A Quad-Double Precision Floating-Point Arithmetic Package

Yozo Hida, U.C. Berkeley
Xiaoye S. Li, Lawrence Berkeley Natl. Lab.
David H. Bailey, Lawrence Berkeley Natl. Lab.

Presentation given by David H. Bailey
`http://www.nersc.gov/~dhbailey`

Applications of extended precision arithmetic

1. Determining the extent of numerical round-off error in scientific simulations.
2. Overcoming numerical round-off error in scientific simulations.
3. Resolving uncertainties in computational geometry.
4. Computing numerical integrals to high precision.
5. Recovering integer relations in real numbers.
6. Recognizing numerical constants.
7. Cryptography.

Available software for extended precision arithmetic

1. Double-double arithmetic in some Fortran systems (`real*16`).
   (a) Advantages: reliable, accurate, full functionality.
   (b) Disadvantages: performance is often poor; only double-double.
2. Brigg's double-double software.
   (a) Advantages: reliable, accurate.
   (b) Disadvantages: performance is fair; only double-double; no high-level language support.
3. Extended precision facilities in Maple and Mathematica.
   (a) Advantages: scalable precision, very complete functionality.
   (b) Disadvantages: performance is poor; somewhat expensive.
4. Multiple precision software packages (DHB's `mpfun`, Brent's `mp`, Gnu's `gnump`).
   (a) Advantages: scalable precision, high-level language support (DHB).
   (b) Disadvantages: performance is poor for modest precision levels.

A New Quad-Double, Double-Double Package

- Written mostly in C and C++.
- Targeted exclusively to IEEE compliant computer systems.
- Includes basic arithmetic operations, plus binary-decimal, square root and common transcendental functions.
- Includes C++ and Fortran-90 "wrappers" that permit ordinary programs to utilize these routines with only minor changes to conventional programs.

Application: Climate Modeling

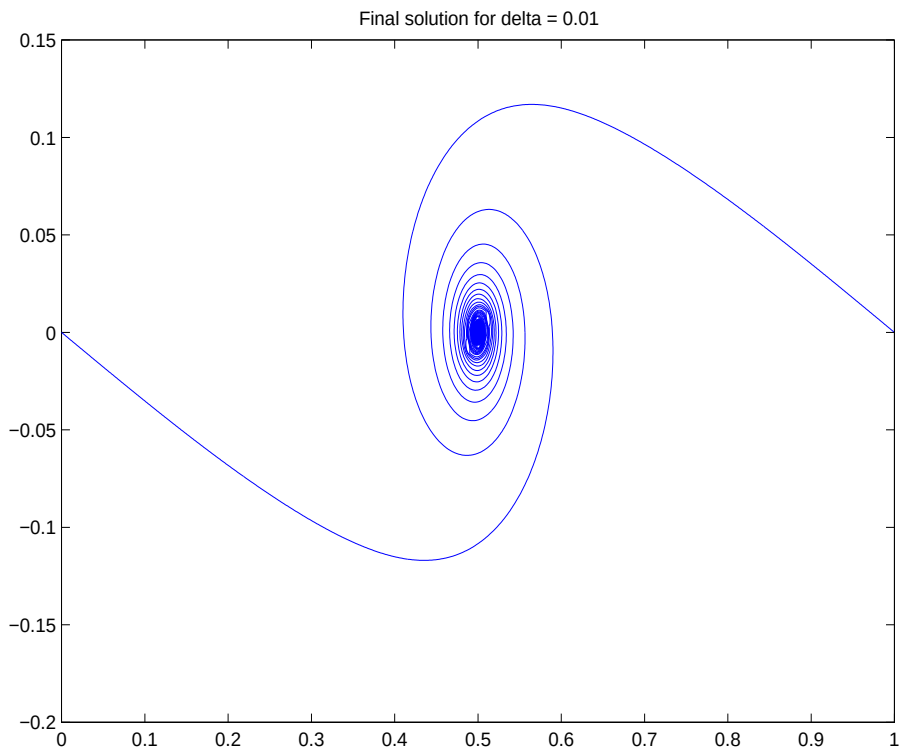
Many climate modeling codes are beset with severe numerical non-reproducibility:

- Codes ported to different vendor systems.
- Codes run on different systems of the same vendor.
- Codes run on the same system, but with different numbers of processors.
- Is there a bug? No way to tell.

