High-Precision Division and Square Root

ALAN H. KARP and PETER MARKSTEIN
Hewlett-Packard Laboratories

We present division and square root algorithms for calculations with more bits than are handled by the floating-point hardware. These algorithms avoid the need to multiply two high-precision numbers, speeding up the last iteration by as much as a factor of 10. We also show how to produce the floating-point number closest to the exact result with relatively few additional operations.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*computer arithmetic*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Division, quad precision, square root

1. INTRODUCTION

Floating-point division and square root take considerably longer to compute than addition and multiplication. The latter two are computed directly while the former are usually computed with an iterative algorithm. The most common approach is to use a division-free Newton-Raphson iteration to get an approximation to the reciprocal of the denominator (division) or the reciprocal square root, and then multiply by the numerator (division) or input argument (square root). Done carefully, this approach can return a floating-point number within 1 or 2 ULPs (units in the last place) of the floating-point number closest to the exact result. With more work, sometimes a lot more, the error can be reduced to half a ULP or less.

The usual approach works quite well when the results are computed to no more precision than the hardware addition and multiplication. Typically, division and square root take 10 to 20 machine cycles on a processor which does a multiplication in the same precision in 2 or 3 cycles. The situation is not so good for higher-precision results because the cost of the

Authors' addresses: A. H. Karp, Future Systems Department, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304; email: alan_karp@hpl.hp.com; P. Markstein, Computer Systems Laboratory, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304; email: peter_markstein@hpl.hp.com.

final multiplication is much higher. The key feature of our algorithm is that it avoids this expensive multiplication.

Section 2 discusses some methods used to do division and square root. Section 3 describes the floating-point arithmetic and the assumptions we are making about the floating-point hardware. Section 4 describes our new algorithms. In Section 5 we present an analysis of the accuracy of the algorithm. Next, in Section 6, we show how to get the correctly rounded results with relatively few additional operations, on average. The procedures we use for testing are described in Section 7. Programs written in the programming language of the Unix desktop calculator utility `bc` [Hewlett-Packard 1991] that implement these algorithms are included in the appendix.

2. ALGORITHMIC CHOICES

There are a number of ways to perform division and square root. In this section we discuss these methods with emphasis on results that exceed the precision of the floating-point hardware.

One approach is to do elementary school long division, a scheme identical to successive subtraction for base 2 arithmetic [Goldberg 1990]. The procedure is straightforward. To compute B/A, we initialize a remainder to $R = 0$. Let S be a shift register containing R concatenated with B. Then, for each bit we do the following steps:

(1) Shift S left one bit so that the high-order bit of B becomes the low-order bit of R.

(2) Subtract A from the R part of S.

(3) If the result is negative, set the low-order bit of the B part of S to zero; otherwise set it to 1.

(4) If the result of the subtraction is negative, add A to the R part of S.

At the end of N steps, the N-bit quotient is in B, and the remainder is in R.

A similar scheme can be used for square root [Monuschi and Mezzalama 1990]. We start with the input value A and a first guess $Y = 0$. For each bit we compute the residual, $R = A - Y^2$. Inspection of R at the kth step is used to select a value for the kth bit of Y. The selection rules give some flexibility in tuning the algorithm.

There are improvements to these approaches. For example, nonrestoring division avoids adding A to R. The SRT algorithm [Patterson and Hennessy 1990] is an optimization based on a carry-sum intermediate representation. Higher-radix methods compute several bits per step. Unfortunately, these improvements cannot overcome the drawback that we only get a fixed number of bits per step, a particularly serious shortcoming for high-precision computations.

Polynomial approximations can also be used [Monuschi and Mezzalama 1990]. Chebyshev polynomials are most often used because they form an approximation with a guaranteed bound on the maximum error. However, each term in the series adds about the same number of bits to the precision of the result, so this approach is impractical for high-precision results.

Another approach is one of the CORDIC methods which were originally derived for trigonometric functions, but can be applied to square root [Monuschi and Mezzalama 1990]. These methods treat the input as a vector in a particular coordinate system. The result is then computed by adding and shifting some tabulated numbers. However, the tables would have to be inconveniently large for high-precision results.

There are two methods in common use that converge quadratically instead of linearly as do these other methods: Goldschmidt's method and Newton iteration. Quadratic convergence is particularly important for high-precision calculations because the number of steps is proportional to the logarithm of the number of digits.

Goldschmidt's algorithm [Patterson and Hennessy 1990] is based on a dual iteration. To compute B/A we first find an approximation to $1/A = A'$. We then initialize $x = BA'$ and $y = AA'$. Next, we iterate until convergence.

(1) $r = 2 - y$

(2) $y = ry$

(3) $x = rx$

If $0 \leq AA' < 2$, the iteration converges quadratically. A similar algorithm can be derived for square root. Round-off errors will not be damped out because Goldschmidt's algorithm is not self-correcting. In order to get accurate results, throughout the calculation we have to maintain more digits than appear in the final result. While carrying an extra word of precision is not a problem in a multiprecision calculation, the extra digits required will slow down a calculation that could have been done with quad-precision hardware.

Newton's method for computing B/A is to approximate $1/A$, apply several iterations of the Newton-Raphson method, and multiply the result by B. The iteration for the reciprocal of A is

$$x_{n+1} = x_n + x_n(1 - Ax_n). \tag{1}$$

Newton's method for computing $\sqrt{A}$ is to approximate $1/\sqrt{A}$, apply several iterations of the Newton-Raphson method, and multiply the result by A. The iteration for the reciprocal square root of A is

$$x_{n+1} = x_n + \frac{x_n}{2}(1 - Ax_n^2). \tag{2}$$

Newton's method is quadratically convergent and self-correcting. Thus, round-off errors made in early iterations damp out. In particular, if we know that the first guess is accurate to N bits, the result of iteration k will be accurate to almost 2^kN bits.

3. FLOATING-POINT ARITHMETIC

Our modification to the standard Newton method for division and square root makes some assumptions about the floating-point arithmetic. In this section we introduce some terminology, describe the hardware requirements to implement our algorithm, and show how to use software to overcome any deficiencies.

We assume that we have a floating-point arithmetic that allows us to write any representable number

$$x = \beta^e \sum_{k=0}^{N} \beta^{-k} f_k, \tag{3}$$

where β is the number base; the integer e is the exponent; and the integers f_k are in the interval $[0,\beta)$.

As shown in Section 4, we compute a $2N$-bit approximation to our function from an N-bit approximation. We refer to these as full-precision and half-precision numbers, respectively. In the special case where the operations on half-precision numbers are done in hardware, we refer to $2N$-bit numbers as quad precision and N-bit numbers as hardware precision. The term "high precision" is used to denote both quad- and multiprecision operations.

Note that N is not necessarily the number of mantissa bits in the memory or register format. In IBM hexadecimal floating point and on the RS/6000, a quad number is stored as two doubles. The former has 28 hexadecimal digits; the latter has 106 bits. In contrast, IEEE-type quad-precision numbers might have 113 bits of mantissa; a double-precision number has 53 bits; and a double extended number has 64 bits of precision.

We will not be precise in counting bits until we get to the proofs. For the remainder of this section, we assume that N is at least half the number of bits required for the result. Hence, for quad numbers represented in double-double format, N is the number of mantissa bits in a double. For quad numbers in an IEEE-style 113-bit format, we assume $N \geq 57$.

Our algorithm will not be particularly useful if it requires special properties of the floating-point arithmetic, especially if the required properties are not widely available. We now describe what our algorithm needs from the hardware and how we can code around any deficiencies.

There are several ways to implement floating-point arithmetic. We can round the result of every operation to the floating-point number nearest the exact result. We can truncate the result by throwing away any digits

```
void prod(a,b,c)   /* All digits of c = a*b */
  double a, b, c[]
{
  long double t;
  t = (long double) a * (long double) b;
  c[0] = t;
  c[1] = t - c[0];
}
```

Fig. 1. One way to get the high and low parts of the product of two double-precision numbers on a machine that supports quad-precision variables. The result is returned as two double-precision numbers.

beyond the number of digits in the input format. When we truncate, we can use a guard digit to improve the average accuracy.

It is possible to do a multiply-add as a multiplication followed by an addition, which has two roundings, or as a fused operation with only one rounding. The fused multiply-add operation can be done with full precision or with only some of the digits in the intermediate result. Our algorithms are insensitive to what choices are made. Surprisingly enough, we are able to get accurate results even if we truncate all intermediate operations. Furthermore, we show a rounding algorithm that is most efficient if the intermediate results are truncated.

