving in this way. In every iteration, a new reciprocal approximation is calculated based on the actual result of the previous iteration(s). Thus, whatever error has been introduced is forgiven and forgotten; the corrective power of the iteration takes the previous result and finds a better estimate.

This suggests a method where the number of bytes used in calculations increases only as the accuracy of the result increases. This is illustrated by the following example, where we are trying to calculate the reciprocal of $b = [1.][166][210][58]$. (In our algorithm and in the examples in this paper, we will use a divisor b normalized to $1 \le b < 2$ and thus obtain a reciprocal estimate r in the range $1/2 \le r < 1$, although it would work just as well the other way around.) Assume that we have an initial 1-byte estimate of $[152]$ ($1/[1.][166][210][58] \sim [154][255][56][187]$). Since one iteration of our equation will roughly double the number of bytes of accuracy to two, we begin with calculations aimed at producing 2-byte results. Here are the three calculations that comprise the first iteration of $r_{i+1} = r_i(2 - br_i)$:



We have made full use of all savings suggested in the previous section here. In the first multiplication, the ⋈ partial product is omitted. In the second step, we subtracted from $[255][254]$ because of this omitted partial product (a loss of up to 1 lsb) and the truncation (of $[(176)]$) of what *was* multiplied (a loss of another 1 lsb potentially).

Now that we have a 2-byte result, we will do another iteration. Again, we expect to double the number of bytes of accuracy, so this time must aim our calculations at producing 4-byte results:



Notice in these calculations that the remainders $1 - br_i$ and corrections $r_i \times (1 - br_i)$ begin with small-order bytes. This is typical when we have a good beginning reciprocal

estimate. It is worth noting that even the best 1-byte reciprocal (for example) can produce an initial remainder with a small leading term, and can, on the following iteration, produce a small leading term on the correction as well. Thus, especially in a parallel processor application where we must plan for the worst case, there is no point in trying to avoid such terms.

Although in the above example we used the classic NR equation 1 to examine the properties of equation iterations in general, if we were iterating over

$$r_{i+1} = r_i \left[1 + (1 - br_i) + (1 - br_i)^2\right] \quad (4)$$

(or any other form of equation 2), the same principles and conclusions would apply. We could, in fact, alternate equations if we wanted, the key features being that we recalculate the remainder each time on the basis of our last reciprocal estimate, but to the expected accuracy of our next one.

A final and important issue concerning the efficient use of successive NR iterations is how we introduce the dividend into the process. Although we could wait until finding a full reciprocal, it is possible to do so with advantage one iteration sooner.

To see how this can be done, consider the situation just before our last iteration. We have a reciprocal r_i that is accurate to half the targeted number of bytes. Ordinarily we would iterate equation 1 again:

$$r_{i+1} = r_i \left[1 + (1 - br_i)\right],$$

and then multiply by the dividend a to get our quotient:

$$\begin{aligned} Q &= ar_{i+1} \\ &= ar_i \left[1 + (1 - br_i)\right]. \end{aligned} \quad (5)$$

We can proceed in a slightly different way by calculating a quotient estimate $q = ar_i$ and then using equation 5 (slightly rearranged) to refine our estimate:

$$\begin{aligned} q_1 &= ar_i \left[1 + (1 - br_i)\right] \\ &= ar_i + r_i \left[a - b(ar_i)\right] \\ &= q + r_i(a - bq). \end{aligned} \quad (6)$$

Notice the similarity here to the traditional NR equation; in fact it has very similar convergence properties. This is important here: even if q is calculated only to the accuracy of r_i, the improved quotient estimate q_1 still has nearly the same accuracy as if we had proceeded in the usual way, as can be seen by the following. Letting $r_i = 1/b - \epsilon$ and $q = ar_i - \delta$, the quotient estimate from equation 5 would be:

$$\begin{aligned} Q &= ar_i(2 - br_i) \\ &= a(1/b - \epsilon)\left[2 - b(1/b - \epsilon)\right] \\ &= a/b - ab\epsilon^2, \end{aligned} \quad (7)$$

whereas our new method (equation 6) would produce:

$$\begin{aligned} q_1 &= q + r_i(a - bq) \\ &= (ar_i - \delta) + r_i\left[a - b(ar_i - \delta)\right] \\ &= (a(1/b - \epsilon) - \delta) + \\ &\quad (1/b - \epsilon)\left[a - b(a(1/b - \epsilon) - \delta)\right] \\ &= a/b - ab\epsilon^2 - b\epsilon\delta. \end{aligned} \quad (8)$$

Here, $b\epsilon\delta$ represents the added error introduced by this new process. Since the error in our reciprocal estimate (ϵ) will generally be considerably larger than the error resulting from just the one multiplication ar_i, we will generally have $\epsilon > \delta$, and the added term will not add significantly to the $ab\epsilon^2$ error that must be present.