These difficulties have been dramatically reduced by using double-double routines in just one key inner product subroutine.

Application: Vortex roll-up simulation

Vortex roll-up simulation runs 5 times faster with quad-double package than with DHB's mpfun package. Has saved thousands of CPU-hours on a highly parallel supercomputer.



Application: Numerical quadrature

Consider the constant

$$\alpha = \int_0^\infty \cos(2t) \left(\prod_{k=1}^{\infty} \cos(t/k) \right) dt$$

This integral can be evaluated numerically by means of tanh-sinh quadrature and an approximation to the tail of the infinite product.

Result using double precsion (approx. 16 decimal digits):

$$\begin{aligned} \alpha &= 0.392699081698724 \\ \pi/8 &= 0.392699081698724 \end{aligned}$$

Is this a coincidence?

Application: Numerical quadrature

Consider the constant

$$\alpha = \int_0^\infty \cos(2t) \left(\prod_{k=1}^{\infty} \cos(t/k) \right) dt$$

This integral can be evaluated numerically by means of tanh-sinh quadrature and an approximation to the tail of the infinite product.

Results using double-double precision (approx. 32 digits):

$$\begin{aligned} \alpha &= 0.39269908169872415480783042290993 \\ \pi/8 &= 0.39269908169872415480783042290993 \end{aligned}$$

These values still appear equal.

Application: Numerical quadrature

Consider the constant

$$\alpha \;=\; \int_0^\infty \cos(2t) \left(\prod_{k=1}^{\infty} \cos(t/k) \right) \, dt$$

This integral can be evaluated numerically by means of tanh-sinh quadrature and an approximation to the tail of the infinite product.

Results using quad-double precision (approx. 64 digits):

$$\begin{aligned} \alpha &= 0.3926990816987241548078304229099378605246454341872315959268122874 \\ \pi/8 &= 0.3926990816987241548078304229099378605246461749218882276218680740 \end{aligned}$$

These results differ in the 42nd decimal place!

Conclusion: $\alpha \neq \pi/8$.

Adding two IEEE double numbers exactly

(Priest) 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)

(Knuth) 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)

Multiplying two IEEE double numbers exactly

(Shewchuk, Dekker) The following algorithm splits a 53-bit IEEE double precision floating point number into a_{hi} and a_{lo}, each with 27 bits of significand, such that $a = a_{\text{hi}} + a_{\text{lo}}$.

SPLIT(a)

