Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations

Derek C. Wong and Michael J. Flynn
Electrical Engineering Department
Stanford University
Stanford, CA 94305

Abstract

A class of iterative integer division algorithms is presented based on look-up table and Taylor-series approximations to the reciprocal. The algorithm iterates by using the reciprocal to find an approximate quotient and then subtracting the quotient multiplied by the divisor from the dividend to find a remaining dividend. Fast implementations can produce an average of either 14 or 27 bits per iteration, depending on whether the basic or advanced version of this method is implemented. Detailed analyses are presented to support the claimed accuracy per iteration. Speed estimates using state-of-the-art ECL components show that this method is faster than the Newton-Raphson technique and can produce 53-bit quotients of 53-bit numbers in about 28 or 22 ns for the basic and advanced versions. [1]

1 Introduction

This is a description of a high-speed divide algorithm. The algorithm operates on positive, binary, non-redundant integers. Since an exact remainder can be produced, rounding is straightforward. A reduced number of iterations is used to produce the result, and the algorithm is general and can be applied to any size numbers. With a sufficient number of iterations, any accuracy of quotient can be produced, with a well-defined remainder (where remainder is defined relative to the significance of the last digit of the quotient). For the purposes of this paper, we concentrate our examples on 53-bit integers (same number of bits as the mantissa of the IEEE floating-point standard).

Unlike other subtractive algorithms which produce 1-3 bits of quotient per iteration, a reasonable implementation of the basic algorithm takes 4 iterations to divide 53-bit numbers to produce a 53-bit quotient or about 14 bits/iteration. Implementations of the advanced version of this algorithm can produce a 53-bit result in 2 or even 1 iteration.

Some comparative work may be found in the following references. In references [1] and [5], Atkins and Fandrianto describe higher-radix methods using SRT. Krishnamurthy [9] describes a related idea for iterative division by transforming the range of the divisor to a number close to one. In this case, the leading bits of the dividend can be used as bits of the quotient. The Cyrix co-processor's division algorithm [11] is based on an algorithm similar to the one discussed in section 2.2 (more details later in this paper).

As RAM density and speed increases, the use of larger look-up tables becomes practical and advantageous. This algorithm accurately estimates the reciprocal using large look-up tables. Since the algorithm can then estimate quotients accurately, subtraction can reduce the dividend by many bits per iteration.

This paper first outlines our method. This method is then compared to other methods using reciprocal approximations. Then speed estimates are made for fast implementations of this scheme. This is compared to the performance of other algorithms using the same technology assumptions.

2 The Algorithm

This algorithm uses a fixed look-up table to get adjustments to the current quotient. Each adjustment, appropriately shifted, is added to the previous cumulative estimate of the quotient. Thus, each new estimate is an adjusted version of the previous. In this method, each estimate is always less than or equal to the true quotient, so that each new adjustment is always non-negative (unlike in non-restoring division methods).

The quotient is successively refined: after the N-th iteration, the quotient is accurate to approximately within 1 part in $2^{(14*N)}$ using the basic algorithm.

2.1 Notation

Denote the dividend register by X, the divisor register by Y, and the quotient register by Q. Denote a bit of a register using an index (e.g. Q_i) with higher indices indicating more significant bits. The LSB has index 0.

Without loss of generality, in this discussion, we consider that the binary representations of X, Y, Q, and other numbers are actually fixed-point fractional numbers between 0 and 1. The actual hardware operations work on binary numbers which could represent either integers or fixed-point numbers.

We describe the algorithm for positive numbers; handling negative numbers is straightforward using a sign-magnitude representation. All numbers are assumed to

[1] This work was performed with support from the Center for Integrated Systems at Stanford University, and from NSF Contract No. MIP88-22961 using equipment provided by NASA under contract NAGW 419.

have the same width q; the algorithm can easily be modified for numbers with different width. The most significant bits are then denoted by $X_{(q-1)}$ and $Y_{(q-1)}$. (Note: later it will be explained that some of the registers must be extended with low-order bits to make registers of width greater than q. These extended bits have negative indices.)

2.2 Division by Approximating Q and Subtracting

Ordinary pencil-and-paper division can be generalized into the following method:

1. Initialize the initial quotient $Q = 0$ and a shift index $j = 0$.
2. Find an approximation Q_a to X/Y such that $Q_a \leq X/Y$.
3. $Q' = Q + Q_a * (1/2^j)$.
4. $X' = X - Q_a * Y$.
5. Shift X' left by k bits until the leading bit is 1. Set $j = j + k$.
6. $X = X'$, $Q = Q'$.
7. Repeat 2 through 6 until $j \geq q$.
8. The final X is the remainder.

If the approximation Q_a is quite accurate, then the number of bits k that are reduced in an iteration is large. The number of iterations is then small.

Unlike most division algorithms that calculate several bits of the quotient per iteration, this algorithm calculates an additive adjustment to the previous quotient estimate. The adjustments get exponentially smaller with each iteration. However, no bits of the quotient are guaranteed until the algorithm terminates; even the MSB of the quotient Q could change during the last iteration if a carry propagation occurs. For instance, if the intermediate result is for instance 10011111111..111, the actual result could be 1010000000..000001.

We now describe two versions of this method based on calculating Q_a using an approximate reciprocal multiplied by the leading bits of X. The second version also uses a novel approach to approximate the reciprocal.

2.3 A Basic Method

Suppose X and Y are left-shifted until their leading bits are both 1 (denote this operation as *pre-shifting or normalization*). The amount of shifting is remembered, so that the proper significance of the leading quotient bit is known. Given the first p bits of X, the value of X is known within an error of $1/(2^p)$. The minimal value of X would be $X_{(q-1)}..X_{(q-p)}000..0$ and the maximal value would be $X_{(q-1)}..X_{(q-p)}111..1$. The range of uncertainty is slightly smaller than the significance of $X_{(q-p)}$, which is $1/(2^p)$. Similarly, given the first m bits of Y, we know Y to within $1/(2^m)$.

Define X_h as the leading p bits of X extended with 0's to get a q-bit number:

$$X_h = X_{(q-1)}..X_{(q-p)}00000..0 \qquad (1)$$

Define Y_h as the leading m bits of Y extended with 1's to get a q-bit number:

$$Y_h = Y_{(q-1)}..Y_{(q-m)}11111..1 \qquad (2)$$

Let $\Delta X = X - X_h$ and $\Delta Y = Y - Y_h$. The deltas ΔX and ΔY are the adjustments needed to get the true X and Y from X_h and Y_h. Due to the definitions of X_h and Y_h, ΔX is always non-negative, and ΔY is always non-positive.

Suppose that we know the answer to $1/Y_h$. This is clearly less than or equal to the true reciprocal $1/Y$. Similarly, X_h/Y_h is always less than or equal to X/Y.