The reason we must wait until our last iteration is that once we have introduced the dividend, we do not calculate an improved reciprocal estimate. For example, if we tried a second correction on our quotient estimate in equation 6,

$$q_2 = q_1 + r_{i+1}(a - bq_1),$$

the term r_{i+1} would not be available to produce the quadratic convergence expected after going to the trouble of calculating $a - bq_i$.

Also, it is important to note that the above process works best only for equation 1, since the $(1 - br_i)$ needed to produce additional inner terms of equation 2 has not been calculated.

To see the significance of the savings offered by this early introduction of the dividend, suppose we have an 8-byte dividend and are aiming toward an 8-byte quotient. Suppose further that we have calculated a 4-byte reciprocal. Then using the above equation, our dividend can be introduced via an 8×4 byte multiplication with a 4-byte result (requiring 17 partial products), rather than via an 8×8 byte multiplication with an 8-byte result if we wait (requiring 44 partial products). Clearly, even if a bit of accuracy has been lost, it can be recovered at less cost elsewhere!

3.2. Inner NR Iterations

We now turn to the second method listed earlier; that of refining a reciprocal estimate within a single iteration of equation 2, reprinted here:

$$r_{i+1} = r_i \left[1 + (1 - br_i) + (1 - br_i)^2 + \ldots + (1 - br_i)^n\right].$$

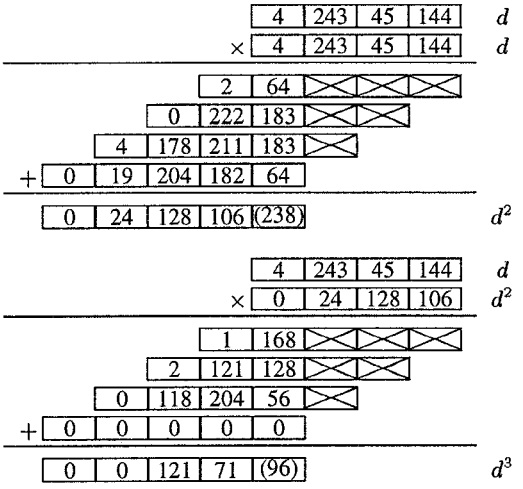
As mentioned earlier, we can achieve any degree of accuracy by performing only inner iterations, assuming enough of them are performed. There are some significant differences in how we must proceed here, however. To see what they are, we will calculate the same reciprocal earlier performed using traditional NR iterations. As before, the first step is to calculate $d = (1 - br_0)$. This time, however, because it will be used to produce all of the inner terms (which in turn produce the correction to r_0), we must do the calculations to the full accuracy of our desired result, in this case 4 bytes:

1.	166	210	58	b
		× 152		r_0
251	12	210	112	br_0

1.	0	0	0	0	1
− 251	12	210	112		br_0
4	243	45	144		$1 - br_0$

We can now calculate the remaining terms needed to obtain the desired accuracy. Since each inner term extends the accuracy linearly and we want to produce a 4-byte result from a 1-byte estimate, we will need three terms alto-

gether. Thus we will need to calculate the additional terms $d^2 = (1 - br_0)^2$ and $d^3 = (1 - br_0)^3$, each accurate to 4 bytes:



We can then combine these results to obtain our quotient:

$$\begin{aligned} Q &= r_0 \times (1 + d + d^2 + d^3) \qquad (9)\\ &= [152](1 + [4\ 243\ 45\ 144] \\ &\quad + [0\ 24\ 128\ 106] + [0\ 0\ 121\ 71]) \\ &\sim [154\ 255\ 55\ 78]. \end{aligned}$$

There is another way of obtaining our estimate of $(d + d^2 + d^3 ...)$ that offers considerable savings over the straightforward approach above. Mathematically it is based on a series of steps where a quantity u_i (initially set to d) is divided into its most significant part m_i (typically two bytes) and the remaining portion n_i. The recurrence is:

$$\begin{aligned} (m_i, n_i) &= \text{split}(u_{i-1}) \\ u_i &= dm_i + n_i. \qquad (10) \end{aligned}$$