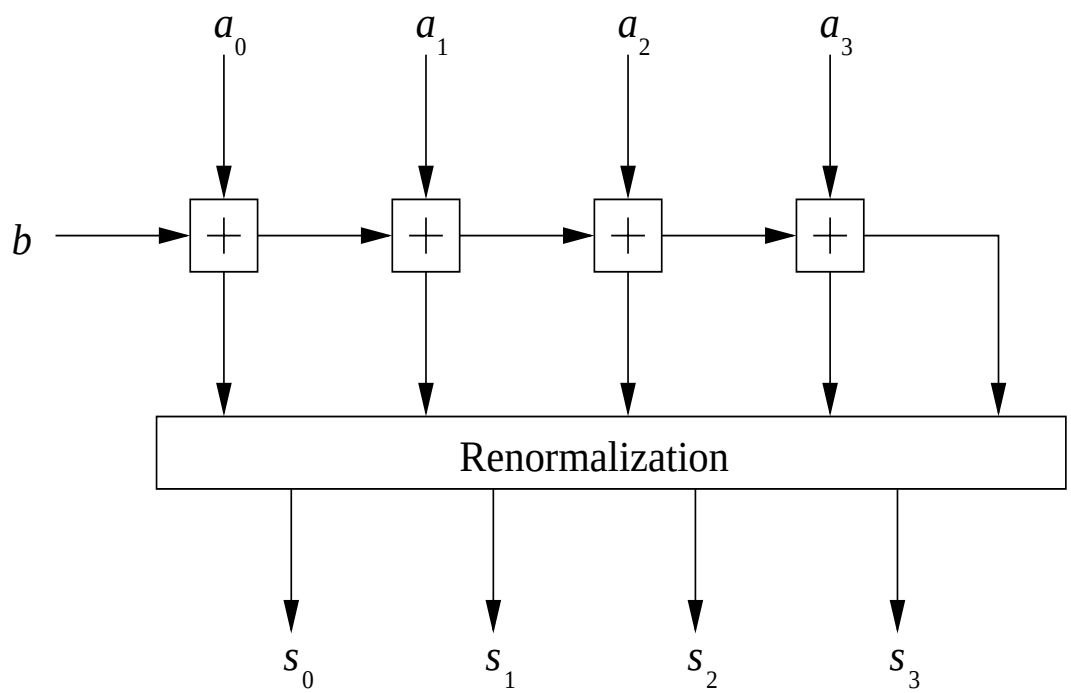
1. $t \leftarrow (2^{27} + 1) \otimes a$
2. $a_{\text{hi}} \leftarrow t \ominus (t \ominus a)$
3. $a_{\text{lo}} \leftarrow a \ominus a_{\text{hi}}$
