Quad-Double Arithmetic: Algorithms, Implementation, and Application[*]

Yozo Hida[†] Xiaoye S. Li[‡] David H. Bailey[‡]

October 30, 2000

Abstract

A quad-double number is an unevaluated sum of four IEEE double precision numbers, capable of representing at least 212 bits of significand. Algorithms for various arithmetic operations (including the four basic operations and various algebraic and transcendental operations) are presented. A C++ implementation of these algorithms is also described, as well as an application of this quad-double library.

[*]This research was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

[†]Computer Science Division, University of California, Berkeley, CA 94720 (`yozo@cs.berkeley.edu`).

[‡]NERSC, Lawrence Berkeley National Laboratory, 1 Cycloton Rd, Berkeley, CA 94720 (`xiaoye@nersc.gov`, `dhbailey@lbl.gov`).

Contents

1 Introduction

Multiprecision computation has a variety of application areas, such as pure mathematics, study of mathematical constants, cryptography, and computational geometry. Because of this, many arbitrary precision algorithms and libraries have been developed using only the fixed precision arithmetic. They can be divided into two groups based on the way precision numbers are represented. Some libraries store numbers in a *multiple-digit* format, with a sequence of digits coupled with a single exponent, such as the symbolic computation package Mathematica, Bailey's MPFUN [2], Brent's MP [4] and GNU MP [8]. An alternative approach is to store numbers in a *multiple-component* format, where a number is expressed as unevaluated sums of ordinary floating-point words, each with its own significand and exponent. Examples of this format include [7, 10, 11]. The multiple-digit approach can represent a much larger range of numbers, whereas the multiple-component approach has the advantage in speed.

We note that many applications would get full benefit from using merely a small multiple of (such as twice or quadruple) the working precision, without the need for arbitrary precision. The algorithms for this kind of "fixed" precision can be made significantly faster than those for arbitrary precision. Bailey [1] and Briggs [5] have developed algorithms and software for "double-double" precision, twice the double precision. They used the multiple-component format, where a double-double number is represented as an unevaluated sum of a leading double and a trailing double.

In this paper, we present some algorithms and an implementation for "quad-double" numbers, in other words, numbers with four times the double precision. We use the multiple-component format to take advantage of speed. A quad-double number is an unevaluated sum of four IEEE doubles. The quad-double number (a_0, a_1, a_2, a_3) represents the exact sum $a = a_0 + a_1 + a_2 + a_3$, where a_0 is the most sigficant component. We have designed and implemented algorithms for basic arithmetic operations, as well as some algebraic and transcendental functions. We have performed extensive correctness tests and compared the results with arbitrary precision package MPFUN. Our quad-precision library is available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`.

The rest of the paper is organized as follows. Section 2 describes some basic properties of IEEE floating point arithmetic and building blocks for our quad-double algorithms. In Section 3 we present the quad-double algorithms for basic operations, including renormization, addition, multiplication and division. Section 4 and 5 present the algorithms for some algebraic operations and transcendental functions. Section 6 describes some auxiliary functions. Section 7 describes a C++ library implementing the above algorithms, including the instructions on how to build and test the software on various computer platforms. Section 8 presents the timing results of the kernel operations on different architectures. Our quad-double library is successfully integrated into a parallel vortex roll-up simulation code; this is briefly described in Section 9. Section 10 discusses future work.

2 Preliminaries

In this section, we present some basic properties and algorithms of IEEE floating point arithmetic used in quad-double arithmetic. These results are based on Dekker [7], Knuth [9], Priest [10], Shewchuk [11], and others. In fact, many of the algorithms and diagrams are directly taken from, or based on, Shewchuk's paper.

All basic arithmetics are assumed to be performed in IEEE double format, with round-to-even rounding on ties. For any binary operator $\cdot \in \{+, -, \times, /\}$, we use $\mathrm{fl}(a \cdot b) = a \odot b$ to denote the floating point result of $a \cdot b$, and define $\mathrm{err}(a \cdot b)$ as $a \cdot b = \mathrm{fl}(a \cdot b) + \mathrm{err}(a \cdot b)$. Throughout this paper, $\varepsilon = 2^{-53}$ is the machine epsilon for IEEE double precision numbers, and $\varepsilon_{\mathrm{qd}} = 2^{-211}$ is the precision one expects for quad-double numbers.

Lemma 1. [11, p. 310] *Let a and b be two p-bit floating point numbers such that $|a| \geq |b|$. Then* $|\mathrm{err}(a + b)| \leq |b| \leq |a|$.

Lemma 2. [11, p. 311] *Let a and b be two p-bit floating point numbers. Then* $\mathrm{err}(a + b) = (a + b) - \mathrm{fl}(a + b)$ *is representable as a p-bit floating point number.*

Algorithm 3. [11, p. 312] The following algorithm computes $s = \mathrm{fl}(a + b)$ and $e = \mathrm{err}(a + b)$, assuming $|a| \geq |b|$.

```
QUICK-TWO-SUM($a, b$)
1.  $s \leftarrow a \oplus b$
2.  $e \leftarrow b \ominus (s \ominus a)$
3.  return $(s, e)$
```

Algorithm 4. [11, p. 314] The following algorithm computes $s = \mathrm{fl}(a+b)$ and $e = \mathrm{err}(a+b)$. This algorithm uses three more floating point operations instead of a branch.

```
TWO-SUM($a, b$)
1.  $s \leftarrow a \oplus b$
2.  $v \leftarrow s \ominus a$
3.  $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
4.  return $(s, e)$
```

Algorithm 5. [11, p. 325] The following algorithm splits a 53-bit IEEE double precision floating point number into a_{hi} and a_{lo}, each with 26 bits of significand, such that $a = a_{\mathrm{hi}} + a_{\mathrm{lo}}$. a_{hi} will contain the first 26 bits, while a_{lo} will contain the lower 26 bits.

```
SPLIT($a$)
1.  $t \leftarrow (2^{27} + 1) \otimes a$
2.  $a_{\mathrm{hi}} \leftarrow t \ominus (t \ominus a)$
3.  $a_{\mathrm{lo}} \leftarrow a \ominus a_{\mathrm{hi}}$
4.  return $(a_{\mathrm{hi}}, a_{\mathrm{lo}})$
```

Algorithm 6. [11, p. 326] The following algorithm computes $p = \mathrm{fl}(a \times b)$ and $e = \mathrm{err}(a \times b)$.

```
TWO-PROD($a, b$)
1.  $p \leftarrow a \otimes b$
2.  $(a_{\mathrm{hi}}, a_{\mathrm{lo}}) \leftarrow$ SPLIT($a$)
3.  $(b_{\mathrm{hi}}, b_{\mathrm{lo}}) \leftarrow$ SPLIT($b$)
4.  $e \leftarrow ((a_{\mathrm{hi}} \otimes b_{\mathrm{hi}} \ominus p) \oplus a_{\mathrm{hi}} \otimes b_{\mathrm{lo}} \oplus a_{\mathrm{lo}} \otimes b_{\mathrm{hi}}) \oplus a_{\mathrm{lo}} \otimes b_{\mathrm{lo}}$
5.  return $(p, e)$
```

Some machines have a fused multiply-add instruction (FMA) that can evaluate expression such as $a \times b \pm c$ with a single rounding error. We can take advantage of this instruction to compute exact product of two floating point numbers much faster. These machines include IBM Power series (including the PowerPC), on which this simplification is tested.