Our algorithms are based on the premise that we can compute the full-precision product or sum of two half-precision numbers, all $2N$ bits. Implementing these sums and products is tedious but straightforward for multiprecision arithmetic. If we want a quad-precision result, we can exploit special hardware available on some machines. Also, we avoid computing digits we do not need by using the high- and low-order parts of full-precision numbers. How we extract these pieces from a quad-precision number depends on the hardware features of the machine.

Some machines, such as the IBM S/370, provide quad arithmetic. On these machines, we get the bits we want by promoting the double-precision inputs to quad, doing the arithmetic operation, and selecting the bits we need. Note that we do not use the fact that the inputs are double-precision numbers. Promoting them to quads sets the low-order bits to zero, a fact the hardware ignores. Hence, this approach is slow because multiplying quads takes four times as long as multiplying doubles. In addition, we need extra arithmetic operations to select the bits we want, as shown in Figure 1.

Other machines have a fused multiply-add instruction [Olsson et al. 1990, pp. 34–42]. These instructions compute the $2N$ bits of the product, add the addend, and round or truncate the result. We get the high-order part of a product from the normal arithmetic operations; we get the low-order part using the trick shown in Figure 2, which takes only twice as long as hardware precision multiplication. Note that if the first product is rounded to produce `c[0]`, the two elements of `c` may be of different sign.

```
void prod(a,b,c)   /* All digits of c = a*b */
  double a, b, c[]
{
  c[0] = a*b;
  c[1] = a*b - c[0];
}
```

Fig. 2. One way to get the high and low parts of the product of two double-precision numbers on a machine that does a fused multiply-add but does not support a quad datatype. The result is returned as two double-precision numbers.

```
void prod(a,b,c)   /* All digits of c = a*b */
  float a[], b[], c[]
{
  double t, u, v, w, x, y;
  u = (double) a[0]*(double) b[0];
  v = (double) a[0]*(double) b[1];
  w = (double) a[1]*(double) b[0];
  x = (double) a[1]*(double) b[1];
  y = v + w + (float) x;
  t = u + y;
  c[0] = t;
  t -= c[0];
  c[1] = t;
  t = t - c[1] + x - (float) x;
  c[2] = t;
  c[3] = t - c[2];
}
```

Fig. 3. One way to get all the digits of the product of two double-precision numbers stored in single-single format. The result is returned as four single-precision numbers.

At least one machine, the IBM S/370, has a special instruction that returns all the bits in the product of two doubles. This operation is more efficient than promoting to quad, since the hardware uses the fact that the inputs are doubles. We can still compute the low-order bits by doing a quad subtraction similar to what we show in Figure 2.

On machines that do not support any of these operations, we can still simulate the operations we need, but we must use a single-single representation as our hardware format. In this representation, a double-precision number is represented as the sum of two singles. No bits are lost on machines that have an exponent range independent of the numerical precision, such as the S/370. Unfortunately, in IEEE arithmetic [ANSI 1985] two singles hold fewer mantissa bits than a double, so we only get a 96-bit quad result.

Figure 3 shows one way to multiply two single-single-precision numbers and return both the high- and low-order parts of the product. We have assumed that the hardware supports double-precision arithmetic and that the double-precision format holds at least two digits more than in the

```
void sum(a,b,c)   /* Leading 2N digits of c = a+b */
  float a[], b[], c[]
{
  double cl, ch, max, min, t;
  max = (fabs(a[0])>fabs(b[0])?a[0]:b[0]);
  min = (fabs(a[0])<=fabs(b[0])?a[0]:b[0]);
  ch = a[0] + b[0];
  t = min - (ch - max);
  cl = a[1] + b[1] + t;
  t = ch + cl;
  cl = cl - (t - ch);
  c[0] = t;
  c[1] = t - c[0];
  c[2] = cl;
  c[3] = cl - c[2];
}
```

Fig. 4. One way to get the leading $2N$ digits of the sum of two numbers stored in the single-single format. We store the high- and low-order parts of the inputs in separate single-precision words, add the high- and low-order parts, and propagate any carries from the low-order part to the high-order part of the result. The result is returned as four single-precision numbers.

product of two single-precision numbers, a condition met by IEEE floating point [ANSI 1985].

Addition is different. The sum of two floating-point numbers can have a very large number of digits in the result if the exponents differ by a lot. Figure 4 shows one way to add two numbers in the single-single format and return the leading quad-precision part of the result in four single-precision numbers. Each such addition takes 12 floating-point operations.

There are significant differences in the implementations of multiprecision and quad-precision addition and multiplication. Full-precision multiplication takes up to four times longer than half-precision multiplication; quad-precision multiplication takes at least four times longer than hardware precision multiplication with hardware support and as much as 100 times longer without. On the other hand, carry propagation is much simpler in quad precision, since we are dealing with only two numbers.

Our algorithms use many of the combinations of precisions shown in Table I. In this table, the subscript indicates the number of digits in the result, d denoting the hardware (double) precision. We can compute V_d in hardware. U_{2d} is available in hardware on some systems. If it is not, we will have to use software. The cost estimates are for double-precision operations that return quad results done in hardware (Q_h), these operations done in software (Q_s), multiprecision numbers of up to a few hundred digits (denoted C, since we use the conventional approach), and multiprecision numbers of more than a few hundred digits (denoted F, since we use an FFT method).

Table I. Different Multiplications

Input	Input	Output	Operation	Relative Cost Q_h	Q_s	C	F
Full	Full	All bits	$P_{4d} = A*B$	11	132	8	4
Full	Full	Full	$Q_{2d} = A*B$	8	84	4	4
Full	Half	All bits	$R_{3d} = A*b$	7	96	3	2
Full	Half	Full	$S_{2d} = A*b$	4	48	2	2
Full	Half	Half	$T_d = A*b$	3	36	2	1
Half	Half	Full	$U_{2d} = a*b$	1	12	2	1
Half	Half	Half	$V_d = a*b$	1	1	1	1

Entries are listed in order of decreasing cost. A and B are full-precision numbers of $2N$ digits, while a and b are half-precision numbers of N digits. The subscript kd denotes a number consisting of k half-precision words.

Henceforth we assume that our hardware returns all the bits we need. If it does not, we have to implement functions such as the ones shown for all unsupported operations.

3.1 Quad

Quad precision is the number format with approximately twice as many digits as the largest format handled in hardware. Some systems have a true quad-precision data type; others use a double-double representation. Some do quad arithmetic in hardware; others use software. In almost all cases, quad operations are built up out of operations on hardware precision numbers stored in registers.

We assume that our quad numbers are either in double-double format or in an IEEE style with 113 mantissa bits. Our numbers are written as $A = A_h + \eta A_l$, where η is the precision of our half-precision numbers, with $\eta = 2^{-53}$ for IEEE double precision on most machines, but with $\eta = 2^{-64}$ for machines that have a double-extended format like the Intel x87. We assume the latter for IEEE-style quad inputs.

In order to round our results correctly, we need two additional computations not shown in Table I. These operations, $W_d = C - AB$ and $X_{2d} = C - AB$, are each a multiply-add of three full-precision numbers, where $AB \approx C$. The approximation is such that W_d is less than two ULPs of C_h and such that X_{2d} is less than two ULPs of C. We exploit the special structure of the result by proceeding from high-order terms to low-order terms, something we cannot do without these conditions. Our implementation, shown later in Figure 12 (Appendix C), makes use of the fact that there is a lot of cancellation of terms. For example, `g[8]` can have at most two nonzero digits. In addition, we do not always have to finish the calculation, as we show in Section 6.

3.2 Multiprecision

Our discussion is based on the public domain package `mpfun` [Bailey 1992]. A multiprecision number is stored in a single-precision floating-point array.

The first word is an integer-valued floating-point number whose absolute value represents the number of words in the mantissa; the sign of this word is the sign of the multiprecision number. The next word contains an integer-valued floating-point number representing the exponent of the number base, β. In contrast to the notation of Eq. (3), the decimal point follows the first mantissa word, not the first digit.

Multiplication is based on the fact that virtually every floating-point system in existence will return all the digits of the product of two single-precision numbers. Converting this double-precision result into two single-precision numbers gets us ready for the next operation. A similar trick can be used for addition; all intermediate sums are computed in double precision, and carries are propagated to single-precision numbers.