4. **return** $(a_{\text{hi}}, a_{\text{lo}})$

(Shewchuk, Veltkamp) The following algorithm computes $p = \text{fl}(a \times b)$ and $e = \text{err}(a \times b)$.

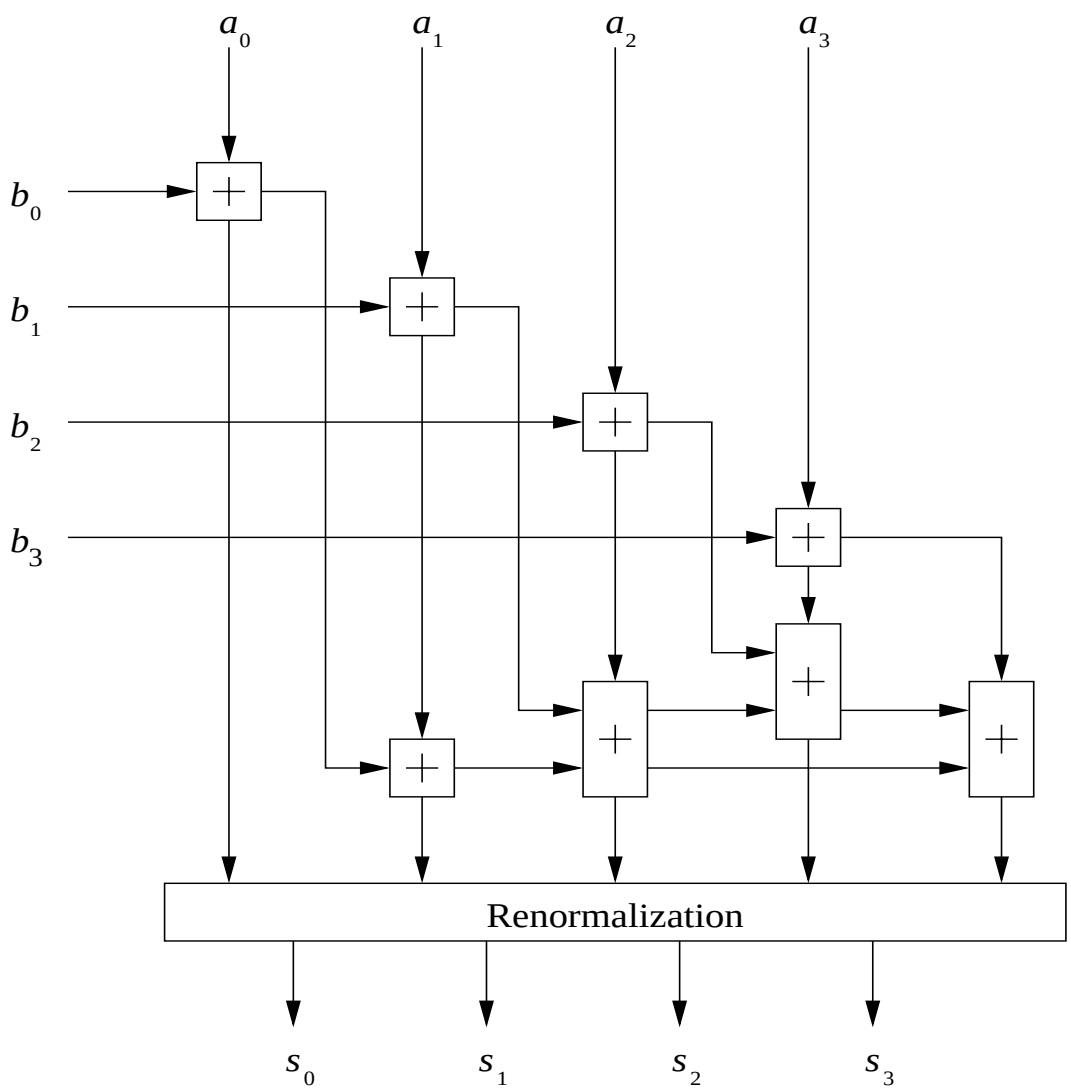
TWO-PROD(a, b)

