Computer arithmetic and the MathCW library

Nelson H. F. Beebe

Research Professor
University of Utah
Department of Mathematics, 110 LCB
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA

Email: beebe@math.utah.edu, beebe@acm.org,
beebe@computer.org (Internet)
WWW URL: http://www.math.utah.edu/~beebe
Telephone: +1 801 581 5254
FAX: +1 801 581 4148

13 December 2008
Common misconceptions about computer arithmetic

- Integer arithmetic is always exact
- Integer overflows are caught
- Floating-point arithmetic is *fuzzy*
- Floating-point equality comparisons are unreliable
- Floating-point precision and range are adequate for everyone
- Rounding errors accumulate
- Computers execute arithmetic code in the order and precision in which it is written
- Underflows are harmless
- Overflows are disastrous
- Sign of zero does not matter
- Arithmetic exceptions should cause job termination
Infamous disasters in computer arithmetic

- USS Yorktown nuclear-weapons carrier dead in water in 1997 for three hours after integer overflow in spreadsheet shut down software control (potential cost: loss of a war)
- US Patriot missiles fail to shoot down Iraqi Scuds in 1990 Gulf War due to timer counter integer overflow
- European Space Agency Ariane 5 missile loss off West Africa in 1996 due to arithmetic overflow in floating-point to integer conversion in guidance system (cost $\approx$ US$1B$)
- Intel Pentium floating-point divide flaw in 1994 (US$400M – US$600M)
- New York and Vancouver Stock Exchange shutdowns (cost: $n \times$ US$10M$)
German state election in Schleswig–Holstein reversed in 1992 because of rounding errors in party vote percentages

The US presidential elections in 2000 and 2004, the State of Washington gubernatorial election in 2004, and the Mexico presidential election in 2007, were so close that even minor errors in counting could have changed the results

Some US electronic voting machines in 2004 were found to have counted *backwards*, subtracting votes, either because of counter overflow and wraparound, or malicious tampering with the software.
Historical floating-point arithmetic

- Konrad Zuse’s Z1, Z3, and Z4 (1936–1945): 22-bit (Z1 and Z3) and 32-bit Z4 with exponent range of $2^{\pm 63} \approx 10^{\pm 19}$
- Burks, Goldstine, and von Neumann (1946) argued against floating-point arithmetic
- *It is difficult today to appreciate that probably the biggest problem facing programmers in the early 1950s was scaling numbers so as to achieve acceptable precision from a fixed-point machine*, Martin Campbell-Kelly (1980)
- IBM mainframes from mid-1950s supplied floating-point arithmetic
- IEEE 754 Standard (1985) proposed a new design for binary floating-point arithmetic that has since been widely adopted
- IEEE 754 design first implemented in Intel 8087 coprocessor (1980)
Historical flaws on some systems

Floating-point arithmetic can make error analysis difficult, with behavior like this in some older designs:

- $u \neq 1.0 \times u$
- $u + u \neq 2.0 \times u$
- $u \times 0.5 \neq u/2.0$
- $u \neq v$ but $u - v = 0.0$, and $1.0/(u - v)$ raises a zero-divide error
- $u \neq 0.0$ but $1.0/u$ raises a zero-divide error
- $u \times v \neq v \times u$
- underflow wraps to overflow, and vice versa
- division replaced by reciprocal approximation and multiply
- poor rounding practices increase cumulative rounding error
IEEE 754 binary floating-point arithmetic

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>22</td>
</tr>
</tbody>
</table>

- $s$ is sign bit (0 for $+$, 1 for $-$)
- $exp$ is unsigned biased exponent field
- Smallest exponent: zero and *subnormals* (formerly, *denormalized*)
- Largest exponent: Infinity and NaN (Not a Number)
- Significand has implicit leading 1-bit in all but 80-bit format
- $\pm 0, \pm \infty$, signaling and quiet NaN
IEEE 754 binary floating-point arithmetic