Since we are interested in the efficiency of our algorithms, we must consider the cost of the basic operations. Adding two $2N$-bit numbers takes about twice as long as adding two N-bit numbers. The procedure is straightforward. First, align the numbers by scaling one of them until their exponents are the same. Next, add corresponding elements. Finally, we propagate the carries. The value returned is the leading $2N$ bits of the result.

Multiplication is more complicated. The time to compute the product of an N-digit number and an M-digit number scales as NM if we use the direct method. Methods based on fast Fourier transforms (FFT) for the product of two N-digit numbers take a time proportional to $N \log N \log \log N$ [Bailey 1992]. Of course, the coefficients of these scalings are much larger for multiplication than for addition. Hence, algorithms that avoid multiplying full-precision numbers together will run faster than those that need such products.

4. NEW ALGORITHMS

We have seen how complicated high-precision arithmetic can be. It is clear that our algorithm should avoid such operations whenever possible. The algorithms presented in this section perform high-precision division and square root with no full multiplications of the precision of the desired result.

First look at the standard Newton method for division and square root shown in Eqs. (1) and (2). If we are doing a high-precision calculation, we can implement these iterations without doing any operations on two numbers in the longest precision. First of all, in the early iterations, when the approximation is less accurate than the base number format, we use hardware addition and multiplication. At each iteration beyond this accuracy, we double the number of digits carried to match the accuracy of the approximation [Bailey 1992].

The last iteration needs some care to avoid multiplying two long precision numbers. Look at the square root. First we compute $u = Ax_n$ as a full times a half, keeping a full-precision result; then we compute $v = ux_n$ the

same way. Since x_n is a very good approximation to $\sqrt{A}$, $1 - v$ will have at most 1 more bit than x_n. Hence, we can compute $x_n(1 - v)$ as a half times a half. We do not even need to compute more than half the digits in the product, since this quantity is a small correction to the current estimate. Hence, both the last multiplication and the last addition can be done in half precision.

The problem comes in computing the desired result from the last iteration. For division, we must multiply the full x_{n+1} times the full B; for square root, we multiply by the full A. These operations are expensive, anywhere from 2 to 100 times the time of half-precision operations.

There is a simple way to avoid these long multiplications; do them before the last iteration rather than after. Now the last iteration for square root becomes

$$y_{n+1} = y_n + \frac{x_n}{2}(A - y_n^2), \tag{4}$$

with $y_n = Ax_n$, while for division,

$$y_{n+1} = y_n + x_n(B - Ay_n), \tag{5}$$

where $y_n = Bx_n$.

The key to this approach is that, in both cases, these are the Newton iterations for the final result—quotient and square root—respectively. Hence, y_{n+1} is the desired approximate result accurate to nearly the number of digits in a full-precision number. Since the Newton iteration is self-correcting, we do not even need an accurate value for y_n. In our implementations we compute y_n as the half-precision result of multiplying two half-precision numbers.

There is a subtle point in the way the terms have been collected in the final iterations. In both cases, we have brought the multiplicand inside the parentheses. In this way, we compute the residual based on y_n, the number we are correcting. If we had made the other choice, we would be correcting y_n with a residual computed from x_n. This choice would have forced us to compute y_n as the product of a full- and a half-precision number in order to get the right answer.

We can now see the savings. For square root, we compute y_n^2 as the full-precision product of two half-precision numbers. After subtracting the result from A, we are left with at most one bit more than a half-precision number, so we can use a half-times-half to half-precision result when we multiply by x_n. We do a little bit more work for division. Here we must multiply the full-precision number A times the half y_n, but the rest of the operations are the same as for square root.

The savings are summarized in Tables II and III, much of which was supplied by David Bailey. We look at division and square root for three

Table II. Comparison of Cost of New versus Standard Division Algorithms for Three Precisions (Quad (Q_hS_s), up to a Few Hundred Digits (C), and More than a Few Hundred (F))

Standard Approach $X_{n+1} = x_n + x_n(1 - Ax_n)$ $B / A \approx BX_{n+1}$					New Approach $y_n = Bx_n$ $Y_{n+1} = y_n + x_n(B - Ay_n)$				
Operation	Q_h	Q_s	C	F	Operation	Q_h	Q_s	C	F
$T = Ax_n$	4	48	2	2	$y_n = Bx_n$	1	1	1	1
$t = 1 - T$	1	2	0	0	$T = Ay_n$	4	48	2	2
$t = x_nt$	1	1	1	1	$t = B - T$	1	2	0	0
$X_{n+1} = x_n + t$	1	3	0	0	$t = x_nt$	1	1	1	1
$T = BX_{n+1}$	8	96	4	2	$Y_{n+1} = y_n + t$	1	3	0	0
Total	15	150	7	5	Total	8	55	4	4

Table III. Comparison of Cost of New versus Standard Square Root Algorithms for Three Precisions (Quad (Q_hS_s), up to a Few Hundred Digits (C), and More than a Few Hundred (F))

Standard Approach $X_{n+1} = x_n(1 - Ax_n^2) / 2$ $\sqrt{A} \approx AX_{n+1}$					New Approach $y_n = Ax_n$ $Y_{n+1} = y_n + x_n(A - y_n^2) / 2$				
Operation	Q_h	Q_s	C	F	Operation	Q_h	Q_s	C	F
$T = x_n^2$	–	12	–	1	$y_n = Ax_n$	1	1	1	1
$T = AT$	–	96	–	2	$T = y_n^2$	1	12	2	1
$T = Ax_n$	4	–	2	–	$t = A - T$	1	2	0	0
$T = Tx_n$	4	–	2	–	$t = x_nt / 2$	2	2	2	2
$t = 1 - T$	1	2	0	0	$Y_{n+1} = y_n + t$	1	3	0	0
$t = x_nt / 2$	2	2	2	2					
$X_{n+1} = x_n + t$	1	3	0	0					
$T = AX_{n+1}$	8	96	4	2					
Total	20	211	10	7	Total	6	20	5	4

precisions—quad (Q), up to a few hundred digits (C, since we use the conventional multiplication algorithm), and more than a few hundred digits (F, since we use FFT-based multiplication). Times if hardware returns the quad result of the product or sum of two doubles are in the Q_h column; times without hardware assist are in the Q_s column. In each column, the unit used is the time to compute the half-precision product of two half-precision numbers. For quad precision, the unit is the hardware multiplication time. Hardware precision addition is assumed to take as long as hardware multiplication, and the time to do addition is ignored for multiple-precision calculations. Uppercase letters denote full-precision numbers; and lowercase letters denote half-precision numbers. If assignment is made to a half-precision number, we need to calculate only the leading digits of the result. We only count the operations in the last iteration because that is the only place we modify the standard algorithm.

5. ANALYSIS

Several of our results depend on properties of quotients, square roots, and Newton Raphson approximation methods. In this section we sketch proofs of these properties.

THEOREM 5.1. *In k-digit floating-point arithmetic, using a prime radix, a quotient cannot be exact in $k + 1$ digits in which the low-order result digit is significant.*

PROOF. Suppose that the quotient $c = a/b$ is exactly $c = p^e \sum_{i=0}^{k} c_i p^{-i}$. If $b = p^f \sum_{i=0}^{k-1} b_i p^{-i}$, it must be the case that

$$\begin{aligned} a &= b \times c \\ &= p^{e+f} \sum_{i=0}^{k} c_i p^{-1} \sum_{i=0}^{k-1} b_i p^{-i} \\ &= p^{e+f}(c_0 b_0 + \cdots + c_k b_j p^{-(k+j)}), \end{aligned}$$

where j is the lowest-order nonzero digit in b. Since c_k and b_j are both nonzero, $c_k b_j$ is not divisible by the radix, so that this quantity requires at least $k + j + 1$ digits. But a was representable as a k-digit number. □

The same line of reasoning shows that a square root cannot be representable as a $(k + 1)$-digit result.[1]

THEOREM 5.2. *To round a quotient correctly to k bits, the quotient must be computed correctly to at least $2k + 1$ bits.*

PROOF. A proof of this proposition has been published previously [Kahan 1987]. With certain precautions, $2k$ bits suffice [Markstein 1990]. □

For binary radix, the following empirical evidence suggests this proposition. Consider the quotient $1/(1 - 2^{-k})$. The closest binary fraction to the result, up to $2k$ bits, is $1 + 2^{-k}$ (a $(k + 1)$-bit number). To get the correctly rounded result (either 1 or $1 + 2^{-k+1}$), we must know whether $1 + 2^{-k}$ is an overestimate or an underestimate, a task which takes almost as much computation as to establish the quotient to $2k + 1$ bits. Only when the quotient is computed to $2k + 1$ bits (or more) is it clear that the closest binary fraction to the quotient is $1 + 2^{-k} + 2^{-2k}$ (up to $3k$ bits), which clearly should be rounded to $1 + 2^{-k+1}$.

