FAST IEEE ROUNDING FOR DIVISION BY FUNCTIONAL ITERATION

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-96-700

July 1996

This work was supported by NSF under contract MIP93-13701.

FAST IEEE ROUNDING FOR DIVISION BY FUNCTIONAL ITERATION

by

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-96-700

July 1996

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-9040

pubs@shasta.stanford.edu

Abstract

A class of high performance division algorithms is functional iteration. Division by functional iteration uses multiplication as the fundamental operator. The main advantage of division by functional iteration is quadratic convergence to the quotient. However, unlike non-restoring division algorithms such as SRT division, functional iteration does not directly provide a final remainder. This makes fast and exact rounding difficult. This paper clarifies the methodology for correct IEEE compliant rounding for quadratically-converging division algorithms. It proposes an extension to previously reported techniques of using extended precision in the computation to reduce the frequency of back multiplications required to obtain the final remainder. Further, a technique applicable to all IEEE rounding modes is presented which replaces the final subtraction for remainder computation with very simple combinational logic.

Key Words and Phrases: Division, Goldschmidt's algorithm, IEEE rounding, Newton-Raphson, variable latency

Contents

List of Figures

List of Tables

1 Introduction

In recent years computer applications have increased in their computational complexity. High speed floating-point hardware is a requirement to meet these increasing demands. An important component of the floating point unit is the divider. There are many methods for designing division hardware. These include linear converging algorithms, the most common of which is SRT, and quadratically converging algorithms, such as Newton-Raphson and Goldschmidt's algorithm [7]. Linear converging algorithms retire a fixed number of quotient digits in each iteration. After each iteration, a partial remainder is available. At the conclusion of the iterations, the quotient is available, as is the final remainder. By noting the sign and magnitude of the final remainder, it is possible to adjust the quotient appropriately by 1 unit in the last place (ulp) to obtain an exactly rounded result that complies with the IEEE floating-point standard [5]. In contrast, both Newton-Raphson and Goldschmidt's algorithms produce a quotient, but with no final remainder. For exact rounding of the quotient, it is typically necessary to use an additional multiplication of the quotient and the divisor and then to subtract the product from the dividend to form the final remainder. Accordingly, quadratically-converging algorithms can incur a latency penalty of one multiplication and a subtraction in order to produce IEEE exactly rounded quotients.

Previous implementations of quadratically-converging dividers have demonstrated various techniques of achieving close-to-exact rounding as well as exact rounding. However, all implementations yielding exactly rounded quotients have suffered from a rounding penalty. In this paper, an extension of a technique presented by Schwarz [9] is proposed which further reduces the frequency of final remainder calculations required by increasing the precision of the quotient. For those cases where a final remainder calculation is required, a technique is proposed which reduces the full-width subtraction to combinational logic operating on one bit of the dividend, one bit of the back multiplication product, and the sticky bit from the multiplier.

The remainder of this paper is organized as follows: Section 2 describes the principals of IEEE rounding. Section 3 reviews the theory of division by functional iteration. Section 4 presents previously implemented techniques for exact rounding. Section 5 presents the methodology for reducing the frequency of remainder computations. Section 6 presents the technique for fast magnitude comparison of the dividend and the back multiplication product. Section 7 is the conclusion.

2 IEEE Rounding

The IEEE 754 standard for floating-point representation describes two different formats: single and double precision. The standard also suggests the use of extended precision formats, but their use is not mandatory. The most common format used in modern processors is double precision, which comprises a 1-bit sign, an 11-bit biased exponent, and a 52-bit significand with one hidden significand bit, for a total of a 64 bits. A significand is a normalized number M, such that $1 \leq M < 2$. The standard includes four rounding modes: RN, RZ, RM, and RP. RN is unbiased rounding to nearest, rounding to even in the case of a tie.

RZ is simple truncation. The two directed rounding modes RM and RP are round towards minus infinity and round towards plus infinity respectively. For IEEE compliance, exactly rounded results must be computable for all four rounding modes. The result generated by an operation according to any of the four rounding modes must be the machine number which is identical to an intermediate result that is correct to infinite precision and is then rounded according to the same rounding mode.

The significand immediately before rounding has the format as given in figure 1. In this

Figure 1: Significand format before rounding

figure, L is the LSB before rounding, G is the guard bit, and S is the sticky bit. The guard bit is one bit less significant than is strictly required by the precision of the format. The sticky bit is essentially a flag, noting the existence of any bits in the result less significant than the guard bit. It is the logical OR of all of the less significant bits in the result.