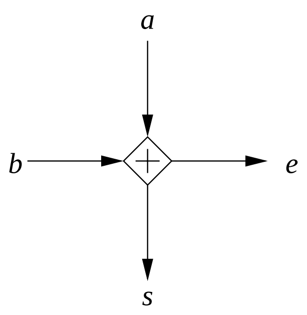

Figure 1: Quick-Two-Sum

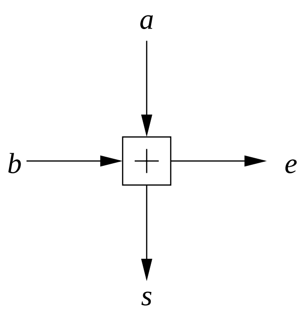

Figure 2: Two-Sum

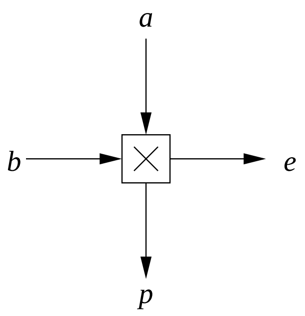

Figure 3: Two-Prod

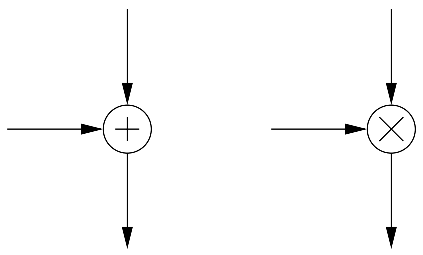

Figure 4: Normal IEEE double precision sum and product

Algorithm 7. The following algorithm computes $p = \mathrm{fl}(a \times b)$ and $e = \mathrm{err}(a \times b)$ on a machine with a FMA instruction. Note that some compilers emit FMA instructions for $a \times b + c$ but not for $a \times b - c$; in this case, some sign adjustments must be made.

```
Two-Prod-FMA($a, b$)
1.  $p \leftarrow a \otimes b$
2.  $e \leftarrow \mathrm{fl}(a \times b - p)$
3.  return $(p, e)$
```

The algorithms presented are the basic building blocks of quad-double arithmetic, and are represented in Figures 1, 2, and 3. Symbols for normal double precision sum and product are in Figure 4.

3 Basic Operations

3.1 Renormalization

A quad-double number is an unevaluated sum of four IEEE double numbers. The quad-double number (a_0, a_1, a_2, a_3) represents the exact sum $a = a_0 + a_1 + a_2 + a_3$. Note that for any given representable number x, there can be many representations as an unevaluated sum of four doubles.

Hence we require that the quadruple (a_0, a_1, a_2, a_3) to satisfy

$$|a_{i+1}| \leq \frac{1}{2}\mathrm{ulp}(a_i)$$

for $i = 0, 1, 2$, with equality occurring only if $a_i = 0$ or the last bit of a_i is 0 (that is, round-to-even is used in case of ties). Note that the first double a_0 is a double-precision approximation to the quad-double number a, accurate to almost half an ulp.

Lemma 8. *For any quad-double number $a = (a_0, a_1, a_2, a_3)$, the normalized representation is unique.*

Most of the algorithms described here produce an expansion that is not of canonical form – often having overlapping bits. Therefore, a five-term expansion is produced, and then renormalized to four components.

Algorithm 9. This renormalization procedure is a variant of Priest's renormalization method [10, p. 116]. The input is a five-term expansion with limited overlapping bits, with a_0 being the most significant component.

```
RENORMALIZE($a_0, a_1, a_2, a_3, a_4$)
 1.  $(s, t_4) \leftarrow$ QUICK-TWO-SUM($a_3, a_4$)
 2.  $(s, t_3) \leftarrow$ QUICK-TWO-SUM($a_2, s$)
 3.  $(s, t_2) \leftarrow$ QUICK-TWO-SUM($a_1, s$)
 4.  $(t_0, t_1) \leftarrow$ QUICK-TWO-SUM($a_0, s$)

 5.  $s \leftarrow t_0$
 6.  $k \leftarrow 0$
 7.  for[1] $i \leftarrow 1, 2, 3, 4$
 8.    $(s, e) \leftarrow$ QUICK-TWO-SUM($s, t_i$)
 9.    if $e \neq 0$
10.      $b_k \leftarrow s$
11.      $s \leftarrow e$
12.      $k \leftarrow k + 1$
13.    end if
14.  end for
15.  return $(b_0, b_1, b_2, b_3)$
```

Necessary conditions for this renormalization algorithm to work correctly are, unfortunately, not known. Priest proves that if the input expansion does not overlap by more than 51 bits, then the algorithm works correctly. However, this condition is by no means necessary; that the renormalization algorithm (Algorithm 9) works on all the expansions produced by the algorithms below remains to be shown.

[1]In the implementation, this loop is unrolled to several `if` statements.

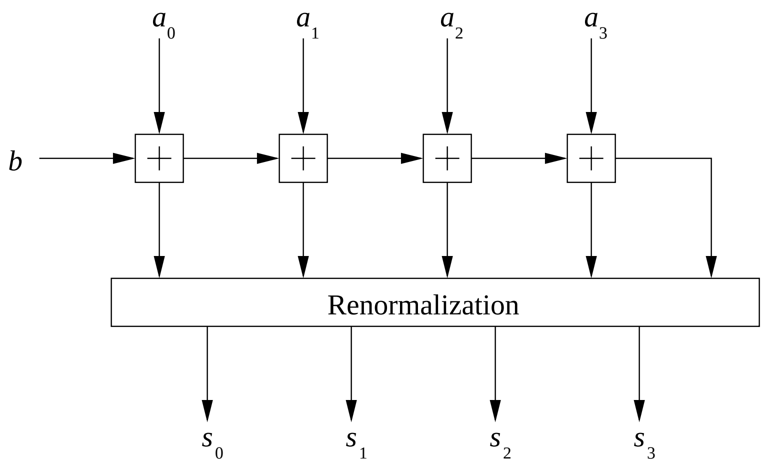

Figure 5: Quad-Double + Double

3.2 Addition

3.2.1 Quad-Double + Double

The addition of a double precision number to a quad-double number is similar to Shewchuk's GROW-EXPANSION [11, p. 316], but the double precision number b is added to a quad-double number a from most significant component first (rather than from least significant). This produces a five-term expansion which is the exact result, which is then renormalized. See Figure 5.

Since the exact result is computed, then normalized to four components, this addition is accurate to at least the first 212 bits of the result.

3.2.2 Quad-Double + Quad-Double

We have implemented two algorithms for addition. The first one is faster, but only satisfies the weaker (Cray-style) error bound $a \oplus b = (1 + \delta_1)a + (1 + \delta_2)b$ where the magnitude of δ_1 and δ_2 is bounded by $\varepsilon_{\text{qd}} = 2^{-211}$.

Figure 6 best describes the first addition algorithm of two quad-double numbers. In the diagram, there are three large boxes with three inputs to them. These are various THREE-SUM boxes, and their internals are shown in Figure 7.

Now for a few more lemmas.

Lemma 10. *Let a and b be two double precision floating point numbers. Let $M = \max(|a|, |b|)$. Then $|\text{fl}(a + b)| \le 2M$, and consequently, $|\text{err}(a + b)| \le \frac{1}{2}\text{ulp}(2M) \le 2\varepsilon M$.*

Lemma 11. *Let x, y, and z be inputs to* THREE-SUM. *Let u, v, w, r_0, r_1, and r_2 be as indicated in Figure 7. Let $M = \max(|x|, |y|, |z|)$. Then $|r_0| \le 4M$, $|r_1| \le 8\varepsilon M$, and $|r_2| \le 8\varepsilon^2 M$.*

Proof. This follows from applying Lemma 10 to each of the three TWO-SUM boxes. First TWO-SUM gives $|u| \le 2M$ and $|v| \le 2\varepsilon M$. Next TWO-SUM (adding u and z) gives $|r_0| \le 4M$ and $|w| \le 4\varepsilon M$. Finally, the last TWO-SUM gives the desired result. ☐