A division algorithm that uses a reduced number of iterations can then be conceptually summarized as follows:

1. Set the estimated quotient Q to 0 initially.
2. Let j and k denote the number of bits that X and Y have been left-shifted. Initially, both are set to the amounts of pre-shifting required to normalize X and Y.
3. Get an approximation of $1/Y_h$ from a look-up table called G_1. The index to the look-up table is the leading m bits of Y. Each table word is b_1 bits wide where $b_1 = m$ as discussed later in this paper.
4. Compute the new dividend as
   $X' = X - X_h * (1/Y_h) * Y$ (footnote [2]).

   Although this step appears to require two sequential multiplications to compute $X_h * (1/Y_h) * Y$, this is not really true. By doing the multiplication of $X_h * Y$ while the look-up of $(1/Y_h)$ is being performed, only one subsequent multiplication is needed in the first iteration. In the second and later iterations, $(1/Y_h) * Y$ should ideally be pre-computed as it is invariant across the iterations. For full speed, this requires 2 multipliers during the 1st iteration, one to compute $(1/Y_h) * (X_h * Y)$ and one to compute $(1/Y_h) * Y$.
5. Compute the new quotient
   $Q' = Q + (X_h/Y_h) * (1/2^{(j-k)})$
   $= Q + X_h * (1/Y_h) * (1/2^{(j-k)})$.

   In the above formulas for Q' and X', X_h can be composed of $p = m + 1$ leading digits of X followed by 0's.
6. Left shift X' by $m - 2$ bits to get rid of guaranteed leading 0's that occur (as discussed in the next subsection). Shift zeroes into the LSB of X' during this process.

   Calculate a revised $j' = j + m - 2$.
7. Set $j = j'$, $Q = Q'$, and $X = X'$.
8. Repeat steps 4 through 7 until $j \geq q$.

[2] Note: Although it might appear possible to simplify this to $X' = X - X_h * (Y_h + \Delta Y)/Y_h = X_h + \Delta X - X_h * (1 + \Delta Y/Y_h)$ $= \Delta X - X_h * \Delta Y/Y_h$,
this does not actually work because the calculation of X' and Q' would be uncoordinated. Different round-off errors would occur to X' and Q' in each iteration.

9. The final Q is the quotient. Note that there are more bits of the quotient register than q. Let $Q_h = Q_q..Q_0$ and let $Q_l = Q_{-1}..Q_{-e}$ where e is the number of excess bits. The Q_h bits are correct to within one unit of the least significant bit. Some of the Q_l bits are not necessarily correct since another iteration of the algorithm would add into those bits.
10. The residual dividend X should be right-shifted by $j - q$ bits to get the remainder assuming the entire Q is the quotient. If Q is truncated to q bits, then the true remainder can be computed by then adding $Q_l * Y$ to X.

Since the algorithm reduces X' by $m-2$ bits per iteration, the algorithm terminates after $\lceil q/(m-2) \rceil$ iterations.

Next, we analyze this method to show why $m-2$ bits per iteration are guaranteed in the worst-case.

2.4 How Many Bits per Iteration are Guaranteed Using the Basic Method?

In this section, an analysis is performed to show the following Theorem:

Theorem 1: Number of Bits Per Iteration Using the Basic Method

X' is reduced by at least $m-2$ bits per iteration if $m \geq 5$, the number p of significant bits in X_h is equal to $m+1$, and the word width b_1 of Table G_1 is m.

Analysis

How do we compute the minimum number of bits that are reduced from X in each iteration?

The initial value of X is between 1/2 and 1 since the leading bit is 1. By determining the largest possible value of X', we can determine the worst-case number of bits eliminated per iteration. If we can guarantee that $X' < 1/2^r$, then the most significant bit that could be non-zero has significance $1/2^{(r+1)}$. In this case, there are r guaranteed leading zeroes that can be eliminated per iteration.

Let us examine the formula for X':

$$X' = X - X_h * (1/Y_h) * Y \tag{3}$$

In this case, two sources of inaccuracy affect the approximation $(1/Y_h) \approx 1/Y$. Let the error sources be defined by:

$$(1/Y_h) = 1/Y - R_b - R_c \tag{4}$$

where R_b represents an error term due to using $1/Y_h$ instead of $1/Y$, and R_c represents the error in storing $1/Y_h$ to only a finite number of bits in the table G_1. The error R_b is always non-negative since $Y_h \geq Y$ meaning $1/Y_h \leq 1/Y$. The error R_c is always non-negative since $1/Y_h$ is stored into G_1 by always rounding down to the finite width b_1. Then

$$\begin{aligned} X' &= X - X_h * (1/Y_h) * Y \\ &= X_h + \Delta X - X_h * (1/Y - R_b - R_c) * Y \\ &= \Delta X + X_h * (R_b + R_c) * Y \end{aligned} \tag{5}$$

If R_b and R_c can be accurately bounded, then this will give the maximal value of X'.

To compute the worst-case R_b, we can examine a Taylor series expansion of $1/Y$ about the point $Y = Y_h$:

$$\begin{aligned} B &= 1/Y \\ &= 1/Y_h - \Delta Y/{Y_h}^2 + (\Delta Y)^2/{Y_h}^3 \\ &\quad - (\Delta Y)^3/{Y_h}^4 + (\Delta Y)^4/{Y_h}^5 + \ldots \end{aligned} \tag{6}$$

The $(w+1)$-th term is denoted by $B_{w+1} = (-\Delta Y)^w/{Y_h}^{(w+1)}$. All terms of B are non-negative since ΔY is non-positive. The worst-case occurs when $Y_h \approx 1/2$ and $-\Delta Y = 1/2^m - 1/2^q$. In this case, the following bound holds:

$$B_{w+1} < 1/2^{(m*w-w-1)} \tag{7}$$

In this case, the approximation $1/Y_h \approx 1/Y$ is equivalent to truncating the Taylor series after the first term. A bound on the sum of the remaining terms can serve as a bound on R_b:

$$R_b = \sum_{g=1}^{\infty} B_{g+1} < \sum_{g=1}^{\infty} 1/2^{(m*g-g-1)} = \frac{(1/2^{(m-2)})}{(1-1/2^{(m-1)})} \tag{8}$$

For $m \gg 1$, this is just slightly greater than $1/2^{(m-2)}$. For $m \geq 5$, a non-stringent bound on R_b is:

$$R_b < 1.1 * (1/2^{(m-2)}) \tag{9}$$

R_c comes from not using the exact value of $1/Y_h$ but instead using a finite-precision table G_1. Denote the output of the table by $(1/Y_h)_a$.