To implement rounding for each of the rounding modes, an action table listing the appropriate procedure for all combinations of L, G, and S can be written. An example action table for RN is shown in table 1. The rightmost column of the action table dictates

L	G	S	Action	A
X	0	0	Exact result. No rounding.	0
X	0	1	Inexact result, but is correctly rounded.	0
0	1	0	Tie case with even significand, so correctly rounded.	0
1	1	0	Tie case with odd significand, so round to nearest even.	1
X	1	1	Round to nearest.	1

Table 1: Action table for RN rounding mode

whether the result should be rounded. Rounding is accomplished by adding A to L to obtain the correct machine number. Such a table can be implemented simply in random logic. Similar tables can be written for the other three rounding modes.

For division, the complication is the determination of the sticky bit. This determination requires knowledge of the magnitude of the final remainder. Since division by functional iteration does not directly provide the remainder, the design challenge is how to gain the information in the remainder while incurring as minimal a latency penalty as possible. Before presenting these techniques, the theory of functional iteration is reviewed.

3 Division by Functional Iteration

Multiplicative division algorithms take advantage of high-speed multipliers to converge to a result quadratically. Rather than retiring a fixed number of quotients bits in every cycle as in subtractive division algorithms, multiplication-based algorithms are able to double the number of correct quotient bits in every iteration. However, the tradeoff between the two classes is not only latency in terms of the number of iterations, but also the length of each iteration in cycles. Additionally, if the divider shares an existing multiplier, the performance ramifications on regular multiplication operations must be considered. Oberman [8] reports that in typical floating-point applications, the performance degradation due to a shared multiplier is small. Accordingly, if area must be minimized, an existing multiplier may be shared with the division unit with only minimal system performance degradation. This section presents the algorithms used in multiplication-based division, both of which are related to the Newton-Raphson equation.

3.1 Newton-Raphson

Division can be written as the product of the dividend and the reciprocal of the divisor, or

$$Q = a/b = a \times (1/b),$$

where Q is the quotient, a is the dividend, and b is the divisor. In this case, the challenge becomes how to efficiently compute the reciprocal of the divisor. In the Newton-Raphson algorithm, a *priming function* is chosen which has a root at the reciprocal [3]. In general, there are many root targets that could be used, including $\frac{1}{b}$, $\frac{1}{b^2}$, $\frac{a}{b}$, and $1 - \frac{1}{b}$. The choice of which root target to use is arbitrary. The selection is made based on convenience of the iterative form, its convergence rate, its lack of divisions, and the overhead involved when using a target root other than the true quotient.

The most widely used target root is the divisor reciprocal $\frac{1}{b}$, which is the root of the priming function

$$f(X) = 1/X - b = 0. \tag{1}$$

The well-known quadratically converging Newton-Raphson equation is given by:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2}$$

The Newton-Raphson equation of (2) is then applied to (1). The function and its first derivative are evaluated at X_0:

$$\begin{aligned} f(X_0) &= 1/X_0 - b \\ f'(X_0) &= -1/X_0^2. \end{aligned}$$

These results are then used to find an approximation to the reciprocal:

$$X_1 = X_0 - \frac{f(X_0)}{f'(X_0)}$$

$$
\begin{aligned}
X_1 &= X_0 + \frac{(1/X_0 - b)}{(1/X_0^2)} \\
X_1 &= X_0 \times (2 - b \times X_0) \qquad (3) \\
&\vdots \\
X_{i+1} &= X_i \times (2 - b \times X_i) \qquad (4)
\end{aligned}
$$

The corresponding error term is given by

$$\epsilon_{i+1} = \epsilon_i^2(b),$$

and thus the error in the reciprocal decreases quadratically after each iteration. As can be seen from the general relationship expressed in (4), each iteration involves two multiplications and a subtraction. The subtraction is equivalent to the two's complement operation and is commonly replaced by it. Thus, two dependent multiplications and one two's complement operation are performed each iteration. The final quotient is obtained by multiplying the computed reciprocal with the dividend.

It can be seen that the number of operations per iteration and their order are intrinsic to the iterations themselves. However, the number of iterations required to obtain the reciprocal accurate to a particular number of bits is a function of the accuracy of the initial approximation X_0. By using a more accurate starting approximation, the total number of iterations required can be reduced. To achieve 53 bits of precision for the final reciprocal starting with only 1 bit, the algorithm will require 6 iterations:

$$1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 53$$

By using a more accurate starting approximation, for example 8 bits, the latency can be reduced to 3 iterations. By using at least 14 bits, the latency could be further reduced to only 2 iterations.