[1]Exercise for the reader: why does this proof fail for nonprime radices? It does not hold for hex floating-point arithmetic, for example.

THEOREM 5.3. *To round a square root correctly to k bits, the square root must be computed correctly to at least* $2k + 3$ *bits.*

PROOF. As before, with certain precautions, $2k$ bits suffice [Markstein 1990]. □

Again, empirical evidence suggested this proposition. Consider the square root of $1 - 2^{-k}$. The closest binary fraction to the result, using $k + 1$ to $2k + 2$ bits, is $1 - 2^{-k-1}$. Just as in the case of division, the decision whether to round up or down depends on whether this approximation is an overestimate or an underestimate, which requires almost as much computation as to establish the quotient to $2k + 3$ bits. Computing further shows that the best approximation is $1 - 2^{-k-1} - 2^{-2k-3}$ (up to $3k + 4$ bits), so that the best k-bit approximation is $1 - 2^{-k}$.

Theorems 5.2 and 5.3 imply that the trick of looking at a few extra bits in the result to decide on the rounding direction [Patterson and Hennessy 1990] will only work *most* of the time, not *all* of the time as we would like.

In Section 3 we claim that using truncating arithmetic also will produce acceptable results. For square root, if the reciprocal square root is known to n bits using n-bit arithmetic, how much error can be introduced in the application of Eq. (4) if truncated arithmetic is used? Our next result addresses that question directly.

THEOREM 5.4. *If* x_n *approximates* $1/\sqrt{A}$ *with relative error* δ, $y_n = Ax_n$ *is computed with relative error* η *(due to rounding or truncation); the residual* $r = A - y_n^2$ *is computed with relative error* η_1^2*; and* x_n *times the residual r is computed with error* η_2*. An application of Eq. (4) computes an approximation to* $\sqrt{A}$ *with a relative error of approximately*

$$-(3\delta^2 + 4\delta\eta + \eta^2 + 2\eta\eta_2 + 2\eta_2\delta) / 2,$$

where δ, η, η_1, *and* η_2 *are all of the order of the half-precision truncation error* ϵ.

PROOF. Note that we compute y_n^2 as the full-precision result of the product of two half-precision numbers, so there is no error in this computation. Also, y_n is a very good approximation to the square root, so the error in the residual is of order ϵ^2.

We can write $x_n = (1 + \delta)/\sqrt{A}$, $y_n = Ax_n(1 + \eta)$, $r = (A - y_n^2)(1 + \eta_1^2)$, and $z = x_n r(1 + \eta_2)$ as a means of expressing the relative errors given in the statement of the theorem. We now have

$$y_{n+1} = y_n + \frac{x_n}{2}(A - y_n^2)(1 + \eta_1^2)(1 + \eta_2)$$

$$= Ax_n(1 + \eta) + \frac{x_n}{2}\{A - [Ax_n(1 + \eta)]^2\}(1 + \eta_1^2)(1 + \eta_2)$$

$$= \frac{A}{\sqrt{A}}(1 + \delta)(1 + \eta) + \frac{1 + \delta}{2\sqrt{A}}[A - A(1 + \delta)^2(1 + \eta)^2](1 + \eta_1^2)(1 + \eta_2)$$

$$= \sqrt{A}(1 + \delta)\left\{1 + \eta + \frac{1}{2}[1 - (1 + \delta)^2(1 + \eta)^2](1 + \eta_1^2)(1 + \eta_2)\right\}$$

$$= \sqrt{A}\left\{1 - \frac{1}{2}[3\delta^2 + 4\delta\eta + \eta^2 + 2\eta\eta_2 + 2\eta_2\delta + \mathbb{O}(\epsilon^3)]\right\}. \quad \square$$

If we set $\eta = \eta_2 = 0$ in Theorem 5.4, we see that y_{n+1} would approach $\sqrt{A}$ from below, and the coefficient of the leading error term, $-3\delta^2/2$, is the standard error term for the reciprocal square root [Hildebrand 1965]. If all operations used truncating arithmetic, making η, and η_2 nonpositive, all the terms of $\mathbb{O}(\epsilon^2)$ would be nonpositive if x_n underestimates $1/\sqrt{A}$, and Eq. (4) would still approach $\sqrt{A}$ from below. When round-to-nearest is used, some of the terms could be negative and could cause Eq. (4) to overestimate the square root.

Interestingly, we are guaranteed to have an underestimate no matter what sign we have for the initial guess or how accurate that guess is. The convexity and curvature of the square root function guarantee that in exact arithmetic an application of Eq. (2) underestimates the true result. Using truncating arithmetic guarantees that using finite precision can never make the result larger. Hence, we know that x_n in Eq. (4) is always an underestimate.

Unless we are careful in the handling of the early iterations, x_n can be an overestimate. We can guarantee that $\delta \leq 0$ by using the full-precision A in the iterations through n, but this approach is computationally expensive. Another approach is to round A to half precision away from zero. We are now evaluating Eq. (2) with a value for A that is slightly too large. Backward error analysis guarantees that x_n has the same accuracy as if we truncated A; the error we make is less than a half-precision ULP in both cases.

If both x_n and y_n are known to n bits of precision, that is to say, $|\delta| < 2^{-n}$ and $|\eta| < 2^{-n}$, and the truncation error is such that $|\eta_2| < 2^{-n}$, our proof of Theorem 5.4 shows that the relative error due to Eq. (4) may be as large as

$$-6 \cdot 2^{-2n} + \mathbb{O}(2^{-3n}). \tag{6}$$

Using Eq. (4) will leave a result with an error less than 6 units in the $2n$th bit. In other words, Eq. (4) is expected to yield an approximation good to at least $2n - 3$ bits.

We can improve this result by computing x_n times the residual as the full-precision result of the product of two half-precision numbers, which makes the error $\eta_2 = \mathbb{O}(\epsilon^2)$. This change makes the error

$$-4 \cdot 2^{-2n} + \mathbb{O}(2^{-3n}),$$

giving us an underestimate with at most 2 bits in error. In Section 6 we describe a method for rounding this result that, on average, is almost as efficient when the error is 3 bits as when it is 2 bits. Hence, we assume we use the algorithm that leaves us with the slightly higher residual of Eq. (6).

If we had used round-to-nearest arithmetic, then we would have $|\delta| < 2^{-n-1}$ and $|\eta| < 2^{-n-1}$. The error in using Eq. (5) would then be bounded by $2^{-2n} + \mathbb{O}(2^{-3n})$, giving almost $2n$ good bits, but we are no longer guaranteed convergence from below. We show in Section 6 that we can make use of the fact that the result is an underestimate.

When we are computing results in round-to-nearest mode, we can save some computation by allowing x_n to be an overestimate of $1/\sqrt{A}$ by as much as a half-precision ULP. The procedure shown in Appendix B does this by using the truncated value of A in the early iterations. In this case, y_{n+1} may overestimate the square root by as much as 1/6 of a full-precision ULP, as shown in Appendix C.

We now know that y_{n+1} is either an underestimate or the correctly rounded result. We save computation when the result is an overestimate, because the first stage of the Tuckerman test tells us we are done. If y_{n+1} is guaranteed to be an underestimate, we get the correctly rounded result only for exact quotients. Unfortunately, we cannot use this approach for other roundings because we can end up with a double rounding.

THEOREM 5.5. *If x_n approximates $1/A$ with relative error δ, and $y_n = Bx_n$ is computed with relative error η (due to rounding or truncation), then the product Ay_n is computed with relative error η_3^2; the residual $r = B - Ay_n$ is computed with relative error η_1^2; and the product $x_n r$ is computed with relative error η_2. An application of Eq. (5) computes an approximation to B/A with a relative error of approximately*

$$-\delta^2 - \delta\eta - \eta\eta_2 - \eta_2\delta,$$

where δ, η, η_k, $k = 1, 2, 3$, are all of the order of the half-precision truncation error ϵ.