Letting $s_i = m_1 + m_2 + ... + m_i$, we can then use $s_i + u_i$ as our i-th approximation of $S = d + d^2 + d^3 ...$, as after any number of iterations of above we will have (easily shown by induction):

$$s_i + u_i = S - u_i S. \qquad (11)$$

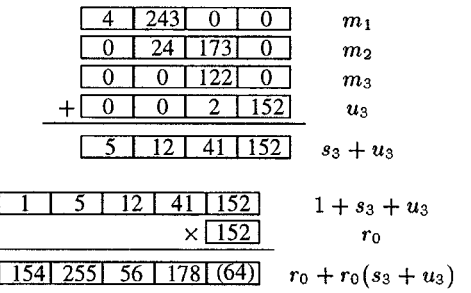
Thus, our error will be proportional to the size of u_i.

To see how this works in practice, let us look at our earlier problem again. After calculating $d = 1 - br$ as above, we perform our new method:

$$\begin{aligned} u_0 &= d = [4\ 243\ 45\ 144] \\ m_1 &= [4\ 243] \text{ and } n_1 = [0\ 0\ 45\ 144] \\ u_1 &= dm_1 + n_1 = [0\ 24\ 173\ 25] \\ m_2 &= [0\ 24\ 173] \text{ and } n_2 = [0\ 0\ 0\ 25] \\ u_2 &= dm_2 + n_2 = [0\ 0\ 122\ 61] \\ m_3 &= [0\ 0\ 122] \text{ and } n_3 = [0\ 0\ 0\ 61] \\ u_3 &= dm_3 + n_3 = [0\ 0\ 2\ 152] \end{aligned}$$

Note that above we chose an m_3 with only one non-zero byte instead of the usual two. This is because it is generally not efficient to use a 2-byte m_i unless the product $d \times m_i$ zeros an additional byte.

We now calculate $r_0 + r_0(m_1 + m_2 + m_3 + u_3)$ to get our reciprocal estimate:



By the end of the above example, we have obtained a significantly more accurate result than in equation 9 while saving several instructions.

3.3. NR Accuracy Fix

There is an extremely useful application of the extended Newton-Raphson equation that deserves special attention. This is the method promised earlier (when promoting liberal rounding) that provides a cheap way of gaining needed accuracy. Consider our first example, in which we calculated $1/[1.\ 166\ 210\ 58]$ to 4 bytes from an initial estimate of $[152]$. We achieved the following values after the second iteration:

$$\begin{aligned} r_1 &= [154\ 239] \\ 1 - br_1 &= [0\ 26\ 202\ 215] \\ r_1(1 - br_1) &= [0\ 16\ 55\ 7] \\ r_2 &= [154\ 255\ 55\ 7] \\ \text{true reciprocal} &= [154\ 255\ 56\ 187] \end{aligned}$$

Notice that in fact we do not have 4 bytes accuracy as promised—since our initial reciprocal estimate was accurate to less than a full 8 bits, we have ended up with only about 3 bytes of accuracy. This error, however, can be easily and significantly reduced by adding a corrective term from the extended NR equation:

$$\begin{aligned} r' &= r_2 + r_1(1 - br_1)^2 \qquad (12)\\ &= r_2 + [0\ 26\ 202\ 215] \times [0\ 16\ 55\ 7] \\ &\sim [154\ 255\ 56\ 185]. \end{aligned}$$

The multiplication above can be carried out to whatever accuracy is needed. The above was performed using our liberal rounding techniques, but even a 1×1-byte multiplication leads to an improved quotient estimate of $[154\ 255\ 56\ 167]$.

This correction could as easily have been obtained by making a corrective fix on the remainder using:

$$r' = r_1 + r_1\left[(1 - br_1) + (1 - br_1)^2\right]. \qquad (13)$$

In either case, the difference between the use of the added inner NR term here and in our earlier discussion is that here it is not used to extend the number of bytes of result (this being limited by the accuracy of the remainder), but is rather

used to make a correction on the bytes we already have. In fact, it is because we do not need to calculate a new or more accurate remainder that makes it such an inexpensive fix.