3.2 Series Expansion

A different method of deriving a division iteration is based on a series expansion. A name sometimes given to this method is *Goldschmidt's algorithm*. Consider the familiar Taylor series expansion of a function $g(y)$ at point a p,

$$g(y) = g(p) + (y-p)g'(p) + \frac{(y-p)^2}{2!}g''(p) + \cdots + \frac{(y-p)^n}{n!}g^{(n)}(p) + \cdots.$$

In the case of division, it is desired to find the expansion of the reciprocal of the divisor, such that

$$q = \frac{a}{b} = a \times g(y),$$

where $g(y)$ can be computed by an efficient iterative method. A straightforward approach might be to choose $g(y)$ equal to $1/y$ with $p = 1$, and then to evaluate the series. However, it is computationally easier to let $g(y) = 1/(1+y)$ with $p = 0$, which is just the Maclaurin series. Then, the function is

$$g(y) = \frac{1}{1+y} = 1 - y + y^2 - y^3 + y^4 - \cdots.$$

So that $g(y)$ is equal to 1/b, the substitution $y = b - 1$ must be made, where b is bit normalized such that $0.5 \leq b < 1$, and thus $|Y| \leq 0.5$. Then, the quotient can be written as

$$q = a \times \frac{1}{1+(b-1)} = a \times \frac{1}{1+y} = a \times (1 - y + y^2 - y^3 + \cdots)$$

which, in factored form, can be written as

$$q = a \times [(1-y)(1+y^2)(1+y^4)(1+y^8)\cdots]. \tag{5}$$

This expansion can be implemented iteratively as follows. An approximate quotient can be written as

$$q_i = \frac{N_i}{D_i}$$

where N_i and D_i are iterative refinements of the numerator and denominator after step i of the algorithm. By forcing D_i to converge toward 1, N_i converges toward q. Effectively, each iteration of the algorithm provides a correction term $(1 + y^{2i})$ to the quotient, generating the expansion of (5).

Initially, let $N_0 = a$ and $D_0 = b$. To reduce the number of iterations, a and b should both be prescaled by a more accurate approximation of the reciprocal, and then the algorithm should be run on the scaled a' and b'. For the first iteration, let $N_1 = R_0 \times N_0$ and $D_1 = R_0 \times D_0$, where $R_0 = 1 - y = 2 - b$, or simply the two's complement of the divisor. Then,

$$D_1 = D_0 \times R_0 = b \times (1 - y) = (1 + y)(1 - y) = 1 - y^2.$$

Similarly,

$$N_1 = N_0 \times R_0 = a \times (1 - y).$$

For the next iteration, let $R_1 = 2 - D_1$, the two's complement of the new denominator. From this,

$$\begin{aligned} R_1 &= 2 - D_1 = 2 - (1 - y^2) = 1 + y^2 \\ N_2 &= N_1 \times R_1 = a \times [(1-y)(1+y^2)] \\ D_2 &= D_1 \times R_1 = (1-y^2)(1+y^2) = (1-y^4) \end{aligned}$$

Continuing, a general relationship can be developed, such that each step of the iteration involves two multiplications

$$N_{i+1} = N_i \times R_i \quad \text{and} \quad D_{i+1} = D_i \times R_i$$

and a two's complement operation,

$$R_{i+1} = 2 - D_{i+1}$$

After i steps,

$$N_i = a \times [(1-y)(1+y^2)(1+y^4)\cdots(1+y^{2i})] \tag{6}$$
$$D_i = (1-y^{2i}) \tag{7}$$

Accordingly, N converges quadratically toward q and D converges toward 1. This can be seen in the similarity between the formation of N_i in (6) and the series expansion of q in (5). So long as b is normalized in the range $0.5 \leq b < 1$, then $y < 1$, each correction factor $(1 + y^{2i})$ doubles the precision of the quotient. This process continues as shown iteratively until the desired accuracy of q is obtained.

Consider the iterations for division. A comparison of equation (6) using the substitution $y = b - 1$ with equation (4) using $X_0 = 1$ shows that the results are identical iteration for iteration. Thus, the series expansion is mathematically identical to the Newton-Raphson iteration for $X_0 = 1$. Additionally, each algorithm can benefit from a more accurate starting approximation of the reciprocal of the divisor to reduce the number of required iterations. However, the implementations are not exactly the same. First, Newton-Raphson converges to a reciprocal, and then multiplies by the dividend to compute the quotient, whereas the series expansion first prescales the numerator and the denominator by the starting approximation and then converges directly to the quotient. Each iteration in both algorithms comprises two multiplications and a two's complement operation. From (4), it can be noted that the multiplications in Newton-Raphson are dependent operations. In the series expansion implementation, though, the two multiplications of the numerator and denominator are independent operations and may occur in parallel. As a result, the series expansion implementation can take advantage of an existing pipelined multiplier to obtain higher performance.