PROOF. We compute Ay_n as the full-precision result of the product of a full-precision and a half-precision number making the error $\mathbb{O}(\epsilon^2)$. Also, y_n

is a very good approximation to the ratio, so the error in the residual is of order ϵ^2.

We can write $x_n = (1 + \delta)/A$, $y_n = Bx_n(1 + \eta)$, $w = Ay_n(1 + \eta_3^2)$, $r = (B - Ay_n)(1 + \eta_1^2)$, and $z = x_n r(1 + \eta_2)$ as a means of expressing the relative errors given in the statement of the theorem. We now have

$$
\begin{aligned}
y_{n+1} &= y_n + x_n(B - Ay_n(1 + \eta_3^2))(1 + \eta_1^2)(1 + \eta_2) \\
&= Bx_n(1 + \eta) + x_n\{B - [ABx_n(1 + \eta)(1 + \eta_3^2)]\}(1 + \eta_1^2)(1 + \eta_2) \\
&= \frac{B}{A}(1 + \delta)(1 + \eta) \\
&\quad + \frac{1 + \delta}{A}[B - B(1 + \delta)(1 + \eta_3^2)(1 + \eta)](1 + \eta_1^2)(1 + \eta_2) \\
&= \frac{B}{A}(1 + \delta)\{1 + \eta + [1 - (1 + \delta)(1 + \eta_3^2)(1 + \eta)](1 + \eta_1^2)(1 + \eta_2)\} \\
&= \frac{B}{A}[1 - \delta^2 - \delta\eta - \delta\eta_2 - \eta\eta_2 + \mathcal{O}(\epsilon^3)]. \quad \square
\end{aligned}
$$

Theorem 5.5 shows that, for division, Eq. (5) always approaches the result from below when the results are truncated, as long as x_n underestimates $1/A$. (As with square root, we guarantee an underestimate by computing x_n with a value of A rounded to a half-precision ULP away from zero.) If the arithmetic operations round, the solution may be approached from above. As is the case for square root, an application of the Newton-Raphson iteration in Eq. (1) guarantees an underestimate in exact arithmetic, and using truncating arithmetic preserves this property.

If both x_n and y_n are known to n bits of precision, that is to say, $|\delta| < 2^{-n}$ and $|\eta| < 2^{-n}$, our proof of Theorem 5.5 shows that the relative error due to Eq. (5) may be as large as $-4 \cdot 2^{-2n} + \mathcal{O}(\epsilon^3)$. So, using Eq. (5) will leave a result with an error less than 4 units in the $2n$th bit. In other words, Eq. (5) is expected to yield an approximation good to at least $2n - 2$ bits.

As with square root, we can improve this result by computing x_n times the residual as the full-precision result of the product of two half-precision numbers, which makes the error $-2 \cdot 2^{-2n} + \mathcal{O}(2^{-3n})$, leaving only 1 bit in error. As with square root, we will be more computationally efficient with the larger error when we want correctly rounded results. However, if a 1 ULP error suffices, our algorithm requires no additional steps if we use the higher-precision product.

If we had used round-to-nearest arithmetic, then we would have had $|\delta| < 2^{-n-1}$ and $|\eta| < 2^{-n-1}$. The error in using Eq. (5) would then be bounded by $2^{-2n} + \mathcal{O}(2^{-3n})$, giving almost $2n$ good bits, but we are no longer guaranteed convergence from below.

Under special circumstances, the results from Eqs. (4) and (5) will then produce correctly round-to-nearest $2n$-bit results [Markstein 1990] even though Theorems 5.2 and 5.3 require more precision than these equations normally deliver.

When we are computing results in round-to-nearest mode, we will obtain correctly rounded results more often by allowing x_n to be an overestimate of $1/A$ by as much as a half-precision ULP. The procedure shown in Appendix B does this by using the truncated value of A in the early iterations. In this case, y_{n+1} may overestimate the quotient by as much as 1/4 of a full-precision ULP as shown in Appendix C. Unfortunately, we cannot use this approach for other roundings because we can end up with a double rounding.

6. ROUNDING

We have shown how to get the quotient and square root accurate to a few ULPs. This accuracy usually suffices for isolated evaluations, but there are times when more accuracy is needed. For example, if the result is not correct to half a ULP or less, the computed value for the function may not satisfy such algebraic properties as monotonicity. In addition, if the result returned is not the floating-point number closest to the exact result, future implementations may not return the same value causing confusion among users.

The need to maintain monotonicity and the identical results for different implementations is important for both quad- and high-precision arithmetic. In the latter case, simply asking the user to use one more word precision than the application needs is not a major inconvenience. However, if the user takes all the words in the multiprecision result and does not round to the needed number of bits, the desired arithmetic properties may be lost. In addition, in the worst case, the rounding cannot be done correctly unless twice as many bits as needed are known. In the case of quad precision we have no choice; we cannot return more digits than the user asks for so we must do the rounding ourselves.

Quad-precision rounding depends on how numbers are stored in registers. Most machines have registers with the same number of digits as the storage format of the numbers; others, such as the Intel x87 floating-point coprocessors, implement the IEEE standard [ANSI 1985] recommended extended format which keeps extra bits for numbers in the registers. We consider both these implementations. First, we look at getting the correct value on a machine that keeps more digits in the registers than in the memory format. We also show how to get the correct result on a machine that keeps no additional digits in the register.

For concreteness, we look at the problem of computing the quad-precision quotient and square root. For simplicity, we ignore error conditions and operations on the characteristics of the floating-point numbers, since these are the same for both the usual algorithm and ours. We also assume that the machine has an instruction that returns the quad-precision result of arithmetic operations on two double-precision numbers. If we do not have this capability, we have to build up the algorithm using the double-precision product of two single-precision numbers as shown in Figures 3 and 4.

As shown in Section 5, our algorithm produces a $2N$-bit mantissa with all but the last few bits correct. We also showed that there are some numbers we cannot round correctly without computing at least $4N + 3$ bits of the result. The implementations are different if our registers are wider than our words in memory or not, so we describe them separately.

6.1 Long Registers

Here we show how to compute the floating-point number closest to the exact square root or quotient when the output from our final iteration has more bits than we need return to the user. In a multiprecision calculation we can always carry an additional word of precision but return a result with the same number of digits as the input parameters. For quad results we assume we are running on a machine that does base 2 arithmetic and keeps more bits in the registers than in memory. More specifically, we assume that a double-precision number has 53 mantissa bits in memory and 64 in the registers. Our input is assumed to have 113 mantissa bits. Our goal is to return the correctly rounded 113-bit result.

As shown in Section 5 our algorithm produces a mantissa with at least 125 correct bits. We also showed that there are square roots that we cannot round correctly unless we know the result to at least 229 bits. Rather than do an additional, expensive iteration, we use the fact that we have underestimated the correct result. Hence, we know the correctly rounded value is the 128-bit number we have computed truncated to 113 bits or that 113-bit number plus one ULP.

One way to find out which is correct is to compute the residual with the computed value y_{n+1} and with $y_{n+1} + \mu_{113}$, where μ_{113} is one ULP of a 113-bit number. The smaller residual belongs to the correct result. This approach is expensive, both because we must compute two residuals and because each residual needs the product of two quad-precision numbers.

Tuckerman rounding [Agarwal et al. 1986; Markstein 1990] avoids this problem. We need only compute the residual for $y_{n+1} + \mu_{113}/2$. If the sign is positive, the larger number is the desired result; if negative, we want the smaller. As shown in Section 5 the value cannot be exactly zero for a machine with a prime number base so we do not have to worry about breaking ties.

The full Tuckerman test for precisions with more digits than the hardware can handle is expensive. For example, for square root, even if we make the simplification that

$$(y_{n+1} + \mu_{113}/2)^2 \geq y_{n+1}(y_{n+1} + \mu_{113}), \tag{7}$$

we must compute a lot of terms. These operations are almost exactly what we must do first to do another iteration, but we need only look at the sign of the result. Finishing the iteration would require only a half-precision multiplication and an addition.