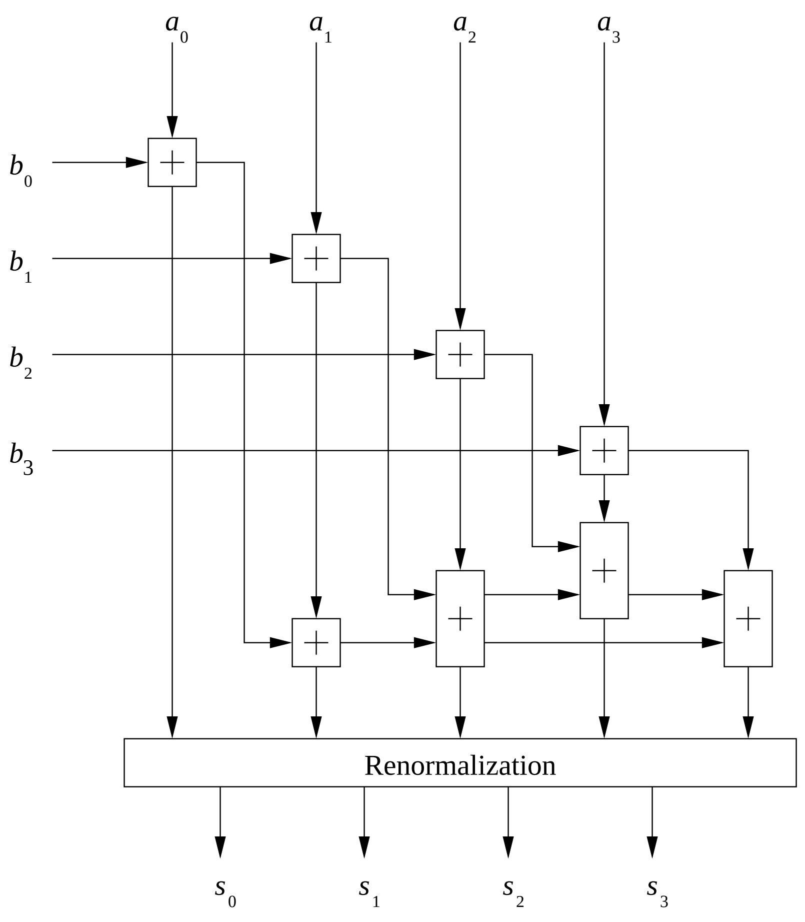

Figure 6: Quad-Double + Quad-Double

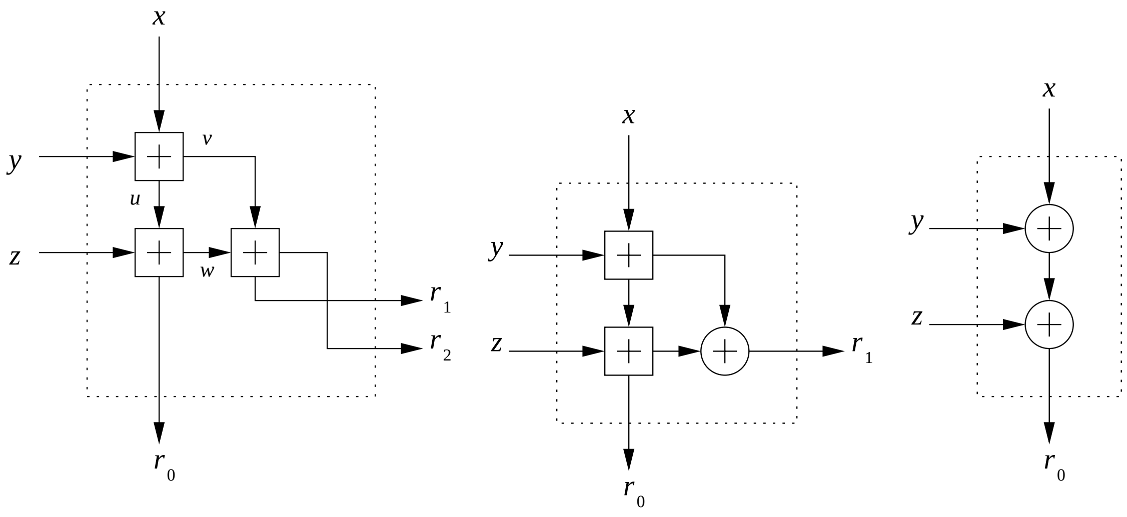

Figure 7: Three-Sums

Note that the two other Three-Sums shown are simplification of the first Three-Sum, where it only computes one or two components, instead of three; thus the same bounds apply.

The above bound is not at all tight; $|r_0|$ is bounded closer to $3M$ (or even $|x|+|y|+|z|$), and this makes the bounds for r_1 and r_2 correspondingly smaller. However, this suffices for the following lemma.

Lemma 12. *The five-term expansion before the renormalization step in the quad-double addition algorithm shown in Figure 6 errs from the true result by less than $\varepsilon_{\text{qd}}M$, where $M = \max(|a|, |b|)$.*

Proof. This can be shown by judiciously applying Lemmas 10 and 11 to all the Two-Sums and Three-Sums in Figure 6. See Appendix A for detailed proof. □

Assuming that the renormalization step works (this remains to be proven), we can then obtain the error bound

$$\text{fl}(a+b) = (1+\delta_1)a + (1+\delta_2)b \qquad \text{with} \quad |\delta_1|, |\delta_2| \leq \varepsilon_{\text{qd}}.$$

Note that the above algorithm for addition is particularly suited to modern processors with instruction level parallelism, since the first four Two-Sums can be evaluated in parallel. Lack of branches before the renormalization step also helps to keep the pipelines full.

Note that the above algorithm does not satisfy the IEEE-style error bound

$$\text{fl}(a+b) = (1+\delta)(a+b) \qquad \text{with} \quad |\delta| \leq 2\varepsilon_{\text{qd}} \text{ or so.}$$

To see this, let $a = (u, v, w, x)$ and $b = (-u, -v, y, z)$, where none of w, x, y, z overlaps and $|w| > |x| > |y| > |z|$. Then the above algorithm produces $c = (w, x, y, 0)$ instead of $c = (w, x, y, z)$ required by the stricter bound.

The second algorithm, due to J. Shewchuk and S. Boldo, computes the first four components of the *result* correctly. Thus it satisfies more strict error bound

$$\text{fl}(a+b) = (1+\delta)(a+b) \qquad \text{with} \quad |\delta| \leq 2\varepsilon_{\text{qd}} \text{ or so.}$$

However, it has a corresponding speed penalty; it runs significantly slower (factor of 2–3.5 slower).

The algorithm is similar to Shewchuk's FAST-EXPANSION-SUM [11, p. 320], where it merge-sorts the two expansions. To prevent components with only a few significant bits to be produced, a double-length accumulator is used so that a component is output only if the inputs gets small enough to not affect it.

Algorithm 13. Assuming that u, v is a two-term expansion, the following algorithm computes the sum $(u, v) + x$, and outputs the significant component s if the remaining components contain more than one double worth of significand. u and v are modified to represent the other two components in the sum.

```
DOUBLE-ACCUMULATE($u, v, x$)
 1.  $(s, v) \leftarrow$ TWO-SUM($v, x$)
 2.  $(s, u) \leftarrow$ TWO-SUM($u, s$)
 3.  if $u = 0$
 4.     $u \leftarrow s$
 5.     $s \leftarrow 0$
 6.  end if
 7.  if $v = 0$
 8.     $v \leftarrow u$
 9.     $u \leftarrow s$
10.     $s \leftarrow 0$
11.  end if
12.  return $(s, u, v)$
```

The accurate addition scheme is given by the following algorithm.

Algorithm 14. This algorithm computes the sum of two quad-double numbers $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$. Basically it merge-sorts the eight doubles, and performs DOUBLE-ACCUMULATE until four components are obtained.

```
QD-ADD-ACCURATE($a, b$)
 1.  $(x_0, x_1, \ldots, x_7) \leftarrow$ MERGE-SORT($a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3$)
 2.  $u \leftarrow 0$
 3.  $v \leftarrow 0$
 4.  $k \leftarrow 0$
 5.  $i \leftarrow 0$
 6.  while $k < 4$ and $i < 8$ do
 7.     $(s, u, v) \leftarrow$ DOUBLE-ACCUMULATE($u, v, x_i$)
 8.     if $s \neq 0$
 9.        $c_k \leftarrow s$
10.        $k \leftarrow k + 1$
11.     end if
12.     $i \leftarrow i + 1$
13.  end while
14.  if $k < 2$ then $c_{k+1} \leftarrow v$
15.  if $k < 3$ then $c_k \leftarrow u$
16.  return RENORMALIZE($c_0, c_1, c_2, c_3$)
```

3.3 Subtraction

Subtraction $a-b$ is implemented as the addition $a+(-b)$, so it has the same algorithm and properties as that of addition. To negate a quad-double number, we can just simply negate each component. On a modern C++ compiler with inlining, the overhead is noticeable but not prohibitive (say 5% or so).