A performance enhancement that can be used for both algorithms is to perform early computations in reduced precision. This is acceptable, because the early computations do not generate many correct bits. As the iterations continue, quadratically larger amounts of precision are required in the computation. However, this has an effect on the precision of the final quotient approximation. Consider the series expansion algorithm. For n bit input operands, so long as all computations are at least n bits wide, then

$$D_{i+1} = 0.11 \cdots xxx$$

approaching 1 from below. Similarly N_{i+1} approaches the quotient from below. Accordingly the final n bit result can have at most a 1 ulp error which satisfies:

$$0 \leq \epsilon_t < 2^{-n}$$

and therefore the error in the final n bit result Q' is satisfied by:

$$0 \leq Q - Q' < 2^{-n} \tag{8}$$

where Q is the infinitely precise result. Should either of the two iteration products

$$D_{i+1} = D_i \times R_i$$

or

$$N_{i+1} = N_i \times R_i$$

be computed with multipliers only k bits wide, where $k < n$, it is possible that additional error will be induced into the approximation. If the multiplications and two's complement operation have only k bits of precision, then

$$R_{it} = 2 - D_{it}$$

This induces an error ϵ_r in D_{i+1} that satisfies

$$0 \leq \epsilon_r < 2^{-k}$$

Because of the resulting error in D_{i+1}, it will continue to converge towards 1, but it will converge from either below or above rather than from strictly below. Thus, if n bit operations are used for the final iteration while k bit operations are used for the intermediate operations, with $k < n$, then the final n bit result will satisfy

$$-2^{-n} < Q - Q' < 2^{-n} \tag{9}$$

4 Previously Implemented Techniques

There have been three main techniques used in previous implementations to compute rounded results when using division by functional iteration. The IBM 360/91 implemented division using Goldschmidt's algorithm [1]. In this implementation, 10 extra bits of precision in the quotient were computed. A hot-one was added in the LSB of the guard bits. If all of the 10 guard bits were ones, then the quotient was rounded up. This implementation had the advantage of the fastest achievable rounding, as it did not require any additional operations after the completion of the iterations. However, while the results could be considered "somewhat round-to-nearest," they were definitely not IEEE compliant. There was no concept of exact rounding in this implementation, as accuracy was sacrificed in favor of speed.

Another implemented method requires a datapath twice as wide as the final result, and it is the method used to implement division in the IBM RS/6000 [6]. The quotient is computed to a little more than twice the precision of the final quotient, and then the extended result is rounded to the final precision. An explanation of this procedure is as follows. Consider that the dividend X and the divisor Y are both normalized and represented by b bits, and the final quotient $Q = X/Y$ is represented by b bits. It must be first noted that the exact halfway quotient can not occur when dividing two b bit normalized numbers. For an exact halfway case, the quotient would be represented by exactly a $b+1$ bit number with both its MSB and LSB equal to 1, and thus having exactly $b - 1$ bits between its most significant and least significant 1's. The product of such a number with any non-zero finite binary number must also have the same property, and thus the dividend must have this property. But, the dividend is defined to be a normalized b bit number, and thus can it can have a maximum of $b - 2$ bits between its most significant and least significant 1's.

To obtain b significant bits of the quotient, b bits are computed if the first quotient bit is 1, and $b+1$ bits if the first quotient bit is 0. At this point, because the exact halfway case can not occur, rounding can proceed based solely on the values of the next quotient bit and the sticky bit. The sticky bit is 0 if the remainder at this point is exactly zero. If any bit of the remainder is 1, then the sticky bit is 1. Let R_0 be the value of the remainder after this computation, assuming the first bit is 1:

$$X = Q_0 \times Y + R_0, \quad \text{with } R_0 < 2^{-b}$$

Then, compute another b bits of quotient, denoted Q_1.

$$R_0 = Q_1 \times Y + R_1, \quad \text{with } R_1 < 2^{-2b}$$

Q_1 is less than 2^{-b}, with an accuracy of 2^{-2b}, and Y is normalized to be accurate to 2^{-b}. Accordingly if $Q_1 = 0$, then $R_0 = R_1$. But, R_0 can equal R_1 if and only if $R_0 = R_1 = 0$. This is because $R_0 < 2^{-b}$ and $R_1 < 2^{-2b}$ and Y is a b bit quantity. Similarly, if $Q_1 \neq 0$, then the remainder R_0 can not equal 0. The computation proceeds in the same manner if the first quotient bit is 0, except that $b+1$ bits will have been computed for Q_0. From this analysis, it is apparent that by computing at most $2b+1$ bits, the sticky bit can be determined, and the quotient can be correctly rounded.