Let the error between the table lookup and the true value be ϵ_1:

$$(1/Y_h)_a = \epsilon_1 + 1/Y_h \tag{10}$$

$$R_c = \epsilon_1 \tag{11}$$

Suppose table G_1 is b_1 bits wide. Since $1/2 < Y_h < 1$, the maximal value of $1/Y_h$ is slightly less than 2. If words in the table G_1 can represent values up to but not including 2, the unit of the most significant binary digit in the table G_1 should have value 1. The unit of the least significant binary digit for table G_1 is then $1/2^{(b_1-1)}$. The error ϵ_1 is less than the unit of the least significant digit:

$$R_c = \epsilon_1 < 1/2^{(b_1-1)} \tag{12}$$

Using this, we can determine the proper word width b_1. Suppose that b_1 is set to:

$$b_1 = m \tag{13}$$

The error R_c is then bounded by:

$$R_c < 1/2^{(m-1)} \tag{14}$$

Now we can substitute the bounds for R_b and R_c into equation 5 for X'. Since we set $Y_h \approx 1/2$ previously, Y should also equal 1/2. Since $X_h < 1$ and $\Delta X \leq 1/2^p - 1/2^q$, the worst-case X' is bounded by:

$$\begin{aligned} X' &= \Delta X + X_h * R_b * Y + X_h * R_c * Y \\ &< 1/2^p - 1/2^q + 1.1 * (1/2^{(m-2)}) * 1/2 \\ &\quad + (1/2^{(m-1)}) * 1/2 \end{aligned} \quad (15)$$

If $p = m + 1$, then this can be converted to:

$$X' < 1/2^{(m+1)} - 1/2^q + 1.6 * (1/2^{(m-1)}) \quad (16)$$

$$X' < 0.25/2^{(m-1)} - 1/2^q + 1.6/2^{(m-1)} \quad (17)$$

$$X' < 1/2^{(m-2)} \quad (18)$$

This gives an insight into a good value of p, the number of significant bits in X_h. If $p \geq m + 1$, then the worst-case value of $X' < 1/2^{(m-2)}$. In this case, the highest-order bit of X' that could possibly be 1 is the $(m-1)$-st bit, $X'_{(q-m+1)}$. Since p determines the size of the multipliers needed to evaluate X' and Q' and there is very little advantage to having $p > m+1$, the best value is $p = m + 1$.

Thus, at least $m - 2$ bits are skipped per iteration if $p = m + 1$ and $b_1 = m$.

The average number of bits per iteration is a couple of bits more than this worst-case value. However, to take advantage of this requires a leading 0's detect and variable-sized shift in step 6 of the above algorithm. This slows down the iteration time sufficiently that it is generally not worthwhile for $m \geq 10$ or so. In this case, we have omitted the analysis for the average number of bits.

2.5 An Advanced Method

Let us examine the Taylor series approximation equation for $1/Y$ about $Y = Y_h$:

$$\begin{aligned} 1/Y &= 1/Y_h - \Delta Y/{Y_h}^2 + (\Delta Y)^2/{Y_h}^3 \\ &\quad - (\Delta Y)^3/{Y_h}^4 + (\Delta Y)^4/{Y_h}^5 + \ldots \end{aligned} \quad (19)$$

All these terms are non-negative since $\Delta Y \leq 0$.

Our current method is to consider the terms beyond $1/Y_h$ to be an unavoidable error. Actually, however, it is possible to rapidly calculate one or more adjustment terms to attain additional accuracy in estimating $1/Y$.

The revised method would be as follows:

1. Set the estimated quotient Q to 0 initially.
2. Let j and k denote the number of bits that X and Y have been left-shifted. Initially, both are set to the amounts of pre-shifting required to normalize X and Y.
3. Get an approximation of $1/Y_h$ from look-up table G_1. The index to the look-up table is the leading m bits of Y. Each table word is b_1 bits long. (Note: later in the paper, the table size is optimized.)

   Simultaneously get approximations of ${1/Y_h}^2$, ${1/Y_h}^3$, etc. using look-up tables G_2, G_3, etc. with word widths b_2, b_3, etc. respectively. All tables are indexed using the first m bits of Y.

   In the next subsection, it will be shown that reasonable table widths b_i are given by:

   $$b_i = (m * t - t) + \lceil log_2 t \rceil - (m * i - m - i) \quad (20)$$

   This equation states that the tables of more significance require more bits of precision.
4. Compute an approximation B to $1/Y$ using the first t terms from the following series:

   $$\begin{aligned} B &= 1/Y_h - \Delta Y/{Y_h}^2 + (\Delta Y)^2/{Y_h}^3 \\ &\quad - (\Delta Y)^3/{Y_h}^4 + (\Delta Y)^4/{Y_h}^5 + \ldots \end{aligned} \quad (21)$$

   The number of terms t is at least 2; all terms after the second are optional. This can be done with a generation of partial products followed by a carry-save adder tree. (To accelerate the calculation of B, the least significant partial products in calculating powers of ΔY can be truncated for the higher-order terms since the higher-order terms are less significant (details in next subsection).)
5. Compute the new dividend as $X' = X - X_h * B * Y$

   Although this step appears to require two sequential multiplications to compute $X_h * B * Y$, in fact this is not true (see step 4 of basic algorithm).
6. Compute the new quotient $Q' = Q + X_h * B * (1/2^{(j-k)})$.

   It will be shown later in this paper that X_h in the formulas for Q' and X' can be composed of the leading $p = m * t - t + 2$ bits of X followed by 0's, where t is the number of terms used to calculate B. Also, B can be limited to $m * t - t + 4$ bits. This limits the size of the multiplies.
7. Left shift X' by $m * t - t - 1$ bits to get rid of guaranteed leading 0's that occur (as discussed in the next subsection). Shift zeroes into the LSB of X' during this process.

   Calculate a revised $j' = j + m * t - t - 1$.
8. Set $j = j'$, $Q = Q'$, and $X = X'$.
9. Repeat steps 5 through 8 until $j \geq q$.
10. The final Q is the quotient. Note that there are more bits of the quotient register than q. Let $Q_h = Q_q..Q_0$ and let $Q_l = Q_{-1}..Q_{-e}$ where e is the number of excess bits. The Q_h bits are correct to within one unit of the least significant bit. Some of the Q_l bits are not necessarily correct since another iteration of the algorithm would add into those bits.
11. The residual dividend X should be right-shifted by $j - q$ bits to get the remainder assuming the entire Q is the quotient. If Q is truncated to q bits, then the true remainder can be computed by then adding $Q_l * Y$ to X.

Since the advanced version reduces X' by $m*t-t-1$ bits per iteration, the algorithm terminates after $\lceil q/(m*t-t-1)\rceil$ iterations.

2.6 How Many Bits per Iteration are Guaranteed using the Advanced Method?

Theorem 2: Number of Bits Per Iteration Using the Advanced Method

X' is reduced by at least $m*t-t-1$ bits per iteration if

- *The term Y_h includes $m \geq 5$ leading bits of Y.*
- *The number of Taylor-series terms from equation 21 used to construct B is t.*
- *The number p of significant bits in X_h is equal to $m*t-t+2$.*
- *The word width b_i of Table G_i is $(m*t-t)+\lceil log_2 t\rceil-(m*i-m-i)$ for $i=1,2,..t$.*
- *B is represented in $b_b = m*t-t+4$ bits or more.*
- *The total arithmetic error in calculating B is less than $1/2^{(m*t-t+3)}$ (see below for explanation).*

Analysis

As in the basic method, m is the most important parameter, since an increase of 1 requires a doubling of the number of table words. Once m is chosen, the other parameters are set so that the total error from truncating other calculations is less than or equal to the error from including just the leading m bits of Y in Y_h. In this analysis, we show that this is true given the above parameter settings.