Computing the residual is expensive, but if we have extra bits in the registers, we can avoid doing the test most of the time. We know that the first 125 bits of our result are correct and that we have underestimated the correct answer. Hence, if bit 114, the rounding bit, is a 1, we know we must round up. If the rounding bit is a 0, and one of the bits 115 through 125 is a 0, we know that we should return the smaller value. Only if bit 114 is a 0, and bits 115 through 125 are all ones, do we need further computation. In other words, there is exactly one pattern of 12 bits that we cannot round properly. If the trailing bits are random, we need do extra work for only 1 in 2,048 numbers.

Even in the rare cases where we cannot decide which way to round from the 128-bit result, we can often avoid doing the full Tuckerman test. Any time the intermediate result becomes negative or zero we can stop because we know we should return the smaller value, a case which occurs half the time for random bit patterns. We can also stop if the intermediate result is positive and larger in magnitude than a bound on the magnitude of the remaining terms which are all negative.

There are two convenient places to check: after accumulating all terms larger than η^2 and again after computing all terms larger than η^3. If we are using normalized, IEEE floating-point, double-precision numbers which have an implicit leading 1, mantissas lie in the interval [1,2). This means that the coefficient of η^2 and η^3 are less than 10. (See Figure 12.) Therefore, we can stop after computing the first set of terms unless the residual is positive and less than 10η or after the second set unless the residual is positive and less than $10\eta^2$.

For randomly distributed residuals we need the test only one time out of 2,048 inputs. We can stop after nine operations, three of them multiplications, all but once in 16,384 times. The next test will determine the rounding in all but one case in 2^{64} trials. Hence, the average performance for correctly rounded results is almost exactly the same as that for results accurate to one ULP, although in the worst case we must do a large number of additional operations.

The situation is even better for multiprecision calculations. First of all, there are only four half-precision multiplications in the test; all the rest of the operations are additions. Second, it is a simple matter to do all the intermediate calculations with a single extra word. In this case, the only bad situation is when the lowest-order word of the result, when normalized

to be an extension of the next higher-order word, has a leading zero followed by all ones in the bits known to be correct. Since this situation arises only once in 2^{61} evaluations for random trailing bits, we almost never need the Tuckerman test. When we do need the test, the early tests catch an even larger percentage of the cases than for quad precision. However, there is no escaping the fact that there are some input values that require us to compute the full Tuckerman test to decide which way to round.

As originally formulated, Tuckerman rounding can be used only for square root, not division. That formulation uses the approximation in Eq. (7). We have no such identity for division, but we do have extra bits in our registers so our test is to check the sign of $(B - Ay_{n+1}) + A\mu_{113}/2$. This version is what appears in Appendix B.

6.2 Short Registers

We can also compute the floating-point number closest to the exact result on a machine that does not keep extra bits in the registers. In this case we assume that our quad-precision numbers are stored as two double-precision numbers each having 53 mantissa bits in both memory and the registers. Our input is 106 bits long, and we wish to compute the correctly rounded 106-bit result.

We know from Section 5 that the algorithm described in Section 4 has produced a result with at least 103 correct bits. We would have to do two more iterations to get the correct result, a very expensive proposition, but we cannot apply standard Tuckerman rounding, since there may be bits in error. Fortunately, we can apply Tuckerman rounding six times at different bit positions at a modest average number of operations.

The procedure is simple. We take the output of the Newton iteration and set the three low-order bits to zero. Since we have an underestimate of the correct result, we know that the correctly rounded 103-bit result is either the number we have or that number plus μ_{103}, the ULP of a 103-bit number. If the Tuckerman test tells us to use the larger number, we know that bit 103 must be a one, so we add μ_{103} to set this bit. We now have an underestimate of the correct result, but with 104 correct bits. Now we repeat at the 104th and 105th bits. One more application of the test at the 106th bit does the trick, but now we add μ_{106} if the test indicates that our result is too small.

We need to be careful at the start. Say that the correctly rounded result ends in the hexadecimal string `8000001`. If we make a two-ULP underestimate, our working value would be `7FFFFFF`. If we use the procedure just described, we would return `8000000`. Although we have made only a two-ULP underestimate, the last correct bit is 28 positions from the end. We account for this situation by testing our initial estimate, `7FFFFF0` in our example, plus one in the position of the lowest-order correct bit, hexadecimal 10 in the example. If the Tuckerman test indicates that we

have an underestimate, we continue with the larger value, e.g., 8000000. Otherwise, we continue with the smaller, e.g., 7FFFFF0.

Notice that we are repeating a lot of the calculations in the subsequent Tuckerman tests. The formulation of the test in Figure 12 was chosen to minimize the number of operations that must be repeated between applications. Only terms that depend on μ_k, the point at which the test is being applied, must be recomputed. Hence, in the best case where the first test succeeds, the first application takes 9 operations, and each additional takes 1 more for a total of 14 operations. In the worst case, we need 36 operations for the first test and 8 for each remaining test, a total of 76 operations. In the most common case, the second test is definitive so we need 18 operations for the first application and 5 for each additional one for a total of 43. Fortunately, none of the repeated operations is a multiplication.

The alternative to get the error down to a half ULP or less is to do two more Newton iterations, since we need to compute 215 bits to get a 106 bit result. Since we only have 102 bits correct, the first extra iteration only gives us 204-bit accuracy. Repeated application of the Tuckerman test is clearly faster.

Division is a bit trickier. If we had extra bits in the register, we could form $y_{n+1} + \mu_{106}/2$. We do not, however, so we compute the residual from $(B - Ay_{n+1}) - A\mu_{106}/2$. Since y_{n+1} is a very good approximation to B/A, the first two terms will nearly cancel leaving a positive value, since we have underestimated the exact quotient. We have no problem computing $A\mu_{106}/2$, barring underflow, since the result is just a rescaling of A.

6.3 Other Roundings

All the discussion has assumed we want to return the floating-point result closest to the exact answer, round-to-nearest. The IEEE floating-point standard [ANSI 1985] includes three other rounding modes. We can also return the result in any of these modes as well.

If we have extra bits in the registers, we handle the different roundings in the following way.

—*Round to zero:* Return the output from the Newton iteration.

—*Round to positive infinity:* If the result is positive, add one ULP to the output from the Newton iteration. If the result is negative, return the output of the Newton iteration.

—*Round to negative infinity:* If the result is negative, subtract one ULP from the output of the Newton iteration. If the result is positive, return the output of the Newton iteration.

If we do not have extra bits in the registers, the following method returns values with these other roundings. Our computed result is the value obtained after the first five Tuckerman roundings.

—*Round to zero:* Return the computed value.

—*Round to negative infinity:* If the result is positive, add one ULP to the computed result. If the result is negative, return the computed value.

—*Round to positive infinity:* If the result is negative, subtract one ULP from the computed result. If the result is positive, return the computed value.

These procedures do not handle exact results correctly, e.g., 6.0/3.0. The result of our Newton iteration is guaranteed to be an underestimate, so we know the correct bits beyond our desired precision are all ones when the result is exact.

Our procedure is similar to that for round-to-nearest mode. Whenever the trailing bits in a long register are all 1, we add one ULP and apply the Tuckerman test, a result of 0 indicating an exact result. As before, we can stop when the current value and the bound on the remaining terms guarantees that the result cannot be 0. When we have short registers, we need to apply the test every time the last bit is a 1. The upshot is that those quotients or square roots that are easiest to compute by hand take the most time in any rounding other than round to nearest.

7. TEST PROCEDURES

To test our algorithms, we generated division and square root problems which presented difficult rounding problems. For division an algorithm is known for generating divisors and dividends so that the correct quotient is almost exactly 1/2 ULP more than a representable floating-point number [Kahan 1987]. This is accomplished by solving the diophantine equation

$$2^{k-j}A = BQ + r \bmod 2^k \tag{8}$$

for a given odd divisor B, where r is chosen to be an integer near $B/2$, and where k is the precision of the floating-point arithmetic. B is chosen to satisfy

$$2^{k-1} \le B < 2^k,$$

and solutions of Eq. (8) are sought for A and Q satisfying the same inequality (with $j = 0$ or 1).