The RS/6000 implementation uses its fused multiply-accumulate for all of the operations to guarantee accuracy greater than $2n$ bits throughout the iterations. After the completion of the additional iteration,

$$Q' = \text{estimate of } Q = \frac{a}{b} \text{ accurate to } 2n \text{ bits}$$

A remainder is calculated as

$$R = a - b \times Q' \tag{10}$$

A rounded quotient is then computed as

$$Q'' = Q' + R \times b \tag{11}$$

where the final multiply-accumulate is carried in the desired rounding mode, providing the exactly rounded result. The principal disadvantage of this method is that it requires one additional full iteration of the algorithm, and it requires a datapath at least two times larger than is required for non-rounded results.

The more common method for rounding is that which was used in the TI 8847 FPU [2, 4]. In this scheme, the quotient is also computed with some extra precision, but less than twice the desired final quotient width. To determine the sticky bit, the final remainder is directly computed from:

$$\begin{aligned} Q &= \frac{a}{b} - R \\ R &= a - b \times Q \end{aligned}$$

It is not necessary to compute the actual magnitude of the remainder; rather, its relationship to zero is the requirement. In the worst case, a full-width subtraction may be used to form the true remainder R. Assuming sufficient precision is used throughout the iterations such that all intermediate computations are at least n bits wide for n bit input operands, the computed quotient will be less than or equal to the infinitely precise quotient. Accordingly, the sticky bit is zero if the remainder is zero and one if it is nonzero. If truncated multiplications are used in the intermediate iterations, then the computed quotient will be within 1 ulp of the exactly rounded result, but it may be either above or below it. In this case, the sign of the remainder is also required to detect the position of the quotient estimate relative to the true quotient. Thus, to support exact rounding using this method, the latency of the algorithm increases by at least the multiplication delay necessary to form $Q \times b$, and possibly by a full-width subtraction delay as well as zero-detection and sign-detection logic on the final remainder. In the TI 8847, it is reported that the relationship of the quotient estimate and the true quotient is determined using combinational logic on 6 bits of both a and $Q \times b$ without explicit computation of R.

5 Reducing the Frequency of Remainder Computations

5.1 Basic Rounding

To simply the discussion and analysis of the rounding techniques throughout the rest of this paper, it is assumed that the input operands are normalized significands in the range [0.5,1), rather than the IEEE range of [1,2). The analysis is equivalent under both conditions and there is no loss of generality. Accordingly, 1 ulp for such a normalized n bit number is 2^{-n}.

The basic rounding technique is as follows. It is assumed that at least $n+2$ bit computations are used for n bit input operands. Sufficient iterations of the algorithm are then implemented such that the quotient is accurate to $n+1$ bits with an error strictly less than 1 ulp of this $n+1$ bit quantity. As the final result only has n bits, the quotient estimate has an error strictly less than +0.5 ulp, and this estimate satisfies:

$$0 \leq Q - Q' < 2^{-(n+1)} \tag{12}$$

The steps to correctly round this quotient estimate are:

- Add $2^{-(n+2)}$ to Q'.
- Truncate the transformed Q' to $n+1$ bits to form Q''. Q'' will then have strictly less than ± 0.5 ulp error.
- Form the remainder $R = a - b \times Q''$, which is an n bit by $(n+1)$ bit product.
- By observing the sign and magnitude of R and bit $n+1$, the guard bit, all IEEE rounding modes can be implemented by choosing either Q'', $Q'' + 2^{-n}$, or $Q'' - 2^{-n}$.

After the addition in the first step, Q' satisfies:

$$-2^{-(n+2)} \leq Q - (Q' + 2^{-(n+2)}) < 2^{-(n+2)} \tag{13}$$

Guard Bit	Remainder	RN	RP (+/-)	RM (+/-)	RZ
0	=0	**trunc**	trunc	trunc	trunc
0	-	**trunc**	trunc/dec	dec/trunc	dec
0	+	**trunc**	inc/trunc	trunc/inc	trunc
1	= 0	—	—	—	—
1	-	trunc	**inc/trunc**	**trunc/inc**	**trunc**
1	+	inc	**inc/trunc**	**trunc/inc**	**trunc**

Table 2: Action table for basic method

Truncation in the second step induces an error satisfying

$$0 \leq \epsilon_t < 2^{-(n+1)} \tag{14}$$

after which the result Q'' satisfies

$$-2^{-(n+2)} \leq Q - Q'' < 2^{-(n+1)} \tag{15}$$

Thus, the result can have an error of [-0.25,+0.5) ulp. This can be rewritten as a looser but equivalent error bound of (-0.5,+0.5) ulp. Accordingly, the observation of the guard bit and the sign and equality to zero of the remainder are sufficient to exactly round the quotient. The rounding in the last step is accomplished by conditionally incrementing or decrementing L. The action table for correctly rounding Q'' is shown in table 2. For the directed rounding modes RP and RM, the actual action may depend upon the sign of the quotient estimate. Those entries that contain two operations such as *pos/neg* are for the sign of the final result itself being positive and negative respectively. As discussed earlier, the exact halfway case can not occur in division, and thus the table row with $G = 1$ and $R = 0$ has no entries.