As before, we compute the worst-case number of bits that are reduced from X in each iteration. The initial value of X is between $1/2$ and 1 since the leading bit is 1. By determining the largest possible value of X', we can determine the worst-case number of bits eliminated per iteration.

Let us examine the formula for X':

$$X' = X - X_h * B * Y \tag{22}$$

where B is the first t terms from $B = 1/Y_h - \Delta Y/{Y_h}^2 + (\Delta Y)^2/{Y_h}^3 - (\Delta Y)^3/{Y_h}^4 + (\Delta Y)^4/{Y_h}^5 + ...$

In this case, three sources of inaccuracy affect the approximation $B \approx 1/Y$. Let the sources be defined by

$$B = 1/Y - R_b - R_c - R_d \tag{23}$$

where

- R_b represents an error term due to truncating the Taylor series after t terms. The error R_b is always non-negative since the truncated terms in the series are all non-negative.
- R_c represents the error in calculating B using lookup tables with finite width words. The error R_c is always non-negative since B is calculated using tables that are rounded down.
- R_d represents the error in truncating the arithmetic used to calculate B. The error R_d is non-negative since all the arithmetic is truncated, thus underestimating B.

Then

$$\begin{aligned} X' &= X - X_h * B * Y \\ &= X_h + \Delta X - X_h * (1/Y - R_b - R_c - R_d) * Y \end{aligned} \tag{24}$$

$$X' = \Delta X + X_h * (R_b + R_c + R_d) * Y \tag{25}$$

If R_b, R_c, and R_d can be accurately bounded, then this will give the maximal value of X'.

The $(w+1)$-th term of the Taylor-series for B is denoted by $B_{w+1} = (-\Delta Y)^w/{Y_h}^{(w+1)}$. All terms of B are non-negative since ΔY is non-positive. The worst-case occurs when $Y_h \approx 1/2$ and $-\Delta Y = 1/2^m - 1/2^q$. In this case, the following bound holds:

$$B_{w+1} < 1/2^{(m*w-w-1)} \tag{26}$$

If the first t terms are used to construct B, then the remainder R_b is bounded by:

$$R_b = \sum_{g=t}^{\infty} B_{(g+1)} < \sum_{g=t}^{\infty} 1/2^{(m*g-g-1)} = \frac{(1/2^{(m*t-t-1)})}{(1-1/2^{(m-1)})} \tag{27}$$

For $m \gg 1$, this is just slightly greater than $1/2^{(m*t-t-1)}$. For $m \geq 5$, a non-stringent bound on R_b is:

$$R_b < 1.1 * (1/2^{(m*t-t-1)}) \tag{28}$$

R_c comes from using finite-width lookup tables in calculating the value of the first t terms. R_d comes from truncating the arithmetic used to calculate B and truncating the value of B.

A cumulative error can be computed where the error in each table lookup is represented by ϵ_i and the truncation error in computing each multiplicative term is δ_i:

$$B = \sum_{i=1}^{t} [\delta_i + (\epsilon_i + 1/{Y_h}^i) * (-\Delta Y)^{(i-1)}] \tag{29}$$

$$B = \sum_{i=1}^{t} [\frac{(-\Delta Y)^{(i-1)}}{{Y_h}^i}] + \sum_{i=1}^{t} [(\epsilon_i) * (-\Delta Y)^{(i-1)}] + \sum_{i=1}^{t} \delta_i \tag{30}$$

Then R_c is equal to the sum of the terms involving ϵ_i:

$$R_c = \sum_{i=1}^{t} \epsilon_i * (-\Delta Y)^{(i-1)} \tag{31}$$

Let δ_0 be an additional term that represents truncating B to a certain number of bits after the summation. Then

$$R_d = \sum_{i=0}^{t} \delta_i \tag{32}$$

Let us focus first on R_c. Suppose table G_i is b_i bits wide. Since $1/2 < Y_h < 1$, the maximal value of $1/{Y_h}^i$ is slightly less than 2^i. If words in the table G_i can represent values up to but not including 2^i, the unit of the most significant binary digit in the table G_i should have value $2^{(i-1)}$. The unit of the least significant binary digit for table G_i is then $1/2^{(b_i-i)}$. Each ϵ_i is less than the unit of the least significant digit:

$$\epsilon_i < 1/2^{(b_i-i)} \tag{33}$$

The worst-case value of $-\Delta Y$ is $-\Delta Y = 1/2^m - 1/2^q$. Substituting into equation 31 yields:

$$R_c < \sum_{i=1}^{t} 1/2^{(b_i-i)} * (1/2^{(m*i-m)}) \tag{34}$$

$$R_c < \sum_{i=1}^{t} 1/2^{(b_i+m*i-m-i)} \tag{35}$$

This equation helps to determine the proper word widths b_i. Suppose that the b_i are set to:

$$b_i = (m*t-t) + \lceil log_2 t \rceil - (m*i-m-i) \tag{36}$$

As expected since the first tables have more significance, each table T_i is $m-1$ bits wider than the table $T_{(i+1)}$.

The total error R_c is then bounded by:

$$R_c < \sum_{i=1}^{t} 1/2^{((m*t-t)+\lceil log_2 t \rceil)} \tag{37}$$

$$R_c < 1/2^{(m*t-t)} \tag{38}$$

Now consider R_d. The δ_i represent maximal permissible truncation errors. This concept accelerates the calculation of B by allowing the arithmetic for calculating B to be reduced in size. For instance, suppose $t = 4$, $m = 16$, and $q = 53$. Then the last term of B is $B_4 = (-\Delta Y)^3 * (1/{Y_h}^4)$. Since ΔY has $q - m = 53 - 16 = 37$ bits, the full computation of $(-\Delta Y)^3$ would use large multipliers and result in a $37*3-2 = 109$ bit result. This is clearly much more than necessary because the unit of the most significant bit of B_4 will be $1/2^{(3*m-4)} = 1/2^{44}$. The allowable truncation errors δ_i can be used both to prune multiplier trees by discarding least significant partial products and to truncate results to smaller widths. The details are not presented here.

Since $1 \leq B < 2$, the MSB of B has unit 1. If B is truncated to b_b bits, then $\delta_0 <$ unit of LSB so $\delta_0 < 1/2^{(b_b-1)}$.