Some difficult square root problems are given by numbers of the form

$$1 + 2^{-k+1}(2j + 1)$$

and

$$1 - 2^{-k}(2j + 1)$$

whose square roots are slightly less than

$$1 + 2^{-k+1}(j + 1/2)$$

and

$$1 - 2^{-k}(j + 1/2),$$

respectively. These almost-exact square roots require $k + 1$ bits, making the rounding decision difficult (in all cases the result must be rounded downward). To generate cases requiring close rounding decisions, we attempt to find an integer x satisfying

$$2^{k-1} \le x < 2^k$$

for which $(x + 1/2)^2$ is close to a multiple of 2^k. We seek solutions of the diophantine equation

$$(x + 1/2)^2 = 2^{k+j}y + \epsilon \bmod 2^k$$

or, multiplying by 4 to remove fractions,

$$(2x + 1)^2 = 2^{k+j+2}y + 1 + 8m \bmod 2^k$$

for various small integer values of m, in which j can be 0 or 1 (4ϵ must be one more than a multiple of eight, since all odd squares are congruent to 1 mod 8). We require y to satisfy the same inequality as x. For any y which satisfies the above diophantine equation,

$$\sqrt{2^{k+j}y} \cong x + 1/2 + o(-m/x),$$

so that the correctly rounded result is $x + 1$ when $m < 0$, and x when $m \ge 0$.

Using these techniques, a large number of test cases were generated both for division and square root, and our implementations of the algorithms presented in this article successfully rounded each example correctly.

8. CONCLUSIONS

Division and square root account for a small percentage of all floating-point operations, but the time it takes to execute them dominates some calculations. The extra time is even more noticeable for quad- and multiple-precision arithmetic. In this article we have shown how to speed up these calculations by an appreciable amount.

The primary improvement made to the standard Newton algorithm is to multiply by the appropriate factor, the numerator for division and the input argument for square root, before the last iteration instead of after. This trick works because the modified equation is almost exactly the Newton iteration for the desired function instead of a reciprocal approximation. The key observation is that the reciprocal approximation from the penultimate iteration is sufficiently accurate to be used in the last iteration.

```
void prod(a,b,c)
  double a[], b, c
{
  long double t;
  t = a[0]*b + a[1]*b;
  c = (double t);
}
```

Fig. 5. Full times half to half.

```
void prod(a,b,c)
  double a[], b, c[]
{
  long double d, t;
  d = a[0]*b;
  t = d + a[1]*b;
  c[0] = (double t);
  c[1] = t - c[0];
}
```

Fig. 6. Full times half to full.

```
void prod(a,b,c)
  double a[], b, c[]
{
  long double s, t, u;
  s = a[0]*b;
  t = a[1]*b;
  u = s + t;
  c[0] = (double u);
  c[1] = u - c[0];
  c[2] = (s-c[0])-c[1]) +
}
```

Fig. 7. All bits of full times half.

```
void prod(a,b,c)
  double a[], b[], c[]
{
  long double t, u;
  t = a[0]*b[0]+a[0]*b[1]+a[1]*b[0]+a[1]*b[1];
  c[0] = (double t);
  c[1] = t - c[0];
}
```

Fig. 8. Full times full to full.

The performance improvement comes from avoiding any multiplications of full-precision numbers. An interesting result is that it becomes practical to do quad division and square root in hardware because we can use the existing floating-point hardware. Implementing hardware to multiply two full-precision numbers is impractical.

```
void prod(a,b,c)
  double a[], b[], c[]
{
  long double r, s, t, u, v, w, x, y, z;
  r = a[0]*b[0];
  s = a[0]*b[1];
  t = a[1]*b[0];
  u = a[1]*b[1];
  v = s + (double u);
  w = t + (v - (double v));
  x = (double v) + (w - (double w));
  y = (double x) + (double w);
  z = r + y;
  c[0] = (double z);
  c[1] = z - (double z);
  c[2] = x - (double x);
  c[3] = u - (double u);
}
```

Fig. 9. All bits of full times full.

We have also shown how to compute the correctly rounded result with a minimum of additional arithmetic. The method presented is a modification of the Tuckerman test which works for both division and square root. We showed how to extend Tuckerman rounding to the case where the registers do not hold any extra bits.

APPENDIX

A. BASIC OPERATIONS

Figures 5–9 contain the code used for the operation counts of the various multiplications in Table I. These figures assume that the hardware will provide all the bits of the product and the leading quad-precision part of the sum of two double-precision numbers. If the hardware does not have this capability, the multiplication and addition operations must be replaced with functions such as those shown in Figures 3 and 4, respectively. In this case, each addition counts as four operations and each multiplication as 12.

B. A BC IMPLEMENTATION

These algorithms (shown in Figures 10–13) have been tested using the Unix desktop calculator `bc` with programs written in its C-like language. This utility does integer operations with an arbitrary number of digits and floating point to any desired accuracy. We chose to implement the algorithms with integers because it afforded complete control over the bits included in the operations.

The programs that follow use a few utility routines. The routine `h(a,n)` returns the first `n` base `obase` digits in `a`, where `obase` is the number base used for output; `l(a,n)`, the second `n` base `obase` digits of `a`. The function

```
define s(A,n){
  auto b,c,d,E,x,y,z
  b = h(A,n)
  x = table_look_up(b)              /* (n/4)-bits     */
  y = h(x*x,n)/n^2
  z = (n^6-h(b*y,n)/2)/n^4
  x = (x*n^2+h(x*z,n))/n^2          /* (n/2)-bits     */
  y = h(x*x,n)/n^2
  z = ((n^6-h(b*y,n))/2)/n^4
  x = h(x*n^2+h(x*z,n),n)/n^2       /* n-bits         */
  /* Last iteration */
  y = h(b*x,n)/n^4                  /* sqrt(h(A))     */
  c = x/2                           /* 1/(2y)         */
  d = h(A-y*y,n)                    /* A - y^2        */
  E = (y*n^4+h(c*d,n))/n^4          /* Almost 2n bits */
  return ( E )
}
```

Fig. 10. Program in bc to compute the quotient of two full-precision numbers A, to nearly 2n bits.

```
define d(B,A,n){
  auto c,d,e,F,x,y,z
  c = h(B,n)
  d = h(A,n)
  x = table_look_up(d)              /* (n/4)-bits     */
  z = (n^4-h(d*x,n))/n^2
  x = (x*n^2+h(x*z,n))/n^2          /* (n/2)-bits     */
  z = (n^4-h(d*x,n))/n^2
  x = x*n^2+h(x*z,n)                /* n-bits         */
  /* Last iteration */
  y = h(c*x,n)/n^4                  /* h(B)/h(A)      */
  e = h(B*n^2-A*y,n)/n^2            /* B - A*h(y)     */
  F = (y*n^4+h(e*x,n))/n^4          /* Almost 2n bits */
  return ( F )
}
```

Fig. 11. Program in bc to compute the quotient of two full-precision numbers, B/A, to nearly 2n bits.

table_look_up is a place holder for the usual procedure for getting the first guess.

C. BOUND ON OVERESTIMATES

The handling of the iterations done in half precision determines whether or not the final result can overestimate the true value even if all intermediate values are truncated. In this section we prove that this overestimate is bounded by 1/4 of a full-precision ULP for division and by 1/6 of a full-precision ULP for square root.

The basic cause of the overestimate is the decision to use a truncated value of A in the early iterations. Since this value underestimates A, the

```
define t(b,a,y,u){
g[1] = h(a,n)*h(y,n);          g[2] = h(a,n)*l(y,n)
g[3] = l(a,n)*h(y,n);          g[4] = h(b,n)-h(g[1],n)
g[5] = h(g[4],n)-l(g[1],n);    g[6] = h(g[5],n)-h(g[2],n)
g[7] = h(g[6],n)-h(g[3],n);    g[8] = h(g[7],n)+l(b,n)
g[9] = h(g[8],n)-h(a,n)*u
if ( g[9] <= 0 ) return ( 0 )
if ( g[9] > 8*n^2 ) return ( 1 )
g[10] = l(a,n)*l(y,n);         g[11] = l(g[2],n)+l(g[3],n)
g[12] = h(g[10],n)+h(g[11],n); g[13] = l(g[5],n)-h(g[12],n)
g[14] = l(g[6],n)+l(g[7],n);   g[15] = h(g[13],n)+h(g[14],n)
g[16] = l(g[9],n)+h(g[15],n);  g[17] = h(g[9],n)+h(g[16],n)
g[18] = h(g[17],n)-l(a,n)*u
if ( g[18] <= 0 ) return ( 0 )
if ( g[18] > 10*n ) return ( 1 )
g[19] = l(g[13],n)+l(g[14],n); g[20] = l(g[10],n)+l(g[11],n)
g[21] = h(g[20],n)+l(g[12],n); g[22] = h(g[19],n)-h(g[21],n)
g[23] = l(g[15],n)+h(g[22],n); g[24] = l(g[18],n)+h(g[23],n)
g[25] = l(g[17],n)+h(g[24],n); g[26] = l(g[16],n)+h(g[25],n)
g[27] = h(g[18],n)+h(g[26],n)
if ( g[27] <= 0 ) return ( 0 )
if ( g[27] > C ) return ( 1 )
g[28] = l(g[20],n)+l(g[21],n); g[29] = l(g[19],n)-h(g[28],n)
g[30] = l(g[22],n)+l(g[23],n); g[31] = h(g[29],n)+h(g[30],n)
g[32] = l(g[24],n)+l(g[25],n); g[33] = l(g[26],n)+h(g[31],n)
g[34] = h(g[32],n)+h(g[33],n); g[35] = l(g[27],n)+h(g[34],n)
g[36] = h(g[27],n)+h(g[35],n)
if ( g[36] <= 0 ) return ( 0 )
return ( 1 )
}
```