A similar methodology can be used should any of the intermediate iterations of the initial algorithm have been performed using truncated operations. Assuming that at least $n+2$ bits of quotient are computed with sufficient iterations to guarantee an accuracy of $n+1$ bits as before, then the estimate Q' can have at most a ± 0.5 ulp error, and it satisfies

$$-2^{-(n+1)} < Q - Q' < 2^{-(n+1)} \tag{16}$$

due to the convergence to the quotient from above or below. This is not sufficient precision for rounding using the guard bit. Instead, the estimate must be accurate to $n+2$ bits, requiring at least 2 additional bits, rather than 1, to be computed in the iterations. Then, the estimate satisfies

$$-2^{-(n+2)} < Q - Q' < 2^{-(n+2)} \tag{17}$$

In this case, the addition of $2^{-(n+2)}$ is again performed

$$-2^{-(n+1)} < Q - (Q' + 2^{-(n+2)}) < 0 \tag{18}$$

and after truncation to $n+1$ bits forms Q''

$$-2^{-(n+1)} < Q - Q'' < 2^{-(n+1)} \tag{19}$$

After these adjustments, rounding proceeds in the same manner as discussed previously using table 2.

A final possibility is that due to truncated operations in the iterations as well as the choice of initial approximation, the bounds on the error in the quotient estimate are inclusive rather than exclusive. Assuming as before that at least $n+2$ bits of quotient are computed with an accuracy of $n+2$ bits, then the estimate satisfies

$$-2^{-(n+2)} \le Q - Q' \le 2^{-(n+2)} \tag{20}$$

The addition of $2^{-(n+2)}$ to Q' yields

$$-2^{-(n+1)} \le Q - (Q' + 2^{-(n+2)}) \le 0 \tag{21}$$

After truncation to $n+1$ bits, the truncation error ϵ_t causes Q'' to satisfy

$$-2^{-(n+1)} \le Q - Q'' < 2^{-(n+1)} \tag{22}$$

Due to the lower inclusive point, it is not possible to round directly as before. To allow the same rounding methodology, it is necessary to force this bound to be exclusive rather than inclusive. To do this, it is necessary that the accuracy of the original quotient estimate Q' have more than $n+2$ bits of accuracy. As an example, if Q' has $n+3$ bits of accuracy using at least $n+3$ bits of quotient, then it will satisfy

$$-2^{-(n+3)} \le Q - Q' \le 2^{-(n+3)} \tag{23}$$

The addition of $2^{-(n+2)}$ to Q' forms

$$-2^{-(n+3)} - 2^{-(n+2)} \le Q - (Q' + 2^{-(n+2)}) \le 2^{-(n+3)} - 2^{-(n+2)} \tag{24}$$

and after truncation to $n+1$ bits forming Q''

$$-2^{-(n+3)} - 2^{-(n+2)} \le Q - Q'' < 2^{-(n+1)} \tag{25}$$

which clearly satisfies

$$-2^{-(n+1)} < Q - Q'' < 2^{-(n+1)} \tag{26}$$

after which rounding may proceed using table 2. This analysis shows that in this case it is necessary for the quotient estimate to have an accuracy of strictly greater than $n+2$ bits.

5.2 Faster Rounding

By observing the entries in table 2, it can be seen that for any of the rounding modes, half of the column's entries are identical. These identical entries are shown in bold type in the table. Accordingly, for all rounding modes, only half of the entries require knowledge of the remainder itself. As an example, for RN, if $G = 0$, then the correct action is truncation, regardless of the sign or magnitude of the remainder. However, if $G = 1$, then the sign of the remainder is needed to determine whether truncation or incrementing is the correct rounding action. Similarly, for RP, if $G = 1$, then the correct rounding action is to increment if the sign of the quotient is positive, and truncation if negative. Again, no computation of the remainder is required. These results are similar to those reported in [9].