In this case, we wish to restrict R_d as follows:

$$R_d < 1/2^{(m*t-t+2)} \tag{39}$$

For the purposes of this paper, this can be achieved by both restricting δ_0 to

$$\delta_0 < 1/2^{(m*t-t+3)} \tag{40}$$

meaning that B can be truncated to $b_b = m*t-t+4$ bits, and restricting the remaining δ_i to

$$\sum_{i=1}^{t} \delta_i < 1/2^{(m*t-t+3)} \tag{41}$$

Now we can substitute the bounds for R_b, R_c, and R_d into constraint 25 to determine the maximum value of X'. Since we set $Y_h \approx 1/2$ previously, Y should also equal 1/2. Since $X_h < 1$ and $\Delta X \leq 1/2^p - 1/2^q$, the worst-case X' is bounded by:

$$X' = \Delta X + X_h * R_b * Y + X_h * R_c * Y + X_h * R_d * Y \tag{42}$$

$$\begin{aligned} X' \;<\; & 1/2^p - 1/2^q + 1.1 * (1/2^{(m*t-t-1)}) * 1/2 \\ & + (1/2^{(m*t-t)}) * 1/2 + (1/2^{(m*t-t+2)}) * 1/2 \end{aligned} \tag{43}$$

If $p = m*t-t+2$, then this can be converted to:

$$X' < 1/2^{(m*t-t+2)} - 1/2^q + 1.6 * (1/2^{(m*t-t)}) + 1/2^{(m*t-t+3)} \tag{44}$$

$$\begin{aligned} X' \;<\; & 0.25/2^{(m*t-t)} - 1/2^q + 1.6 * (1/2^{(m*t-t)}) \\ & + 0.125 * (1/2^{(m*t-t)}) \end{aligned} \tag{45}$$

$$X' < 1/2^{(m*t-t-1)} \tag{46}$$

If $p \geq m*t-t+2$, then the worst-case value of X' is bounded by the above inequality. In this case, the highest-order bit of X' that could possibly be 1 is the $(m*t-t)$-th bit, $X'_{(q-m*t+t)}$. In fact, using $p > m*t-t+2$ only improves the convergence by a fraction of a bit, so it is best to use $p = m*t-t+2$ to minimize the size of the multiplies.

In summary, at least $m*t-t-1$ bits are skipped per iteration if $p = m*t-t+2$, t is the number of terms used to construct B, and the other conditions in Theorem 2 hold. (Note that this works for the basic method where $t = 1$, $p = m+1$, and the predicted minimum number of bits is $m-2$.)

3 Comparison to Other High-Radix Methods Using Reciprocals

In this section, our method is compared to the Cyrix short-reciprocal method, the Newton-Raphson technique, the MacLaurin series technique, and an implementation of a modified Newton-Raphson technique in the IBM RISC System/6000.

We refer to approximations of the reciprocal in all methods using B. In some cases, such as our basic method, B is the direct result of a table look-up. In other cases, B requires some calculations.

3.1 Cyrix Short-Reciprocal Algorithm

The recently designed Cyrix arithmetic coprocessor briefly described in [4] and [11] employs a "short reciprocal" algorithm similar to the basic technique described in this paper to get 17 bits/iteration. We only discriminate between the two methods here. Unlike our method, the Cyrix method derives its reciprocal approximation using a combination of table lookup and Newton-Raphson. The iterative portion of the Cyrix method uses an intentional overestimate of the reciprocal rather than an underestimate. In addition, the Cyrix method is implemented using a redundant number system while ours is non-redundant. The Cyrix coprocessor can also do square root using a similar iterative scheme. A fuller description of their work may be forthcoming.

3.2 Comparison to Newton-Raphson

In the standard Newton-Raphson technique [13], an accurate reciprocal B is computed first by an iterative method. This reciprocal is accurate enough so that the quotient can be calculated directly by a final multiplication $Q = X * B$. The calculation of the reciprocal approximation in Newton-Raphson is necessarily iterative: the improved reciprocal is calculated using $B_{(n+1)} = B_n * (2 - Y * B_n)$ [13] [10]. Each iteration requires two multiplications in $B_n * Y * B_n$ that cannot be performed simultaneously. Each iteration doubles the number of bits of accuracy in B; to get 56-bit accuracy requires 2 iterations starting with a 14-bit accurate reciprocal. A separate calculation is required to produce a remainder, which has been shown to be non-negative in [13].

In contrast, our method calculates the reciprocal B using one or more terms from a Taylor-series that can be implemented in a non-iterative manner. Our method iterates by calculating a reduced dividend $X' = X - (X * B) * Y$ where $X * B$ is the approximate quotient based on the approximate reciprocal B. Each iteration reduces the dividend X' by a constant number of bits which depends on the accuracy of B. Since estimates of the quotient are calculated simultaneously in each iteration, no final multiplication is needed to get the quotient Q. A form of remainder is available directly in the register X assuming that the entire Q is the quotient. If the quotient is truncated, then the truncated bits multiplied by Y should be added to X. The method is designed so that the remainder is never negative.

3.3 Comparison to IBM RISC System/6000

The IBM RISC System/6000 implements floating-point division using an iterative algorithm employing its multiply and add hardware [12] [10]. The algorithm is a modified version of the Newton-Raphson method. One enhancement is used: after a reciprocal is calculated, the method performs one pass of a reduction similar to our method's iteration. Using our notation, it is as follows:

1. $Q = X * B$
2. $X' = X - Q * Y$
3. $Q' = Q + X' * B$

Reference [10] describes the algorithm and detailed analyses for guaranteed correct rounding.

In addition to the general differences between Newton-Raphson and our method, the above reduction is done using a full precision quotient, i.e. $X' = X - Q * Y$ instead of $X' = X - X_h * B * Y$ used in our method.

3.4 Comparison to MacLaurin Series

The MacLaurin series method [13] uses an approximation to the reciprocal:

$$\begin{aligned} B &\approx 1/Y = 1/(Z+1) \\ &= (1-Z)*(1+Z^2)*(1+Z^4) \\ &\quad *(1+Z^8)*(1+Z^{16})+... \end{aligned} \tag{47}$$

The number of bits of accuracy doubles with each additional factor. In the normal iterative implementation, the first few terms are approximated using a lookup table. Each iteration calculates an additional factor from the previous using one multiplication and an add (or a 2's complement in [13]). An additional multiplication is required to multiply the factor with the current approximation B. This multiplication can overlap the calculation of the next factor, so that each iteration adding one factor takes just 1 multiplication plus a 2's complement. As discussed in [13], the convergence per iteration is mathematically equivalent to the convergence of the Newton-Raphson method where $B_0 = 1$. The quotient and remainder must each be calculated following the generation of B. Unlike our method or Newton-Raphson, the implementation sizes are somewhat fixed. If one implementation includes the first k multiplicative terms in a lookup table of 2^v words, the next larger implementation includes the first $k+1$ terms with table size about 2^{2*v}.

Like the Newton-Raphson method, the convergence of the MacLaurin series differs from our method. Also, our method automatically provides a quotient without extra calculations.

4 Brief Notes on Implementation

1. The look-up table(s) are indexed using the leading m bits of Y. Since the leading bit of Y is known to be 1 after normalization, it is not actually useful to use that bit as part of the index. Therefore, the actual table can be of size $2^{(m-1)}$ words instead of 2^m.
2. The look-up table words are always rounded down to fit into the number of bits in the word so that the estimated reciprocal is never more than the exact answer to $1/Y$.
3. In each iteration, the critical loops are the calculation of Q' and X'. These can be implemented using carry-save adder (Wallace) trees that perform a multiply-and-add or multiply-and-subtract function. The latter can be accomplished by using 2's complement numbers. The hardware should make a 1's complement of each row to be subtracted and add a fixed 1 to that row's LSB in the carry-save adder tree.