1. $p \leftarrow a \otimes b$
2. $(a_{\text{hi}}, a_{\text{lo}}) \leftarrow$ SPLIT(a)
3. $(b_{\text{hi}}, b_{\text{lo}}) \leftarrow$ SPLIT(b)
4. $e \leftarrow ((a_{\text{hi}} \otimes b_{\text{hi}} \ominus p) \oplus a_{\text{hi}} \otimes$
$b_{\text{lo}} \oplus a_{\text{lo}} \otimes b_{\text{hi}}) \oplus a_{\text{lo}} \otimes b_{\text{lo}}$
5. **return** (p, e)

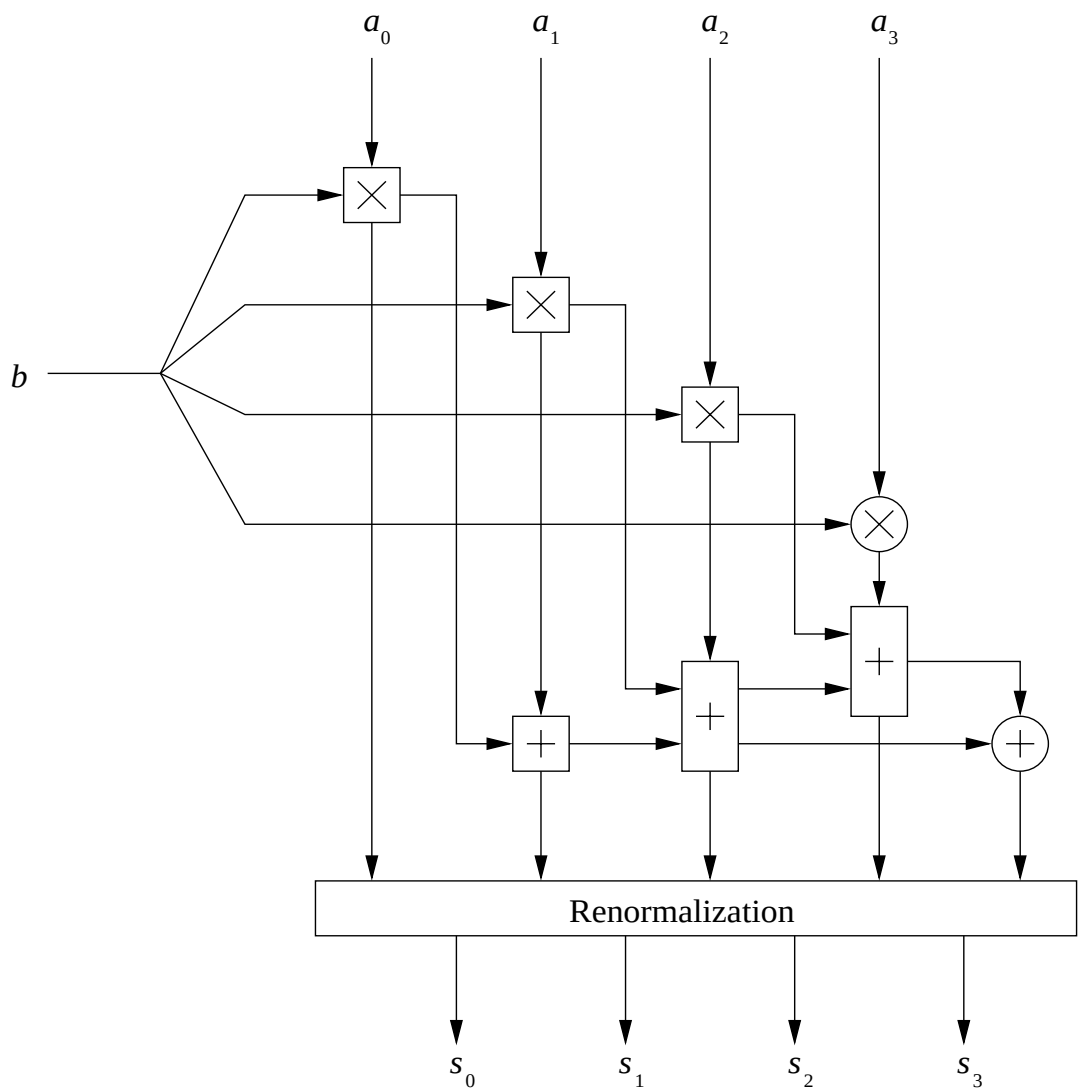
Quad-double + double



Quad-double + quad-double



Quad-double × double



Transcendental functions

- sqrt: Employ Newton iterations (double, double-double, then quad-double).
- exp: Reduce argument to $(-\frac{1}{2}\log 2 < t < \frac{1}{2}\log 2)$, then apply Taylor series.
- log: Employ Newton iterations, based on exp calculations.
- cos, sin: Reduce argument to within an interval of size $\pi/1024$ in the unit circle, then apply Taylor series.
- arccos, arcsin: Employ Newton iterations, based on cos, sin calculations.

High-level language support

High-level translation modules ("wrappers") permit one to utilize double-double and quad-double in ordinary C++ and Fortran-90 programs, with only minor modifications to source code.

Fortran-90 example:

```
program testdd
use qdmodule
type (qd_real) a, b, c
a = -1.d0
b = 2.d0
c = acos (a) * sqrt (b)
call qdwrite (6, c)
stop
end
```

This program produces the result

4.4428829381583662470158809900606936986146216893756902230853956 07E0

which is $\pi * \sqrt{2}$, correct to 63 decimal digits.

Performance (average run times in microseconds)

Operation	Pentium II	UltraSparc	PowerPC	Power3
`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

How Do I Obtain the New Software?

NOTE: This information is not in proceedings.

Software and documentation is available at:

`http://www.nersc.gov/~dhbailey/mpdist`

C++ version requires a C++ compiler.
Fortran-90 version requires a Fortran-90 and a C++ compiler.

Send questions or comments to:

`dhbailey@lbl.gov`