It is worth noting, too, that when using this corrective fix, the larger the magnitude of the leading bytes of the terms being multiplied, the more efficient the fix—such terms contain more information. Thus, if we are going to employ this sort of correction, we should wait as long as possible—until either the magnitude of the most significant bytes of our remainder or correction threaten to creep into the next byte, or until we cannot wait any longer because of the (early) introduction of the dividend into the process (equation 6), where we produce the remainder $(a - bq)$ rather than the $(1 - br)$ needed for such an additional corrective term.

3.4. Comparing and Combining the Methods

There are many ways of combining the methods outlined above; in this section we try to sort them out.

A comparison of particular importance is that of the relative efficiency of multiple iterations of the classic equation 1, and one iteration (with multiple inner iterations) of equation 2. Beginning with a 1-byte reciprocal estimate, we found that both methods require virtually the same number of steps (using the most efficient implementations) to produce either a 4-byte reciprocal or 4-byte quotient. To obtain 8-byte results, however, the classic NR, with precision doubling on each iteration, was found clearly superior. The same pattern was found for calculations beginning with a 2-byte reciprocal estimate.

The conclusion from this seems to be that unless seeking a simple hardware implementation, the extended NR equation should never be used with more than a few inner iterations. In fact, echoing conclusions reached earlier using less byte-oriented analysis [11], it appears that one loses little (if anything) by limiting consideration to just equation 1,

$$r_{i+1} = r_i \times [1 + (1 - br_i)],$$

which doubles the accuracy, and

$$r_{i+1} = r_i \times \left[1 + (1 - br_i) + (1 - br_i)^2\right], \qquad (14)$$

which triples the accuracy.

For example, even a 5-byte target accuracy—which one might expect to be best obtained by the extended NR with 4 inner iterations—can be as efficiently obtained using the above two equations, where on the second iteration, calculations are performed to just 5 bytes.

Because only equation 1 allows the most efficient early introduction of the dividend, it is preferred for the final iteration of any algorithm. Thus, for example, if a 6-byte quotient is needed from a 1-byte reciprocal estimate, it is more efficient to use equation 14 first (providing 3 bytes) and then equation 1 modified for the early introduction of the dividend (6), rather than the other way around.

4. Specifics of our Final Algorithm

We now turn attention to the details of our algorithms for single and double precision division—how the original reciprocal estimate is obtained, what our target accuracy is, and how we get from one to the other.

4.1. Obtaining a reciprocal estimate

Because of the limited memory associated with each processor in Kestrel—as well as for convenience to the user—our primary algorithm was developed to calculate the beginning reciprocal estimate without use of a look-up table.

One method found of obtaining a reasonably good estimate was with the quadratic equation:

$$r_0 = (81 \times n - 203) \times n + 249, \qquad (15)$$

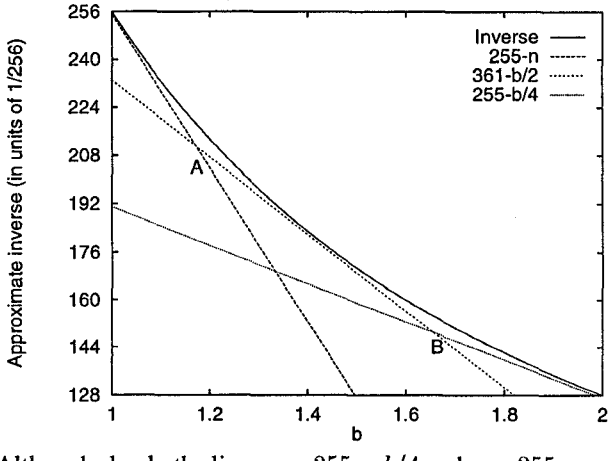
where n is the leading fractional byte of b, and where the result of each multiplication is immediately truncated to one byte. This equation was chosen to best serve the needs of upcoming NR iterations, for which, as it turns out, the accuracy of beginning reciprocal estimates of small b's is less critical than for large b's (the above equation provides its worst estimate for $n = 0$).

A slightly better method was found for Kestrel that takes advantage of a couple of its features:

$$r_0 = \max(255 - n, 361 - b/2, 255 - b/4), \qquad (16)$$

where n is again the leading fractional byte of b, and only the leading bytes of $b/2$ and $b/4$ are used to calculate the other terms.