4. As shown in Appendix 1, the basic method requires three multipliers of size $(m+1) \times m$, $(m+1) \times q$, and $(m+1) \times (q+m)$. Also required are one subtraction of size $(q+2m-1)$ and one addition of size $\lfloor q/(m-2) \rfloor * (m-2) + 2m$ which can actually be merged with the carry-save adder trees for the multiplications.
5. For the advanced method, the minimum size of the additions and multiplications except those to calculate B are as follows:
   - Note that Y has q bits, X_h has $p = m*t-t+2$ bits (rest are 0's), and B has $b_b = m*t-t+4$ bits.
   - Three multipliers are needed of size $p \times b_b$, $b_b \times q$, and $b_b \times (q+b_b-1)$. Also required are one subtraction of size $(q+p+b_b-2)$ and one addition of size $\lfloor (q/(m*t-t-1)) \rfloor * (m*t-t-1) + p + b_b - 1$ which can actually be merged with the carry-save adder trees for the multiplications.

5 Comparative Speed Analysis

In this section, three schemes are analyzed for practical high-speed implementation to demonstrate why the above technique is fast. The other two techniques are ordinary 1 bit or 2 bit/iteration division and Newton-Raphson division.

The charter of our research project is to focus on very fast implementations of arithmetic functions using maximum parallelism and dense, state-of-the-art technologies [7]. For this analysis, we assume a fast hypothetical technology whose characteristics are similar to a 1991 VLSI ECL technology:

1. Simple gate delays are 250 ps.
2. In the worst-case, complex AND/OR gate structures take almost twice as long as simple gates in ECL and are assumed to be 500 ps.
3. The assumed delay for the registers required for iteration is 500 ps.
4. One-way communication time between a logic chip and a RAM chip is .75 ns.
5. Static RAM access takes 3 ns for the RAM and 1.5 ns for a round-trip communication with the division logic chip for RAMs up to 64K x n bits. A RAM access time of 3 ns was demonstrated in a 1988 paper on an experimental 64-Kbit RAM [15].
6. An addition of 64 bits takes 2 ns. This is based on another 1988 paper on a 32-bit ECL adder [2]. Smaller additions take slightly less time, and larger ones take slightly more time.
7. A 53x53 multiplication takes about 6 ns in ECL [3]. Smaller multiplications take less time; for instance, a 32x32 bit multiplication takes 5 ns.
8. A 53x53 multiply plus an add takes 6 ns. A multiply and add takes no more time than a multiply if the add function is incorporated into the carry-save adder tree.
9. Clock skew effects as discussed for example in [8] and [14] are not included in this analysis.

We analyze for critical path delay assuming a fully-parallel implementation, i.e. all operations that can be made independent are performed in parallel.

In this case, we calculate the extra delay to compute the remainder as a separate number. The remainder will be relative to the quotient truncated to q bits. In our method, the previously-discussed correction to the remainder is required to adjust for this truncation (see last step of the algorithm). In Newton-Raphson, the remainder must be computed after the quotient. Since $Q = X * B$ provides about $2 * q$ bits, a simultaneous computation of the remainder and quotient is not possible. The simultaneous equation for the remainder $X' = B*(X*Y)$ is relative to an untruncated quotient. A remainder for a truncated quotient requires either a correction like the one in our method or a sequential calculation of quotient and remainder. The time consumed is about the same.

The details of the speed analysis are presented in Appendix 2 for some example implementations.

Table 1 summarizes the division schemes and their projected speeds for 53-bit division using the given technology assumptions. Extra delays are cited separately for computing a true remainder needed for round-to-nearest calculations. The amount of table storage required is also given in kilobits. A coarse comparison of logic size can be made using the total number of partial product bits needed for the major multipliers.

Some observations are as follows. The Newton-Raphson technique with a 7-bit accurate lookup table uses fewer storage bits than our basic method with $m = 11$ but still requires large multipliers. The speeds of the two are about equal. The fastest implementations are our advanced method using $m = 11$ and $m = 15$ with 2 Taylor-series terms. Using more than 2 terms to approximate B is slightly faster but uses much larger tables than just using 2 terms. However, the number of product bits remains moderate since the iteration multipliers are not needed; only truncated multipliers to calculate B and plus one 53x53 multiplier for the quotient plus remainder are needed.

For comparison, the Cray-2, first delivered in 1986, performs a floating-point division in 152 ns using MSI ECL circuits [6].

6 Summary and Conclusions

By using large, accurate lookup tables for the reciprocal, division can be accomplished in just a few iterations using the basic method. The keys are to design the method to underestimate the quotient and to adjust the quotient by adding rather than appending low-order bits. Using adding, the design is freed from having to produce disjoint sets of quotient bits in each iteration. In addition, a more accurate reciprocal can be quickly calculated using a Taylor-series approximation. Theoretically, the Taylor-series can be calculated directly rather than iteratively, unlike Newton-Raphson. In practice, the higher powers of ΔY may be calculated with staged multiplications, although lookup tables for powers of ΔY remains an unexplored possibility. By using more accurate reciprocals, the advanced version can divide in 2 or even 1 iteration.

Method	Est. Delay	Remainder	Table Size	Product Bits
Canonical 1-bit	159 ns	0 ns	0	53
Radix-4 SRT with 4 levels cascaded/iter	76 ns	0 ns	0	212
Newton-Raphson using 7-bit Initial	35.5 ns	6 ns	896 bits	≈ 4614
Newton-Raphson using 14-bit Initial	30 ns	6 ns	224K bits	≈ 4536
Basic Method using $m = 11$	36.0 ns	3.5 ns	12K bits	1536
Basic Method using $m = 13$	32.0 ns	3.75 ns	56K bits	1848
Basic Method using $m = 16$	27.5 ns	4 ns	544K bits	2346
Advanced Method using $m = 11$ and $t = 2$	26.5 ns	4.5 ns	34K bits	3888
Advanced Method using $m = 15$ and $t = 2$	22 ns	5 ns	736K bits	5952
Advanced Method using $m = 15$ and $t = 4$	20.5 ns	6 ns	2432K bits	≈ 4056

Table 1: Table of Relative Speeds and Required Storage and Multiplier Sizes

Implementations of our method can be faster than Newton-Raphson in modern ECL technology. The advanced method using $m = 11$ and 2 Taylor-series terms is both faster and smaller than a fast Newton-Raphson implementation. The advanced version using 2 Taylor-series terms and $m = 15$ is another 15% faster than using $m = 11$ at the cost of larger tables and multipliers. In a hypothetical implementation in ECL, the advanced method can divide 53-bit numbers in an estimated 22 ns vs. 6 ns for a multiply, thus achieving a ratio between divide and multiply times of less than 4:1. The algorithm can produce an exact remainder, which makes the implementation of exact rounding specifications (e.g. IEEE floating-point) straightforward. Both Newton-Raphson and this technique are substantially faster than 1-bit or 2-bit/iteration schemes. The logic depth of each iteration is more than in 1-bit or 2-bit schemes, so the speedup is less than proportional to the ratio of bits per iteration.