3.4 Multiplication

Multiplication is basically done in a straightforward way, multiplying term by term and accumulating. Note that unlike addition, there are no possibilities of massive cancellation in multiplication, so the following algorithms satisfy the IEEE style error bound $a \otimes b = (1+\delta)(a \times b)$ where δ is bounded by $\varepsilon_{\mathrm{qd}}$.

3.4.1 Quad-Double $\times$ Double

Let $a = (a_0, a_1, a_2, a_3)$ be a quad-double number, and let b be a double precision number. Then the product is the sum of four terms, $a_0 b + a_1 b + a_2 b + a_3 b$. Note that $|a_3| \le \varepsilon^3 |a_0|$, so $|a_3 b| \le \varepsilon^3 |a_0 b|$, and thus only the first 53 bits of the product $a_3 b$ need to be computed. The first three terms are computed exactly using Two-Prod (or Two-Prod-FMA). All the terms are then accumulated in a similar fashion as addition. See Figure 8.

3.4.2 Quad-Double $\times$ Quad-Double

Multiplication of two quad-double numbers becomes a bit complicated, but nevertheless follows the same idea. Let $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$ be two quad-double numbers. Assume (without loss of generality) that a and b are order 1. After multiplication, we need to accumulate 13 terms of order $O(\varepsilon^4)$ or higher.

$$\begin{array}{rcll} a \times b & \approx & a_0 b_0 & O(1) \text{ term} \\ & & + a_0 b_1 + a_1 b_0 & O(\varepsilon) \text{ terms} \\ & & + a_0 b_2 + a_1 b_1 + a_2 b_0 & O(\varepsilon^2) \text{ terms} \\ & & + a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 & O(\varepsilon^3) \text{ terms} \\ & & + a_1 b_3 + a_2 b_2 + a_3 b_1 & O(\varepsilon^4) \text{ terms} \end{array}$$

Note that smaller order terms (such as $a_2 b_3$, which is $O(\varepsilon^5)$) are not even computed, since they are not needed to get the first 212 bits. The $O(\varepsilon^4)$ terms are computed using normal double precision arithmetic, as only their first few bits are needed.

For $i + j \le 3$, let (p_{ij}, q_{ij}) =Two-Prod(a_i, b_j). Then $p_{ij} = O(\varepsilon^{i+j})$ and $q_{ij} = O(\varepsilon^{i+j+1})$. Now there are one term (p_{00}) of order $O(1)$, three (p_{01}, p_{10}, q_{00}) of order $O(\varepsilon)$, five (p_{02}, p_{11}, p_{20}, q_{01}, q_{10}) of order $O(\varepsilon^2)$, seven of order $O(\varepsilon^3)$, and seven of order $O(\varepsilon^4)$. Now we can start accumulating all the terms by their order, starting with $O(\varepsilon)$ terms (see Figure 9).

In the diagram, there are four different summation boxes. The first (topmost) one is Three-Sum, same as the one in addition. The next three are, respectively, Six-Three-Sum (sums six doubles and outputs the first three components), Nine-Two-Sum (sums nine doubles and outputs the first two components), and Nine-One-Sum (just adds nine doubles using normal arithmetic).