Equation 16 finds the best estimate provided by 3 near-tangents to the curve $1/b$:



Although clearly the lines $y = 255 - b/4$ and $y = 255 - n$ are not optimal in terms of minimizing the error of the estimate (there are pockets at A and B), this choice of lines has the advantage of being easily calculated. In Kestrel, equation 16 requires just 3 instructions to evaluate, thanks to its bit-shifter (which obtained $b/4$ at no cost from $b/2$) and especially to its comparator, which allows a subtraction and maximization in one instruction.

Looking at an even simpler equation, if we use only $(361 - b/2)$ for our estimate, the cost of getting to the accuracy obtained by above estimate methods is greater than the savings offered by such; similarly, we did not find a method of obtaining a *more* accurate initial estimate that was cost-effective.

Also, after examining several ways of using nominal look-up tables to obtain r_0, the only practical one found for Kestrel (with only 256 bytes of memory per processing element) was a 192-element table in which each entry includes 1 byte to specify slope and 2 bytes to specify the constant. Using this table allows bypassing of one of the NR iterations in our algorithms, saving several instructions.

4.2. The choice of target accuracy

Before discussing the number of bits of precision required for correct rounding, it needs to be mentioned that our algorithms for single and double precision division are both aimed at producing a quotient in the range $1/2 \leq q < 1$. To accomplish this, a is introduced in the range $1/2 \leq a < 1$ and after our NR processes we obtain a quotient $1/4 \leq q < 1$, requiring left-normalizing, if anything. These ranges are particularly convenient for single precision, where they are easily unpacked from and packed into the standard floating-point format.

In choosing target accuracy, we were particularly interested in two rounding modes:

1. One that inexpensively provides accurate results when perfect results are expressible ($1/2 = 0.1000\ldots$ rather than $0.0111\ldots$), and

2. One that performs round-to-nearest as per single-precision IEEE 754 standards [6]. ('Rounding to even' is thankfully avoided since floating-point division can never yield an accurate result of the form, for example, $1.x_1x_2\ldots x_{24}1 \times 2^n$.)

These two rounding modes turn out to require the same target accuracy for efficient implementation. Limiting our discussion to single precision (the requirements of double precision being analogous), we will need accuracy within 2^{-26}. After normalizing (and thus possibly losing one of the extra two bits of accuracy), the first rounding mode above can be accomplished by adding 2^{-25} to the result. This leads to the accurate result desired, with a maximum error of less than 2^{-24} (1 lsb of result).

Round-to-nearest is more complicated but can also be done efficiently if our quotient is within 2^{-26} of accurate. Here, if examining the 2^{-25} and 2^{-26} bits are not sufficient to do so, rounding can be determined by multiplying the quotient by b or $b/2$ (depending on whether the quotient has been normalized), and then using the 2^{-25}th bit of this result to determine correct rounding.

4.3. Basic algorithm for multiprecision division

Given the preceding, the main body of the algorithm for single precision (getting the 2^{-26} accuracy needed by whichever rounding mode is used) is straightforward:

1. Obtain the 1-byte reciprocal estimate.

2. Perform an NR iteration aimed at 2 bytes (with a corrective fix on the remainder) to obtain a 2-byte reciprocal.

3. Calculate a 2-byte quotient estimate from above.

4. Perform the NR-like iteration of equation 6 to obtain a 4-byte quotient estimate.

In Kestrel, the algorithm produces correct results for single precision when full liberal rounding is used throughout. After obtaining the values of the dividend and divisor from the floating-point representation, the single precision algorithm takes 41 instructions to achieve the 2^{-26} accuracy needed for normalizing and rounding, just over the cost of 2 full 4×4 byte multiplications.

For double precision, an extra iteration of equation 1 is needed after obtaining our initial reciprocal estimate, after which steps 2–4 are aimed at 4 bytes and 8 bytes. Here again, correct results are achieved when full liberal rounding is used throughout. After unpacking values from floating point representation, the double precision algorithm requires 98 instructions in Kestrel to achieve the necessary 2^{-55} accuracy.

As seen from the table below, the entire program without lookup table and using the simpler rounding mode requires 64 instructions for single precision and 151 instructions for double precision (using an exponent of $00\ldots0$ or $11\ldots1$ to represent out-of-range results). The instruction overhead can be reduced from 23 for single precision and 53 for double precision to 13 and 27, respectively, if packing and unpacking can be skipped, as in a chain of floating-point calculations. In the target system of 8 64-PE chips running at 33MHz, the 64-instruction program corresponds to $8 \times 64 \times 33/64 = 264$ million divisions per second.