Implementation of such a variable latency divider is as follows. The iterations for computing the quotient estimate Q' are carried to at least $n + 2$ bits assuming an error bound of

$$0 \leq Q - Q' < 2^{-(n+1)}$$

As previously discussed, other error bounds on the quotient estimate can be tolerated by employing more accuracy in the results. Thus, the technique presented can be easily modified to handle other error bounds. The quantity $2^{-(n+2)}$ is then added to Q'. This can be done either in a dedicated additional addition step, or, for higher performance, as part of a fused multiply-accumulate operation in the last iteration of the division algorithm. The guard bit G is then observed. Depending upon the rounding mode and the value of G, it may be possible to instigate rounding immediately. Otherwise, it may be necessary to perform the back-multiplication and subtraction to form the final remainder, and to observe its magnitude and sign in order to begin rounding. Thus, assuming a uniform distribution of quotients, in half of the cases a back-multiplication and subtraction is not required, reducing the total division latency. It should be noted that a dynamically-scheduled processor would be required to exploit this variable latency functional unit.

5.3 Higher Performance

The previously discussed technique requires the computation of one guard bit, and in so doing, allows for the removal of the back-multiplication and subtraction in stochastically half of the computations. This method can be extended as follows. Consider that at least $n + 3$ bits of quotient estimate are computed such that there are two guard bits with an error in this estimate of at most 1 ulp. This estimate Q' then satisfies:

$$0 \leq Q - Q' < 2^{-(n+2)} \tag{27}$$

The preliminary steps to correctly round this quotient estimate are similar to the previous technique:

- Add $2^{-(n+3)}$ to Q'.
- Truncate the transformed Q' to $n + 2$ bits to form Q''. Q'' will then have at most ± 0.25 ulp error.

Guard Bits	Remainder	RN	RP (+/-)	RM (+/-)	RZ
00	=0	trunc	trunc	trunc	trunc
00	-	trunc	**trunc/dec**	**dec/trunc**	**dec**
00	+	trunc	**inc/trunc**	**trunc/inc**	**trunc**
01	=0	trunc	inc/trunc	trunc/inc	trunc
01	-	trunc	inc/trunc	trunc/inc	trunc
01	+	trunc	inc/trunc	trunc/inc	trunc
10	= 0	—	—	—	—
10	-	**trunc**	inc/trunc	trunc/inc	trunc
10	+	**inc**	inc/trunc	trunc/inc	trunc
11	= 0	inc	inc/trunc	trunc/inc	trunc
11	-	inc	inc/trunc	trunc/inc	trunc
11	+	inc	inc/trunc	trunc/inc	trunc

Table 3: Action table using two guard bits

The action table for correctly rounding this quotient estimate Q'' is shown in table 3. From this table, it can be seen that for each rounding mode, only in 1 out of the 4 possible guard bit combinations is a back-multiplication and subtraction needed, as denoted by the bold-faced operations. In all of the other guard bit combinations, the guard bits themselves along with the sign of the final result are sufficient for exact rounding.

These results can be generalized to the use of m guard bits, with $m \geq 1$:

- Add $2^{-(n+m+1)}$ to Q'.
- Truncate the transformed Q' to $n + m$ bits to form Q''. Q'' will then have at most $\pm 2^{-m}$ ulp error.
- In parallel, observe the guard bits and begin computation of the remainder $R = a - b \times Q''$.
- If the guard bits are such that the sign and magnitude of the remainder are required, wait until the remainder is computed and then round. Otherwise, begin rounding immediately.

After the conversion in the first two steps, Q'' satisfies:

$$-2^{-(n+m)} < Q - Q'' < 2^{-(n+m)} \tag{28}$$

By inspecting the m guard bits, a back-multiplication and subtraction are required for only 2^{-m} of all cases. Specifically, the RN mode needs the computation to check the position around the mid-point between two machine numbers. The other three modes use the guard bits to check the position around one of the two machine numbers themselves. The examination of the m guard bits dictates the appropriate action in all of the other cases.

6 Faster Magnitude Comparison

For those cases where it is necessary to have information regarding the remainder, it is not necessary to perform the complete calculation of the remainder. Rather, as discussed previously, the essential elements are the sign of the remainder and whether the remainder is exactly zero. Stated slightly differently, what is required is whether the remainder is greater than, less than, or exactly equal to zero. The design challenge becomes how to compute this information in less time than that required for an n bit multiplication, subtraction, and subsequent zero-detection logic.

Recall that in the worst case the remainder can be computed from:

$$a - b \times Q'' = R$$

Since the remainder's relationship to zero is the information desired, the equation can be rewritten as:

$$a - b \times Q'' \stackrel{?}{=} 0$$

or

$$a \stackrel{?}{=} b \times Q''$$

Clearly, the back multiplication of $b \times Q''$ is required for this comparison, and this multiplication should be carried in RZ mode, with no effective rounding. However, to reduce the latency penalty, it may be possible to remove the full-width subtraction, replacing it with simpler logic. The remaining question is the number of bits of both a and $b \times Q''$ that are required in this comparison.