Fig. 12. Tuckerman test. The value of u determines the bit position for the test μ_k.

```
R = u(A)                    /* 1 ULP square root     */
F = R + t(A,R,R,u)          /* 1/2 ULP square root   */
S = d(B,A)                  /* 1 ULP quotient        */
G = S + t(B*nV^2,A,S,u)     /* 1/2 ULP quotient      */
```

Fig. 13. Getting the error to half a ULP or less on a machine with extra bits in the registers. The functions u(A) and d(B,A) return the square root and quotient, respectively, with at most three bits in error. Routine t performs the Tuckerman test at a bit position given by the last argument, 1 in these cases. F and G are the square root and quotient with an error no larger than 1/2 ULP.

value for x_n can overestimate $1/A$ or $1/\sqrt{A}$ by as much as a half-precision ULP, denoted ϵ. The following theorems quantify the effect on the computed result.

```
R = u(A)           /* Square root                */
u = 10
V = u*(R/u)        /* Set trailing bits to 0     */
if ( t(A,R,R,2*u) > 0 ) v = v + u  /* Borrow?  */
while ( u > 1 ) {
  V = V + (u/2)*t(A,V,V,u)
  u = u/2
}
F = V + t(A,V,V,1) /* Final rounding             */
```

Fig. 14. Getting the error to half a ULP or less on a machine without extra bits in the registers. We assume an error as large as 15 ULPs.

THEOREM C.1. *If we compute x_n in Eq. (1) using the value of A truncated to half precision, the error in y_{n+1} is*

$$-\epsilon^2 < y_{n+1} < \epsilon^2 / 4.$$

PROOF. The extreme values of the error are the maximum and minimum of the error term given in Theorem 5.5 in the cube bounded by the limits on the three components of the error. More precisely,

$$F(\delta, \eta, \eta_2) = -\delta^2 - \eta\eta_2 - \delta\eta_2,$$

where $-\epsilon < \eta, \eta_2 \leq 0$ and $0 \leq \delta < \epsilon$. (If $\delta < 0$, we are guaranteed to have underestimated the exact result.)

First we show that the extremal values lie on the boundaries of the cube. The partial derivatives

$$\frac{\partial F}{\partial \delta} = -2\delta - \eta - \eta_2, \quad \frac{\partial F}{\partial \eta} = -\delta - \eta_2, \quad \frac{\partial F}{\partial \eta_2} = -\delta - \eta,$$

and the second derivatives

$$\frac{\partial^2 F}{\partial \delta^2} = -2, \quad \frac{\partial^2 F}{\partial \eta_2^2} = \frac{\partial^2 F}{\partial \eta^2} = 0, \quad \frac{\partial^2 F}{\partial \delta \partial \eta} = \frac{\partial^2 F}{\partial \delta \partial \eta_2} = \frac{\partial^2 F}{\partial \eta \partial \eta_2} = -1$$

show that the first derivatives are zero only at saddle points in the interior of the cube and on its faces. Hence, the extreme values must lie on the edges. We now find the extreme values along each of the 12 edges.

On the face with $\delta = 0$, $F(0, \eta, \eta_2) = -\eta\eta_2$. Since this function is linear in each term, the extreme values occur at the end points so that $-\epsilon^2 < F(0, \eta, \eta_2) < 0$.

On edges 5–8, $\delta = \epsilon$. On edge 5, $\eta = 0$. We have $-\epsilon^2 < F(\epsilon, 0, \eta_2) = -\epsilon^2 - \epsilon\eta_2 < 0$. On edge 6, $\eta_2 = 0$, so that $F(\epsilon, \eta, 0) = -\epsilon^2$. We get the same value on edge 7 where $\eta = -\epsilon$.

On edge 8, $\eta_2 = -\epsilon$, which makes $-\epsilon^2 < F(\epsilon, \eta, -\epsilon) = \eta\epsilon < 0$.

On edges 9 and 10, we have $\eta = \eta_2 = 0$ and $\eta = -\epsilon$, $\eta_2 = 0$, respectively. In both cases, $-\epsilon^2 < F = -\delta^2 < 0$.

On edge 11, $\eta = \eta_2 = -\epsilon$ so that $F(\delta, -\epsilon, -\epsilon) = -\delta^2 - \epsilon^2 + \epsilon\delta$. This function has a minimum of $-\epsilon^2$ when $\delta = 0$ and when $\delta = \epsilon$. It has a maximum of $-3\epsilon^2/4$ when $\delta = \epsilon/2$.

Finally, on edge 12 where $\eta = 0$ and $\eta_2 = -\epsilon$, we find that $F(\delta, 0, -\epsilon) = -\delta^2 + \epsilon\delta$ which has a minimum of 0 when $\delta = 0$ and when $\delta = \epsilon$. It has a maximum of $\epsilon^2/4$ when $\delta = \epsilon/2$. □

We get a similar result for square root.

THEOREM C.2. *If we compute x_n in Eq. (2) using the value of A truncated to half precision, the error in y_{n+1} is*

$$-\epsilon^2 < y_{n+1} < \epsilon^2/6.$$

PROOF. The proof follows that for division. □

ACKNOWLEDGMENTS

We would like to thank David Bailey, Dennis Brzezinski, and Clemens Roothaan for their help, and Eric Schwarz, who did an outstanding job refereeing this article. His careful review led to several significant improvements.

REFERENCES

AGARWAL, R. C., COOLEY, J. W., GUSTAVSON, F. G., SHEARER, J. B., SLISHMAN, G., AND TUCKERMAN, B. 1986. New scalar and vector elementary functions for the IBM System/370. *IBM J. Res. Dev. 30*, 2 (Mar.), 126–144.

ANSI. 1985. ANSI/IEEE standard for binary floating point arithmetic. Tech. Rep. ANSI/IEEE Standard 754-1985. IEEE Press, Piscataway, NJ.

BAILEY, D. H. 1992. A portable high performance multiprecision package. RNR Tech. Rep. RNR-90-022. NASA Ames Research Center, Moffett Field, CA.

GOLDBERG, D. 1990. Appendix A. In *Computer Architecture: A Qualitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

HEWLETT-PACKARD. 1991. *HP-UX Reference*. 1st ed. Hewlett-Packard, Fort Collins, CO.

HILDEBRAND, F. B. 1965. *Introduction to Numerical Analysis*. 2nd ed. Prentice Hall Press, Upper Saddle River, NJ.

KAHAN, W. 1987. Checking whether floating-point division is correctly rounded. Monograph. Computer Science Dept., University of California at Berkeley, Berkeley, CA.

MARKSTEIN, P. W. 1990. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM J. Res. Dev. 34*, 1 (Jan.), 111–119.

MONUSCHI, P. AND MEZZALAMA, M. 1990. Survey of square rooting algorithms. *IEE Proc. 137*, 1, Part E (Jan.), 31–40.

OLSSON, B., MONTOYE, R., MARKSTEIN, P., AND NGYUENPHU, M. 1990. *RISC System/6000 Floating-Point Unit*. IBM Corp., Riverton, NJ.

PATTERSON, D. A. AND HENNESSY, J. L. 1990. *Computer Architecture: A Qualitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Received: December 1994; revised: March 1997 and May 1997; accepted: June 1997