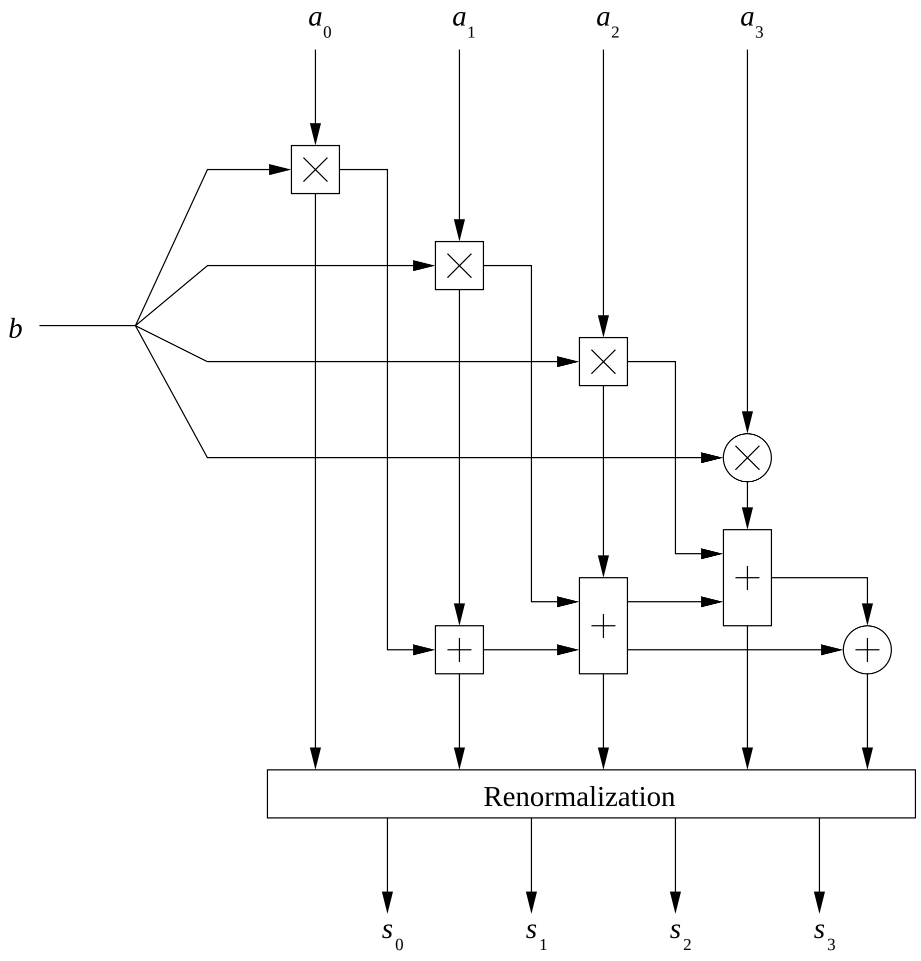

Figure 8: Quad-Double $\times$ Double

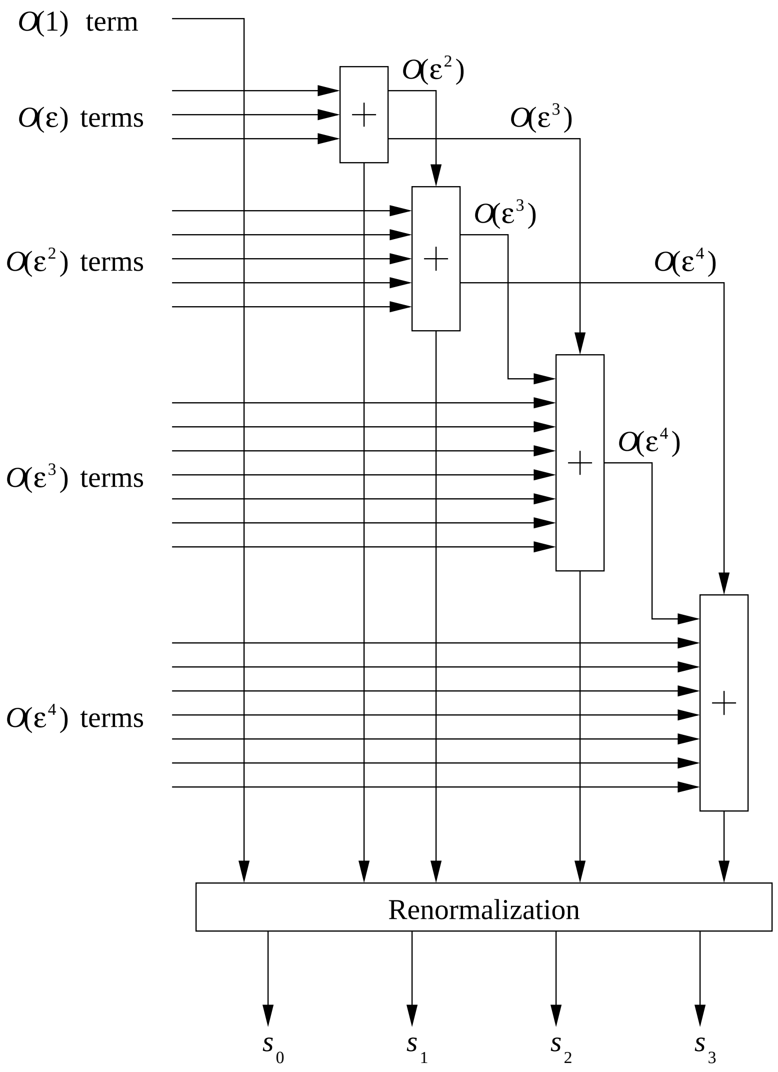

Figure 9: Quad-Double $\times$ Quad-Double accumulation phase

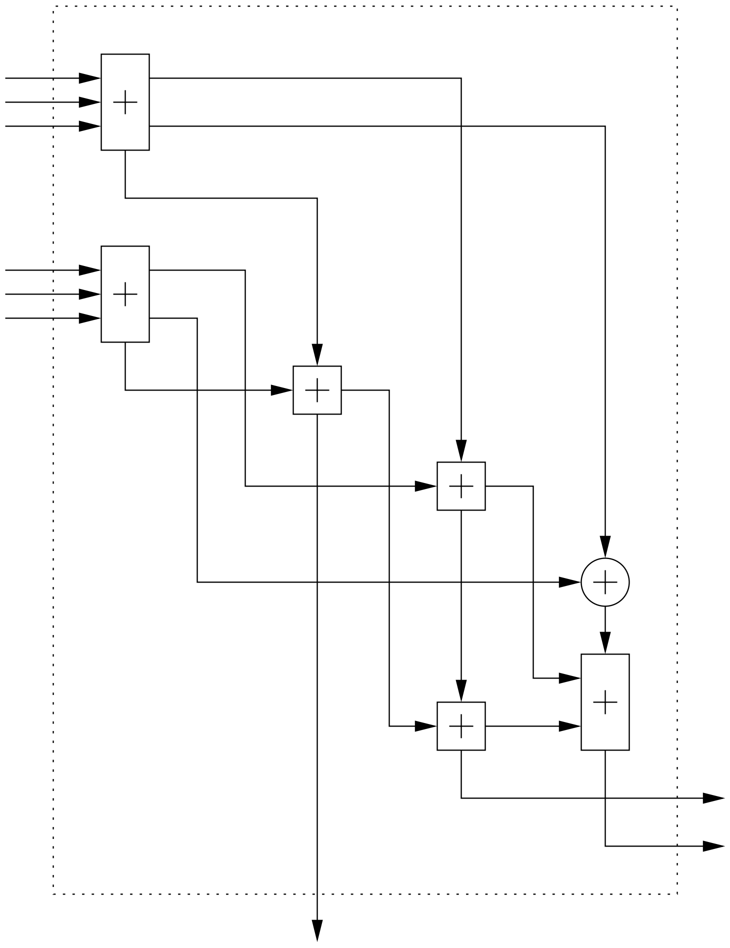

Figure 10: Six-Three-Sum

Six-Three-Sum computes the sum of six doubles to three double worth of accuracy (i.e., to relative error of $O(\varepsilon^3)$). This is done by dividing the inputs into two groups of three, and performing Three-Sum on each group. Then the two sums are added together, in a manner similar to quad-double addition. See Figure 10.

Nine-Two-Sum computes the sum of nine doubles to double-double accuracy. This is done by pairing the inputs to create four double-double numbers and a single double precision number, and performing addition of two double-double numbers recursively until one arrives at a double-double output. The double-double addition (the large square box in the diagram) is the same as David Bailey's algorithm [1]. See Figure 11.

If one wishes to trade few bits of accuracy for speed, we don't even need to compute the $O(\varepsilon^4)$ terms; they can affect the first 212 bits only by carries during accumulation. In this case, we can compute the $O(\varepsilon^3)$ terms using normal double precision arithmetic, thereby speeding up multiplication considerably.

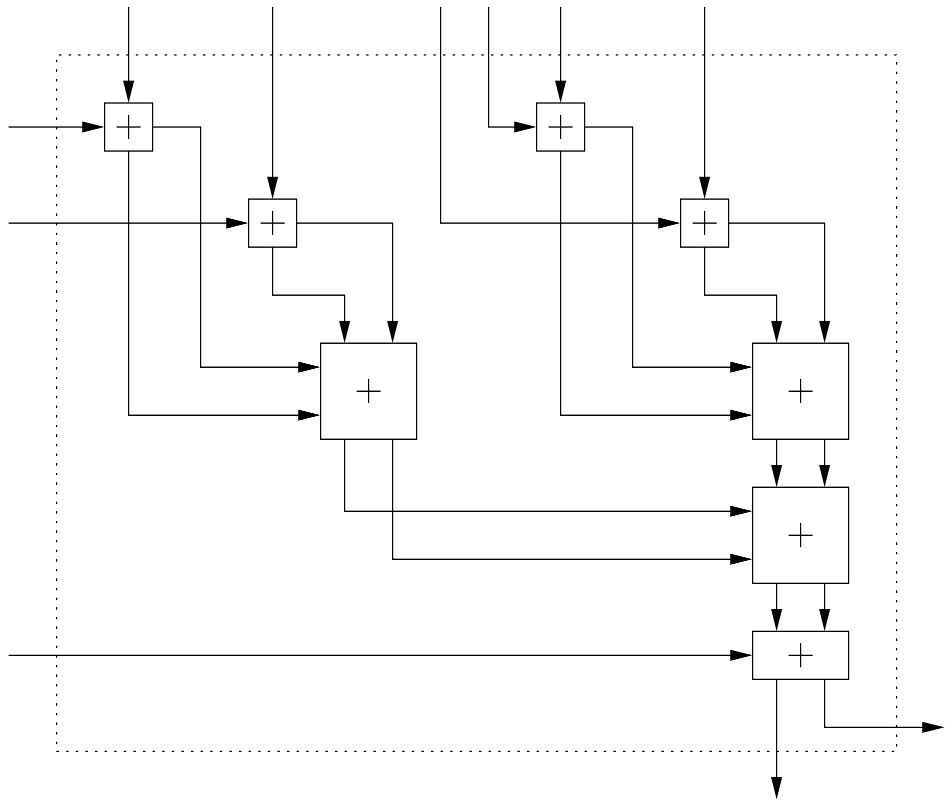

Figure 11: NINE-TWO-SUM

Squaring a quad-double number can be done significantly faster since the number of terms that needs to be accumulated can be reduced due to symmetry.

3.5 Division

Division is done by the familiar long division algorithm. Let $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$ be quad-double numbers. We can first compute an approximate quotient $q_0 = a_0/b_0$. We then compute the remainder $r = a - q_0 \times b$, and compute the correction term $q_1 = r_0/b_0$. We can continue this process to obtain five terms, q_0, q_1, q_2, q_3, and q_4. (only four are needed if few bits of accuracy is not important).

Note that at each step, full quad-double multiplication and subtraction must be done since most of the bits will be canceled when computing q_3 and q_4. The five-term (or four-term) expansion is then renormalized to obtain the quad-double quotient.

4 Algebraic Operations

4.1 N-th Power

N-th Power computes a^n, given a quad-double number a and an integer n. This is simply done by repeated squaring, borrowed from David Bailey [1].