Recall from (28) that the maximum error in Q'' is $\pm 2^{-(n+m)}$. Therefore, the maximum error in the product $b \times Q''$ with respect to a can be derived as follows:

$$\begin{aligned} b_{max} &= 1 - 2^{-n} && (29) \\ error_{max} &= b_{max} \times Q'' \\ &= (1 - 2^{-n}) \times (\pm 2^{-(n+m)}) && (30) \end{aligned}$$

or

$$-2^{-(n+m)} + 2^{-(2n+1)} < error < 2^{-(n+m)} - 2^{-(2n+1)} \quad (31)$$

From this analysis, it can be seen that so long as the number of guard bits m is greater than or equal to 1, then the absolute value of the error in the product $b \times Q''$ is strictly less than 0.5 ulp. Accordingly, the sign of the difference between a and $b \times Q''$ can be exactly predicted by only examining the LSB of a and the LSB of $b \times Q''$.

To demonstrate how this prediction can be implemented, consider the entries in table 4. In this table, let $X = a$ and $Y = b \times Q''$, each with n bits. From this table, it is clear that the sign of the difference can be written as:

$$Sign = X_{lsb} \text{ XNOR } Y_{lsb} \quad (32)$$

X_{lsb}	Y_{lsb}	Error in Y	Sign of $X - Y$
0	0	$< .01111\cdots$	-
0	1	$> .01111\cdots$	+
1	0	$> .01111\cdots$	+
1	1	$< .01111\cdots$	-

Table 4: Sign prediction

Thus, the sign of the difference can be computed by using one gate, rather than requiring a complete full-width carry-propagate addition. This hardware is sufficient to handle the RN rounding mode which only requires the sign of the remainder and no information about the magnitude itself. Again, this is because RN only requires remainder information when trying to determine on which side of the mid-point between two machine numbers the true quotient lies. As the exact halfway case can not occur, exact equality to zero of the remainder need not be checked.

For the other three rounding modes, whether or not the remainder is exactly zero must also be determined. This is due to the fact that RP, RM, and RZ use remainder information to detect if the true quotient is less than, greater than, or exactly equal to a machine number. Rather than using additional hardware to detect remainder equality to zero, it is proposed to reuse existing hardware in the FP multiplier. For proper implementation of IEEE rounding for FP multiplication, most FP multipliers utilize dedicated sticky-bit logic. The sticky-bit of the multiplier is a flag signifying whether all bits below the LSB of the product are zero. In the context of the product of the back-multiplication of $b \times Q''$, this sticky-bit signals whether the product is exactly equal to a, and thus whether the remainder is exactly zero.

For those cases in RP, RM, and RZ requiring remainder information, the product $b \times Q''$ is computed using RZ, and the LSB of a and $b \times Q''$ are observed, along with the sticky-bit from the multiplier. After using equation (32) to determine the sign, the following switching expressions can be used:

$$(b \times Q'' == a) = Sign \text{ AND } \overline{Sticky} \tag{33}$$

$$(b \times Q'' > a) = Sign \text{ AND } Sticky \tag{34}$$

$$(b \times Q'' < a) = \overline{Sign} \tag{35}$$

7 Conclusion

This paper has examined the methodology of rounding when implementing division by functional iteration. The basic techniques of rounding assuming a fixed division latency has been clarified. Extensions to techniques for reducing the rounding penalty have been proposed. It has been shown that by using m guard bits in the adjusted quotient estimate, a back-multiplication and subtraction are required for only 2^{-m} of all cases. Further, a technique has been presented which reduces the subtraction in the remainder formation to very simple combinational logic using the LSB's of the dividend and the back product of the quotient estimate and the divisor, along with the sticky bit from the multiplier. The combination of these techniques allows for increased division performance in dynamically-scheduled processors.

References

[1] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The IBM System/360 Model 91: Floating-point execution unit. *IBM Journal of Research and Development*, 11:34–53, January 1967.

[2] H. M. Darley et al. Floating Point / Integer Processor with Divide and Square Root Functions. U.S. Patent No. 4,878,190, 1989.

[3] M. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8), August 1970.

[4] D. Goldberg. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[5] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

[6] P. W. Markstein. Computation of elementary function on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, pages 111–119, January 1990.

[7] S. Oberman and M. Flynn. An analysis of division algorithms and implementations. Technical Report No. CSL-TR-95-675, Computer Systems Laboratory, Stanford University, July 1995.

[8] S. Oberman and M. Flynn. Design issues in division and other floating-point operations. *To Appear in IEEE Transactions on Computers*, 1996.

[9] E. Schwarz. Rounding for quadratically converging algorithms for division and square root. In *Proceedings of the Asilomar Conference*, 1995.