7 Acknowledgements

Thanks to the reviewers, David Matula, and Tien-Chi Chen for providing many helpful comments and additional references.

Thanks to the entire Nano-Second Arithmetic Research Group at Stanford for many enlightening discussions and seminars. Nhon Quach and Eric Schwarz carefully reviewed drafts of this paper and made many helpful comments.

References

[1] D. Atkins. "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders." Oct. 1968, IEEE Transactions on Computers, pp. 925-934.

[2] G. Bewick, P. Song, G. De Micheli, and M. Flynn. "Approaching a Nanosecond: A 32 bit Adder." 1988 International Conference on Computer Design, pp. 221-226.

[3] G. Bewick. Private communication in October 1990 about work in progress. Computer Systems Laboratory, Stanford University, Stanford, CA.

[4] T. Brightman. Slides from presentation on Cyrix coprocessor. 1989, future Trends panel session of 9th Symposium on Computer Arithmetic.

[5] J. Fandrianto. "Algorithm for High Speed Shared Radix 8 Division and Radix 8 Square Root." Sept. 1989, 9th Symposium on Computer Arithmetic, pp. 68-75.

[6] M. Flynn. "Sub-nanosecond Arithmetic Proposal." 1989, unpublished report, Computer Systems Laboratory, Stanford University, Stanford, CA.

[7] M. Flynn et al. "Sub-nanosecond Arithmetic". May 1990, Technical Report CSL-TR-90-428, Computer Systems Laboratory, Stanford University, Stanford, CA.

[8] L. Glasser and D. Dobberpuhl. The Design and Analysis of VLSI Circuits, Chapter 6. 1985, Addison-Wesley, Reading, Massachusetts.

[9] E. Krishnamurthy. "On Range-Transformation Techniques for Division." February 1970, IEEE Transactions on Computers, pp. 157-160.

[10] P. Markstein. "Computation of Elementary Functions on the IBM RISC System/6000 Processor." January 1990, IBM Journal of Research and Development, vol. 34, no. 1, pp. 111-119.

[11] D. Matula. "Highly Parallel Divide and Square Root Algorithms for a New Generation Floating Point Processor." Oct. 1989, extended abstract from SCAN-89 Symposium on Computer Arithmetic and Self-Validating Numerical Methods.

[12] R. Montoye, E. Hokenek, and S. Runyon. "Design of the IBM RISC System/6000 floating-point execution unit." January 1990, IBM Journal of Research and Development, vol. 34, no. 1, pp. 59-70.

[13] S. Waser and M. Flynn. Introduction to Arithmetic for Digital Systems Designers, Chapter 5. 1982; Holt, Rinehart, and Winston; New York, New York.

[14] D. Wong, G. De Micheli, and M. Flynn. "Designing High-Performance Digital Circuits Using Wave Pipelining." Aug. 1989, VLSI '89 Conference, pp. 241-252.

[15] K. Yamaguchi, H. Nanbu, K. Kanetani, et al. "An Experimental Soft-error Immune 64-Kb 3ns ECL Bipolar RAM." 1988 Bipolar Circuits and Technology Meeting, pp. 26-27.

8 Appendix 1: The Size of Needed Multipliers and Dividers

For the basic method, the minimum size of the additions and multiplications are as follows:

- Note that Y has q bits, X_h has $p = m + 1$ bits (rest are 0's), and $1/Y_h$ has $b_1 = m$ bits.
- In $X' = X - X_h * (1/Y_h) * Y$:

  In the first iteration, $X_h * Y$ is calculated first using a $(m+1) \times q$ bit multiply giving a $q+m$ bit result. This is then multiplied by $(1/Y_h)$ using an $m \times (q + m)$ bit multiply to get a $(q + 2m - 1)$ bit result. The subtraction from X is done using a $(q + 2m - 1)$ bit subtract yielding a $(q + m + 1)$ bit result after the shift of $m - 2$ bits.

  The quantity $(1/Y_h) * Y$ is also calculated during the first iteration using a $m \times q$ bit multiply yielding a $(q + m - 1)$ bit result. In subsequent iterations, $X_h * ((1/Y_h) * Y)$ is computed using a $(m+1) \times (q+m-1)$ bit multiply yielding a $(q + 2m - 1)$ bit result. The subtraction is the same size $(q+2m-1)$ bits yielding a $(q + m + 1)$ bit result.

  With some input MUX'ing, it is possible to use two multipliers total of sizes $(m + 1) \times q$ and $(m + 1) \times (q + m)$.
- In $Q' = Q + X_h * (1/Y_h) * (1/2^{(j-k)})$, the actual multiplication is $X_h * (1/Y_h)$ which is $(m + 1) \times m$ bits yielding a $2m$ bit result. The add increases in size with each iteration. In the last iteration, it is about $\lfloor q/(m-2) \rfloor * (m-2) + 2m$ bits long.
- In summary, three multipliers are needed of size $(m+1) \times m$, $(m + 1) \times q$, and $(m + 1) \times (q + m)$. Some MUX'es are needed to select the inputs to the latter two. Also required are one subtraction of size $(q + 2m-1)$ and one addition of size $\lfloor q/(m-2) \rfloor * (m-2) + 2m$ which can actually be merged with the carry-save adder trees for the multiplications.

The calculation of sizes for the advanced method is similar. The minimum size of the additions and multiplications except those to calculate B are as follows:

- Note that Y has q bits, X_h has $p = m * t - t + 2$ bits (rest are 0's), and B has $b_b = m * t - t + 4$ bits.
- In $X' = X - X_h * B * Y$:

  In the first iteration, $X_h * Y$ is calculated first using a $p \times q$ bit multiply giving a $q+p-1$ bit result. This is then multiplied by B using an $b_b \times (q + p - 1)$ bit multiply to get a $(q + p + b_b - 2)$ bit result. The subtraction from X is done using a $(q + p + b_b - 2)$ bit subtract yielding a $(q + p + 3)$ bit result after the shift of $m * t - t - 1$ bits.

  The quantity $B * Y$ is also calculated during the first iteration using a $b_b \times q$ bit multiply yielding a $(q+b_b-1)$ bit result. In subsequent iterations, $X_h * (B * Y)$ is computed using a $p \times (q+b_b-1)$ bit multiply yielding a $(q + p + b_b - 2)$ bit result. The subtraction is the same size $(q + p + b_b - 2)$ bits yielding a $(q + p + 3)$ bit result.

  With some input MUX'ing, it is possible to use two multipliers total of size $b_b \times q$ and $b_b \times (q + b_b - 1)$.
- In $Q' = Q + X_h * B * (1/2^{(j-k)})$, the actual multiplication is $X_h * B$ which is $p \times b_b$ bits yielding a $p + b_b - 1$ bit result. The add increases in size with each iteration. In the last iteration, it is about $\lfloor (q/(m * t - t - 1)) \rfloor * (m * t - t - 1) + p + b_b - 1$ bits long.
- In summary, three multipliers are needed of size $p \times b_b$, $b_b \times q$, and $b_b \times (q + b_b - 1)$. Some MUX'es are needed to select the inputs to the latter two. Also required are one subtraction of size $(q + p + b_b - 2)$ and one addition of size $\lfloor (q/(m * t - t - 1)) \rfloor * (m * t - t - 1) + p + b_b - 1$ which can actually be merged with the carry-save adder trees for the multiplications.