4.2 Square Root

Square root computes $\sqrt{a}$ given a quad-double number a. This is done with Newton iteration on the function

$$f(x) = \frac{1}{x^2} - a$$

which has the roots $\pm a^{-1/2}$. This gives rise to the iteration

$$x_{i+1} = x_i + \frac{x_i(1 - ax_i^2)}{2}.$$

Note that the iteration does not require division of quad-double numbers. (Multiplication by 1/2 can be done component-wise.) Since Newton's iteration is locally quadratically convergent, only about two iterations are required if one starts out with double precision approximation $x_0 = \sqrt{a_0}$. (In the implementation it is done three times.) After $x = a^{-1/2}$ is computed, we perform a multiplication to obtain $\sqrt{a} = ax$.

4.3 N-th Root

N-th Root computes $\sqrt[n]{a}$ given a quad-double number a and an integer n. This is done again by Newton's iteration on the function

$$f(x) = \frac{1}{x^n} - a$$

which has the roots $a^{-1/n}$. This gives rise to the iteration

$$x_{i+1} = x_i + \frac{x_i(1 - ax_i^n)}{n}.$$

Three iterations are performed, although twice is almost sufficient. After $x = a^{-1/n}$ is computed, we can invert to obtain $a^{1/n} = 1/x$.

5 Transcendental Operations

5.1 Exponential

The classic Taylor-Maclaurin series is used to evaluate e^x. Before using the Taylor series, the argument is reduced by noting that

$$e^{kr+m\log 2} = 2^m (e^r)^k,$$

where the integer m is chosen so that $m \log 2$ is closest to x. This way, we can make $|kr| \leq \frac{1}{2}\log 2 \approx 0.34657$. Using $k = 256$, we have $|r| \leq \frac{1}{512}\log 2 \approx 0.001354$. Now e^r can be evaluated using familiar Taylor series. The argument reduction substantially speeds up the convergence of the series, as at most 18 terms are need to be added in the Taylor series.

5.2 Logarithm

Since the Taylor series for logarithm converges much more slowly than the series for exponential, instead we use Newton's iteration to find the zero of the function $f(x) = e^x - a$. This leads to the iteration

$$x_{i+1} = x_i + ae^{-x_i} - 1,$$

which is repeated three times.

5.3 Trigonometrics

Sine and cosine are computed using Taylor series after argument reduction. To compute $\sin x$ and $\cos x$, the argument x is first reduced modulo 2π, so that $|x| \leq \pi$. Now noting that $\sin(y + k\pi/2)$ and $\cos(y + k\pi/2)$ are of the form $\pm \sin y$ or $\pm \cos y$ for all integers k, we can reduce the argument modulo $\pi/2$ so that we only need to compute $\sin y$ and $\cos y$ with $|y| \leq \pi/4$.

Finally, write $y = z + m(\pi/1024)$ where the integer m is chosen so that $|z| \leq \pi/2048 \approx 0.001534$. Since $|y| \leq \pi/4$, we can assume that $|m| \leq 256$. By using a precomputed table of $\sin(m\pi/1024)$ and $\cos(m\pi/1024)$, we note that

$$\sin(z + m\pi/1024) = \sin z \cos(m\pi/1024) + \cos z \sin(m\pi/1024)$$

and similarly for $\cos(z + m\pi/1024)$. Using this argument reduction significantly increases the convergence rate of sine, as at most 10 terms need be added.

Note that if both cosine and sine are needed, then one can compute the cosine using the formula

$$\cos x = \sqrt{1 - \sin^2 x}.$$

The values of $\sin(m\pi/1024)$ and $\cos(m\pi/1024)$ are precomputed by using arbitrary precision package such as MPFUN [2] using the formula

$$\sin\left(\frac{\theta}{2}\right) = \frac{1}{2}\sqrt{2 - 2\cos\theta}$$

$$\cos\left(\frac{\theta}{2}\right) = \frac{1}{2}\sqrt{2 + 2\cos\theta}$$

Starting with $\cos\pi = -1$, we can recursively use the above formula to obtain $\sin(m\pi/1024)$ and $\cos(m\pi/1024)$.

5.4 Inverse Trigonometrics

Inverse trigonometric function arctan is computed using Newton iteration on the function $f(x) = \sin x - a$.

5.5 Hyperbolic Functions

Hyperbolic sine and cosine are computed using

$$\sinh x = \frac{e^x - e^{-x}}{2} \qquad \cosh x = \frac{e^x + e^{-x}}{2}$$

However, when x is small (say $|x| \leq 0.01$), the above formula for sinh becomes unstable, and the Taylor series is used instead.

6 Miscellaneous Routines

6.1 Input / Output

Binary to decimal conversion of quad-double number x is done by determining the integer k such that $1 \leq |x10^{-k}| < 10$, and repeatedly extracting digits and multiplying by 10. To minimize error accumulation, a table of accurately precomputed powers of 10 is used. This table is also used in decimal to binary conversion.

6.2 Comparisons

Since quad-double numbers are fully renormalized after each operation, comparing two quad-double number for equality can be done component-wise. Comparing the size can be done from most significant component first, similar to dictionary ordering of English words. Comparison to zero can be done just by checking the most significant word.

6.3 Random Number Generator

The quad-double random number generator produces a quad-double number in the range $[0, 1)$, uniformly distributed. This is done by choosing the first 212 bits randomly. A 31-bit system-supplied random number generator is used to generate 31 bits at a time, this is repeated $\lceil 212/31 \rceil = 7$ times to get all 212 bits.

7 C++ Implementation

The quad-double library is implemented in ANSI C++, taking full advantage of operator / function overloading and user-defined data structures. The library should compile fine with ANSI Standard compliant C++ compilers. Some of the test codes may not work with compilers lacking full support for templates.

7.1 Usage

Full C++ implementation of double-double library is included as a part of the quad-double library, including full support for mixing three types: double, double-double, and quad-double. In order to use the library, one must include the header file `qd.h` and link the code with the library `libqd.a`. Quad-double variables are declared as `qd_real`, while double-double variables are declared as `dd_real`.

For example,

```
#include <iostream>
#include "qd.h"

using std::cout;
using std::endl;

int main() {
  x86_FIX_DECL;     // see notes on x86 machines below.
  x86_FIX_START;    // see notes on x86 machines below.

  qd_real a = "3.141592653589793238462643383279502884197169399375105820";
  dd_real b = "2.249775724709369995957";
  dd_real r;

  r = a + b;
  cout << "pi + e = " << r << endl;

  r = sqrt(r + 1.0);

  x86_FIX_END;      // see notes on x86 machines below.
  return 0;
}
```

Note that strings must be used to assign to a quad-double (or double-double) numbers; otherwise the double precision approximation is assigned. For example, `a = 0.1` does not assign quad-double precision 0.1, but rather a double precision number 0.1. Instead, use `a = "0.1"`.

Common constants such as π, $\pi/2$, $\pi/4$, e, $\log 2$ are provided as `qd_real::_pi`, `qd_real::_pi2`, `qd_real::_pi4`, `qd_real::_e`, and `qd_real::_log2`. These were computed using an arbitrary precision package (MPFUN++ [6]), and therefore are accurate to the last bit.

7.2 Building

To build the library, edit the files `make.inc` (for platform specific options) and `make.config` (for library build options) and then type `make`. For several platforms, `make.inc` is already supplied as `make.x86` (x86 Linux), `make.sun` (Sparc Solaris), `make.ibm` (IBM RS/6000, AIX), `make.cray` (Alpha UNICOS), and `make.pps` (PowerPC Linux). The compilation process should build the library `libqd.a`

`-DHAS_FMA` Specify this flag if the machine has a fused multiply-accumulate instruction. These include IBM Power Series (including PowerPCs). Be sure to test that FMA instruction is actually generated by the compiler. (Since this flag is dependent on the architecture, it is found in the file `make.inc`. The rest of the flags described below are in `make.config`.)

`-DSLOPPY` Specify this flag if one wishes to trade a few bits of accuracy for speed. This may speed up multiplication and division in exchange for last few bits of accuracy.

`-DACCURATE` Specify this flag if accurate addition is to be used. By using this flag, addition is guaranteed to have the IEEE style error bound $\mathrm{fl}(a+b) = (1+\delta)(a+b)$ for small δ on the order of $\varepsilon_{\mathrm{qd}} = 2^{-211}$.