Operation	Cycles	
	SP	DP
Unpacking	6	21
Division algorithm	41	98
Rounding	1	1
Normalizing	4	7
Calculate sign and exponent	8	12
Repack	4	12
Total	64	151

The next table displays instruction counts for several variants of the algorithms described in this paper along with

three non-multiplicative algorithms. The first three entries show the relative savings offered by several of our techniques. The next three provide the instruction counts for our final algorithm on Kestrel (with its multiply-accumulate-accumulate instruction), and on Kestrel-like architectures not so optimized for multi-precision multiplication. The next line shows the performance when using a lookup table for initial reciprocal estimates on Kestrel. Finally, the table includes estimated instruction counts for three alternative methods: the well-known SRT division algorithm in radix 4 [7], Ercegovac and Lang's multiprecision division algorithm [2], and radix 2 nonrestoring division (the SP numbers in these cases are for only 24 bits of precision, and assume an appropriately modified Kestrel). None of the values in the table include the overhead of the rest of the floating-point calculation.

Method	Cycles	
	SP	DP
W/o reduced operand size	83	399
W/o omitted partial products	48	154
W/o early dividend	48	125
Our algorithm with mult-acc-acc	41	98
Our algorithm with mult-acc	47	136
Our algorithm with mult	61	186
With 192-byte table	33	93
Radix-4 SRT	96	244
Ercegovac and Lang	87	298
Radix-2 Nonrestoring	96	416

5. Conclusions

Design of an efficient multiplicative division algorithm for small processors requires attention to detail. There are several methods at ones disposal and numerous choices along the way. However, there are several principles that can help in this effort.

First, one should not perform calculations to too much accuracy during the process; it is more efficient to achieve this same accuracy in other ways, especially by means of an accuracy fix as described above.

Second, using this approach one can limit consideration to just two forms of the 'extended' Newton-Raphson equation and still be assured of a good result:

$$r_{i+1} = r_i\left[1 + (1 - br_i)\right]$$

and

$$r_{i+1} = r_i\left[1 + (1 - br_i) + (1 - br_i)^2\right].$$

Third, because of the considerable savings offered by such, one will almost certainly want to include the early introduction of the dividend in any algorithm, in which a quotient estimate q of half the target accuracy is obtained, followed by a last NR-like iteration that achieves the desired target accuracy using

$$Q = q + r_i(a - bq).$$

A final conclusion is that if designing an algorithm for an 8-bit processor without use of a look-up table, the most efficient means of finding an initial reciprocal are ones which take only a handful of instructions, such as the two presented above, and use additional iterations to achieve the final accuracy.

6. Acknowledgments

This work was supported in part by NSF grant MIP-9423985 and its REU supplement. The authors thank Don Speck and the rest of the Kestrel team for many helpful discussions, and the reviewers for their excellent and detailed comments. More information on the Kestrel project can be found at http://www.cse.ucsc.edu/research/kestrel.

References

[1] J. J. F. Cavanagh. *Digital computer arithmetic*. McGraw-Hill Book Co., New York, 1984.

[2] M. D. Ercegovac and T. Lang. Multiplication/ division/ square root module for massively parallel computers. *INTEGRATION, the VLSI journal*, 16:221–234, 1993.

[3] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-recurrence algorithms and applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1994.

[4] D. Ferrari. A division method using a parallel multiplier. *IEEE Trans. Comput.*, EC-16:224–226, Apr. 1967.

[5] M. J. Flynn. On division by functional iteration. *IEEE Trans. Comput.*, C-19(8):702–706, Aug. 1970.

[6] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA, 1996.

[8] J. D. Hirschberg, R. Hughey, K. Karplus, and D. Speck. Kestrel: A programmable array for sequence analysis. In *Proc. Int. Conf. Application Specific Array Processors*, pages 25–34, Los Alamitos, CA, July 1996. IEEE CS.

[9] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[10] E. V. Krishnamurthy. On optimal iterative schemes for high-speed division. *IEEE Trans. Comput.*, C-19(3):227–231, Mar. 1970.

[11] E. V. Krishnamurthy. Economical iterative range-transformation schemes for division. *IEEE Trans. Comput.*, C-20(4):470–472, Apr. 1971.

[12] D. C. Wong and M. J. Flynn. Fast division using accurate quotient approximations to reduce the number of iterations. In *Proc. 10th Symp. Computer Arithmetic*, pages 191–201. IEEE, June 1991.