9 Appendix 2: Speed and Size Calculations

9.1 Speed of the Basic Technique

9.1.1 11 bits/iteration

Set $m = 13$.

The look-up table has size $2^{12} = 4$K words. The table width is $p = m + 1 = 14$ bits, so the table size is 56K bits.

The number of iterations required is $\lceil 53/(m - 2) \rceil = \lceil 53/11 \rceil = 5$.

The initial RAM lookup takes 4.5 ns.

The time per iteration is limited by the loop to compute the remaining dividend X'. For X', the critical path can be implemented using a MUX, a $(m + 1) \times (q + m) =$ 14x66 multiply tree, and a $(q + 2m - 1) = 78$-bit final add. The MUX takes 0.5 ns. The multiply tree takes about 2.25 ns to do the partial products generation and reduction. The addition takes 2.25 ns. Finally, iteration registers are assumed to take 0.5 ns.

The loop time is then $0.5 + 2.25 + 2.25 + .5 = 5.5$ ns.

The total time is then $4.5 + 5.5 * 5 = 32.0$ ns.

The final shift of X' can be done with wiring and takes no extra time. The remainder for the truncated quotient can be computed using a multiply of about 14x53 and an add which takes about an extra 3.75 ns.

The calculations for 14 bits/iteration using $m = 16$ and 9 bits/iteration using $m = 11$ are similar.

9.2 Speed of the Advanced Technique

9.2.1 27 bits/iteration using 2 Terms

Set $m = 15$.

Use 2 terms of the approximation to get B.

The look-up tables have size $2^{14} = 16K$ words. Table G_i has width $m*t-t+\lceil \log_2 t \rceil - (m*i-m-i) = 44-14*i$ bits. Table G_1 has width 30 bits for a size of 480K bits. Table G_2 has width 16 bits for a size of 256K bits.

The number of iterations required is $\lceil 53/(m*t-t-1) \rceil = \lceil 53/27 \rceil = 2$.

The initial RAM lookup takes 4.5 ns.

The computation of B takes 4.5 ns using some multiply and add hardware. The multiplication of $\Delta Y * (1/(Y_h)^2)$ is 16x38 bits. The addition is about 50 bits wide.

The time per iteration is limited by the loop to compute the remaining dividend X'. For X', the critical path can be implemented using a MUX, a $b_b \times (q+b_b-1) = 32$x84 multiply tree, and a $(q+p+b_b-2) = 113$-bit final add. The MUX takes 0.5 ns. The multiply tree takes about 3 ns to do the partial products generation and reduction. The addition takes 2.5 ns. Finally, iteration registers are assumed to take 0.5 ns.

The loop time is then $0.5+3+2.5+0.5 = 6.5$ ns.

The total time is then $4.5+4.5+2*6.5 = 22$ ns.

The final shift of X' can be done with fixed wiring and takes no extra time. The remainder for the truncated quotient can be computed in about an extra 5 ns using about a 32x53 multiply and add.

The calculation for 19 bits/iteration using $m = 11$ and $t = 2$ is similar.

9.2.2 55 bits/iteration using 4 Terms

Set $m = 15$.

Use 4 terms of the approximation to get B. The accuracy of B is about 55 bits.

The look-up tables have size $2^{14} = 16K$ words. Tables $G1$, $G2$, $G3$, and $G4$ have widths 59, 45, 31, and 17 bits respectively. This means that tables $G1$ through $G4$ have sizes of 944K, 720K, 496K, and 272K bits, respectively.

The number of iterations required is $\lceil 53/(m*t-t-1) \rceil = \lceil 53/55 \rceil = 1$.

The initial RAM lookup takes 4.5 ns but is not on the critical path, as explained below.

The computation of B takes 14 ns using some multiply and add hardware. The slowest term is $(\Delta Y)^3 * (1/(Y_h)^4)$ which takes 3 multiplications, 2 to get $(\Delta Y)^3$ and 1 more to get the term. This takes 8 ns for the first two multiplications. Since 8 ns exceeds the lookup time for the corresponding multiplicative factor $1/{Y_h}^4$, the RAM lookup is not part of the critical path. The final multiplication is combined with the addition of all the terms and is estimated to take 6 ns.

The computation of Q' requires a full 53x53 multiplication which takes 6 ns. Finally, storage registers are assumed to take 0.5 ns. The time for this is then $6+0.5 = 6.5$ ns.

The total time is then $14+6.5 = 20.5$ ns.

In this case, the true remainder is most easily calculated using $X' = X - Q_h * B$ where Q_h is the truncated quotient. This requires a 53x53 multiply and an add which takes about 6 ns. (The correction method is no faster.)

Due to the increase in complexity for calculating B and the larger multiplication/additions needed in calculating Q' and X', this method is not much faster than using only 2 terms to approximate B even though the number of iterations is cut in half.

9.3 Speed of ordinary 1-bit iteration division

In each iteration of ordinary division, the divisor is subtracted from the dividend, the result is checked for negativity, the new dividend is selected, and the results are stored in the iteration registers.

The subtraction takes 2 ns. The negativity check is trivial since the high-order bit of the result can be directly used as the control to the MUX selecting the new dividend. The MUX requires 2 gate levels for a delay of .5 ns. The registers take .5 ns.

The total delay per iteration is $2.5+.5 = 3$ ns. The number of iterations is 53 for a total delay of $53*3 = 159$ ns.

In this method, the remainder is available in the dividend register following the computation.

9.4 Speed of Newton-Raphson division using 14-bit Table Lookup

The initial table lookup of Newton-Raphson takes 4.5 ns. The table is about 16K words of 14 bits = 224K bits.

Each iteration of Newton-Raphson requires 1 multiply followed by a second multiply and add. The first iteration requires a 14x14 and a truncated 14x28 multiply; the second requires a 28x28 and a truncated 28x56 multiply.

In the 1st and 2nd iterations, multiplies take 4 and 5 ns respectively. Storage into the iteration registers takes 0.5 ns.

The first iteration takes $4+4+0.5 = 8.5$ ns. The second takes $5+5+0.5 = 10.5$ ns.

Following two iterations, a 56-bit accurate reciprocal is available in a register that we denote by B.

The iterations are followed by a 53x56 bit multiplication to get the quotient which takes 6 ns. A final register storage takes 0.5 ns.

The total delay is then about $4.5+8.5+10.5+6.5 = 30$ ns.

The remainder for the truncated quotient Q_h can be computed using $X' = X - Q_h * Y$. This requires a 53x53 bit multiply and subtract which takes 6 ns.

Table 1 cites multiplier sizes assuming that a dedicated multiplier is used for each iteration. Alternatively, one 53x56 multiplier would be sufficient but all multiplies would take 6 ns thus adding a total of 6 ns to the propagation delay.