`-DNO_INLINE` Specify this flag if no inlining of simple function is to be done. Some inlining of internal function are still done, however. This slows down various operations, but speeds up the compilation time considerably, and useful for debugging purposes.

Note that to use the above flags, they must be specified during the compilation of *all* source file that uses the header file `qd.h` or `dd.h`. This is because these flags affect the inline function definitions found in the header files, as well as the library itself.

There are a few other compiler options that maybe needed. These are found in `make.config`.

`-DCRAY` On a Cray machine, this flag might be needed, as different header is needed for the function `copysign`.

`-Dx86` On a x86 machines with extended floating point registers, the default computation is not done in IEEE double, but rather in a 80-bit extended precision. To avoid this, one needs to set the control word of the FPU correctly to round to double precision after every operation. This is done through the three macros `x86_FIX_DECL`, `x86_FIX_START`, and `x86_FIX_END`. The flag enables these macros (see below).

7.3 Note on Intel x86 Processors

Some processors, most notably Intel's x86 family of processors have extended 80-bit IEEE floating point format, on which most arithmetic are done. Since the algorithms described in this paper relies on each arithmetic to be done in IEEE 64-bit doubles, these extended floating point computations must be disabled. This is done by modifying the control word on floating point processor, and can be done through three macros provided:

`x86_FIX_DECL` This macro declares storage space for the control word so that it can be restored by `x86_FIX_END`. This macro must appear before any of `x86_FIX_START` and `x86_FIX_END`.

`x86_FIX_START`	This macro turns off the extended precision computation (it instructs the FPU to round to double after every arithmetic operation). All other flags in the control word is unaffected. This needs to be called before any call to the quad-double library.
`x86_FIX_END`	This macro restores the original control word flags.

7.4 C/Fortran Wrapper

To interface with other languages, traditional C-style functions are also included. These are found in `c_qd.h`, and can be called from C code. To call these from Fortran, some adjustments may need to be made at compilation stage, since various Fortran compilers expects various symbol name formats. These options can be specified in `make.inc`, and are described below.

`-DADD_UNDERSCORE`	Specify this flag if an underscore (_) is needed after every C-style routine name. This flag is needed with Sun F90 compiler.
`-DUPPER_CASE`	Specify this flag if C-style routine names needs to be capitalized. Cray F90 compiler requires this.
`-DCRAY_STRINGS`	Specify this flag if using Cray F90 compiler, they expect different string formats. This primarily affects I/O routines.

To take advantage of generic functions and operator overloading features of Fortran-90, a module file `ddmod.f` and `qdmod.f` is provided as well. In general, linking Fortran code with C++ code is a tricky business; if one uses Fortran compiler to link, one must locate all the C++ standard libraries, and vice versa if one uses a C++ compiler instead.

Also bear in mind that C/Fortran wrappers create an overhead of function calling, since in C++ many basic operations are inlined. This can lead to speed penalty of as much as 30-40%, especially for addition.

7.5 Tests

Some tests are included with the quad-double library, found in `test` directory. These are `qd_test.cc`, `pslq.cc`, `quadt_test.cc`. Furthermore, the quad-double library has been tested against arbitrary precision library such as MPFUN++ [6], and can be found in `devel` directory.

To compile them, type `make tests` from the root directory, or type `make` *test_name* where *test_name* is one of the following:

`qd_test`	This program uses the quad-double library to do various computations, such as computing π using a quadratically convergent algorithm.
`pslq_test`	This is an implementation of PSLQ integer detection algorithm implemented to use the quad-double library. Use of C++ templates allows only one code to be written for three types: double, double-double, and quad-double. It randomly generates a polynomial with integer coefficients, and computes one of its roots. Then using PSLQ algorithm, it tries to reconstruct the polynomial.

`quadt_test` This is an implementation of a quadrature routine. Again it uses C++ templates to allow one code to function on both double-double and quad-double. It computes the integral

$$I = \int_0^\infty \cos 2t \prod_{k=1}^{\infty} \cos \frac{t}{k} \, dt$$

Result of this test shows that amazingly,

$$7.4 \times 10^{-43} < |I - \pi/8| < 7.5 \times 10^{-43},$$

a result that matches the computation using arbitrary precision package.

Each test takes in command line options, these can be viewed with `-help` flag for each program.

8 Performance

Performance of various operations on quad-double numbers on a variety of machines are presented in Table 8. The tested machines are

- Intel Pentium II, 400 MHz, Linux 2.2.16, g++ 2.95.2 compiler, with `-O3 -funroll-loops -finline-functions -mcpu=i686 -march=i686` optimizations.
- Sun UltraSparc 333 MHz, SunOS 5.7, Sun CC 5.0 compiler, with `-xO5 -native` optimizations.
- PowerPC 750 (Apple G3), 266 MHz, Linux 2.2.15, g++ 2.95.2 compiler, with `-O3 -funroll-loops -finline-functions` optimizations.
- IBM RS/6000 Power3, 200 MHz, AIX 3.4, IBM xlC compiler, with `-O3 -qarch=pwr3 -qtune =pwr3 -qstrict` optimizations.

Note: For some reason, GNU C++ compiler (`g++`) has a terrible time optimizing the code for multiplication; it runs more than 15 times slower than the code compiled by Sun's CC compiler.

Most of the routines runs noticeably faster if implemented in C, it seems that C++ operator overloading has some overhead associated with it – most notably excessive copying of quad-double numbers. This occurs because operator overloading does not account for where the result is going to be placed. For example, for the code

```
c = a + b;
```

the C++ compiler often emits the code equivalent to

```
qd_real temp;
temp = operator+(a, b);      // Addition
operator=(c, temp);          // Copy result to c
```

In C, this copying does not happen, as one would just write

```
c_qd_add(a, b, c);           // Put (a+b) into c
```

Operation	Pentium II 400MHz Linux 2.2.16	UltraSparc 333 MHz SunOS 5.7	PowerPC 750 266 MHz Linux 2.2.15	Power3 200 MHz AIX 3.4
Quad-double				
add	0.583	0.580	0.868	0.710
accurate add	1.280	2.464	2.468	1.551
mul	1.965	1.153	1.744	1.131
sloppy mul	1.016	0.860	1.177	0.875
div	5.267	6.440	8.210	6.699
sloppy div	4.080	4.163	6.200	4.979
sqrt	23.646	15.003	21.415	16.174
MPFUN				
add	5.729	5.362	—	4.651
mul	7.624	7.630	—	5.837
div	10.102	10.164	—	9.180

Table 1: Performance of some Quad-Double algorithms on several machines. All measurements are in microseconds. We include the performance of MPFUN [2] as a comparison. Note, we do not have the MPFUN measurements on the PowerPC, because we do not have a Fortran-90 compiler.

where the addition routine knows where to put the result directly. This problem is somewhat alleviated by inlining, but not completely eliminated. There are techniques to avoid these kinds of copying [12], but they have their own overheads associated with them and is not practical for quad-double with only 32 bytes of data[1].

9 Application: Vortex Roll-Up Simulation

The quad-double library was successfully used in a parallel vortex roll-up simulation[2] [3], which uses various transcendental functions as well as basic arithmetics. On a test on the NERSC IBM SP, using 256 Power3 processors, the quad-double version runs about four times as fast as the multiprecision (MPFUN) version, and delivers almost identical results.

10 Future Work

Currently, the basic routines do not have a full correctness proof. The correctness of these routines rely on the fact that renormalization step works; Priest proves that it does work if the input does not overlap by 51 bits and no three components overlap at a single bit. Whether such overlap can occur in any of these algorithm needs to be proved.

There are improvements due in the remainder operator, which computes $a - \text{round}(a/b) \times b$, given quad-double numbers a and b. Currently, the library does the naïve method of just divide,

[1]These techniques are feasible, for larger data structures, such as for much higher precision arithmetics, where copying of data becomes time consuming.

[2]The Fortran code is due to Robert Krasny and Rich Pelz, converted to MPI version by David Bailey.

round, multiply, and subtract. This leads to loss of accuracy when a is large compared to b. Since this routine is used in argument reduction for exponentials, logarithms and trigonometrics, a fix is needed.

A natural extention of this work is to extend the precision beyond quad-double. Algorithms for quad-double additions and multiplication can be extended to higher precisions, however, with more components, asymptotically faster algorithm due to S. Boldo and J. Shewchuk may be preferrable (i.e. Algorithm 14). One limitation these higher precision expansions have is the limited exponent range – same as that of double. Hence the maximum precision is about 2000 bits (39 components), and this occurs only if the first component is near overflow and the last near underflow.

11 Acknowledgements

We thank Jonathan Shewchuk, Sylvie Boldo, and James Demmel for constructive discussions on various basic algorithms. In particular, the accurate version of addition algorithm is due to S. Boldo and J. Shewchuk. Problems with remainder was pointed out by J.Demmel.

A Proof of Quad-Double Addition Error Bound (Lemma 12)

Lemma 12. *The five-term expansion before the renormalization step in the quad-double addition algorithm shown in Figure 6 errs from the true result by less than* $\varepsilon_{\text{qd}}M$*, where* $M = \max(|a|, |b|)$.

Proof. The proof is done by applying Lemmas 10 and 11 to each of Two-Sums and Three-Sums. Let $e_0, e_1, e_2, e_3, t_1, t_2, t_3, x_0, x_1, x_2, x_3, x_4, u, v, w, z, f_1, f_2$, and f_3 be as shown in Figure 12.

We need to show that the five-term expansion $(x_0, x_1, x_2, x_3, x_4)$ errs from the true result by less than $\varepsilon_{\text{qd}}M$, where $M = \max(|a_0|, |b_0|)$. Note that the only place that any error is introduced is in Three-Sum 7 and Three-Sum 8, where lower order terms f_1, f_2, and f_3 are discarded. Hence it suffices to show $|f_1| + |f_2| + |f_3| \le \varepsilon_{\text{qd}}M$.

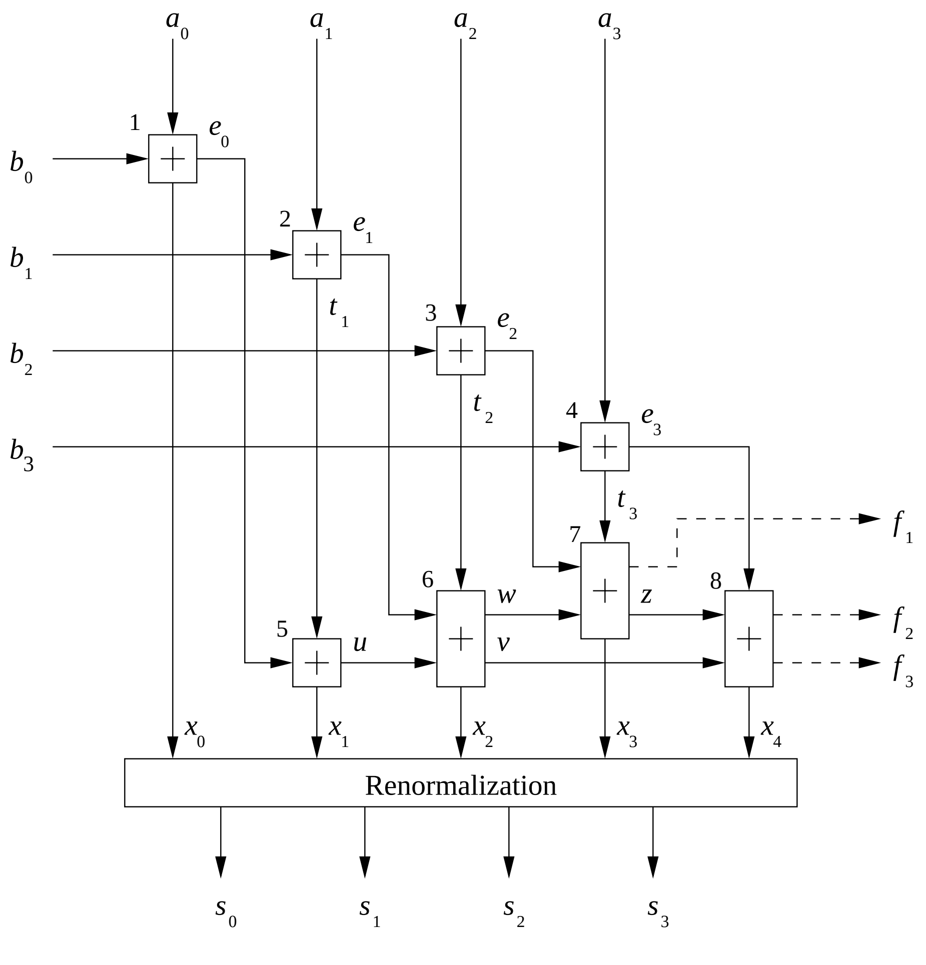

Figure 12: Quad-Double + Quad-Double

First note that $|a_1| \le \varepsilon M$, $|a_2| \le \varepsilon^2 M$,and $|a_3| \le \varepsilon^3 M$, since the input expansions are assumed to be normalized. Similar inequalities applies for expansion b. Applying Lemma 10 to Two-Sums

1, 2, 3, 4, we obtain

$$\begin{array}{rclrcl} |x_0| & \leq & 2M & |e_0| & \leq & 2\varepsilon M \\ |t_1| & \leq & 2\varepsilon M & |e_1| & \leq & 2\varepsilon^2 M \\ |t_2| & \leq & 2\varepsilon^2 M & |e_2| & \leq & 2\varepsilon^3 M \\ |t_3| & \leq & 2\varepsilon^3 M & |e_3| & \leq & 2\varepsilon^4 M. \end{array}$$

Now we can apply Lemma 10 to TWO-SUM 5, to obtain $|x_1| \leq 4\varepsilon M$ and $|u| \leq 4\varepsilon^2 M$. Then we apply Lemma 11 to THREE-SUM 6 to obtain

$$\begin{aligned} |x_2| &\leq 16\varepsilon^2 M \\ |w| &\leq 32\varepsilon^3 M \\ |v| &\leq 32\varepsilon^4 M. \end{aligned}$$

Applying Lemma 11 to THREE-SUM 7, we have

$$\begin{aligned} |x_3| &\leq 128\varepsilon^3 M \\ |z| &\leq 256\varepsilon^4 M \\ |f_1| &\leq 256\varepsilon^5 M. \end{aligned}$$

Finally we apply Lemma 11 again to THREE-SUM 8 to get

$$\begin{aligned} |x_4| &\leq 1024\varepsilon^4 M \\ |f_2| &\leq 2048\varepsilon^5 M \\ |f_3| &\leq 2048\varepsilon^6 M. \end{aligned}$$

Thus we have

$$|f_1| + |f_2| + |f_3| \leq 256\varepsilon^5 M + 2048\varepsilon^5 M + 2048\varepsilon^6 M \leq 2305\varepsilon^5 M \leq \varepsilon_{\text{qd}} M$$

as claimed. □

References

[1] David H. Bailey. A fortran-90 double-double library. Available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`.

[2] David H. Bailey. A fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, 1995. Software available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`.

[3] David H. Bailey, Robert Krasny, and Richard Pelz. Multiple precision, multiple processor vortex sheet roll-up computation. *Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 52–56, 1993.

[4] R. Brent. A Fortran multiple precision arithmetic package. *ACM Trans. Math. Soft.*, 4:57–70, 1978.

[5] K. Briggs. Doubledouble floating point arithmetic. `http://www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`, 1998.

[6] Siddhardtha Chatterjee. MPFUN++: A multiple precision floating point computation package in C++, 1998. Available at `http://www.cs.unc.edu/Research/HARPOON/mpfun++/`.

[7] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.

[8] GMP. `http://www.swox.com/gmp/`.

[9] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1981.

[10] Douglas M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations.* PhD thesis, University of California, Berkeley, November 1992. Available by anonymous FTP at `ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

[11] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[12] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading Massachusetts, 1997.