- NaN from 0/0, $\infty - \infty$, $f(\text{NaN})$, $x \text{ op } \text{NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers
- $\pm\infty$ from big/small, including nonzero/zero
- precisions in bits: 24, 53, 64, 113, 235
- approximate precisions in decimal digits: 7, 15, 19, 34, 70
- approximate ranges (powers of 10): \([-45,38], [-324,308], [-4951,4932], [4966,4932], [-315723,315652]\)
nonstop computing model
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
- no language support for advanced features until 1999 ISO C Standard
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
- no language support for advanced features until 1999 ISO C Standard
- some architectures implement only subsets (e.g., no subnormals, or only one rounding mode, or only one kind of NaN, or in embedded systems, neither Infinity nor NaN)
IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: **underflow**, **overflow**, **zero divide**, **invalid**, and **inexact**
- four rounding modes: **to-nearest-with-ties-to-even** (default), **to-plus-infinity**, **to-minus-infinity**, and **to-zero**
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
- no language support for advanced features until 1999 ISO C Standard
- some architectures implement only subsets (e.g., no subnormals, or only one rounding mode, or only one kind of NaN, or in embedded systems, neither Infinity nor NaN)
- some platforms have nonconforming rounding behavior
Why the base matters

- accuracy and run-time cost of conversion between internal and external (usually decimal) bases
- effective precision varies when the floating-point representation uses a radix larger than 2 or 10
- reducing the exponent width makes digits available for increased precision
- for a fixed number of exponent digits, larger bases provide a wider exponent range, and reduce incidence of rounding
- for a fixed storage size, granularity (the spacing between successive representable numbers) increases as the base increases
- in the absence of underflow and overflow, multiplication by a power of the base is an exact operation, and this feature is essential for many computations, in particular, for accurate elementary and special functions
Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal 0x1.ccccccccccccccccccccccccccccccccccccccc...p-1)
- 5% sales-tax example: binary arithmetic: 0.70 × 1.05 = 0.734999999..., which rounds to 0.73; correct decimal result 0.735 may round to 0.74
- Goldberg (1967) and Matula (1968) showed how many digits needed for exact round-trip conversion
- exact conversion may require many digits: more than 11500 decimal digits for binary-to-decimal conversion of 128-bit format,
- base-conversion problem not properly solved until 1990s
- few (if any) languages guarantee accurate base conversion
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range (10^{±999999999})
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision ($10^9$ digits) and huge exponent range ($10^{\pm999999999}$)
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)

- Business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
- Four additional rounding modes for legal/tax/financial requirements
- Integer, rather than fractional, coefficient means redundant representation, but allows emulating fixed-point arithmetic
- Trailing zeros significant: they change quantization
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range (10^{±999999999})
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range (10^{±999999999})
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
- Business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range (10^{±999999999})
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
- Business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
- Four additional rounding modes for legal/tax/financial requirements
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision \((10^9\) digits) and huge exponent range \((10^{±999999999})\)
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
- Business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
- Four additional rounding modes for legal/tax/financial requirements
- `integer`, rather than `fractional`, coefficient means redundant representation, but allows emulating fixed-point arithmetic
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range (10^{±999999999})
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
- Business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
- Four additional rounding modes for legal/tax/financial requirements
- Integer, rather than fractional, coefficient means redundant representation, but allows emulating fixed-point arithmetic
- Quantization primitives can distinguish between 1, 1.0, 1.00, 1.000, etc.
Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range (10^{±999,999,999})
- IBM decNumber library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
- Business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
- Four additional rounding modes for legal/tax/financial requirements
- integer, rather than fractional, coefficient means redundant representation, but allows emulating fixed-point arithmetic
- Quantization primitives can distinguish between 1, 1.0, 1.00, 1.000, etc.
- Trailing zeros significant: they change quantization
### Decimal floating-point arithmetic

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>cf</th>
<th>ec</th>
<th>cc</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit 0</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>12</td>
<td>63</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>127</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>22</td>
<td>255</td>
</tr>
</tbody>
</table>

- IBM Densely-Packed Decimal (DPD) and Intel Binary-Integer Decimal (BID) in 32-bit, 64-bit, 128-bit, and 256-bit formats provide \(3n+1\) digits: 7, 16, 34, and 70.
- Wider exponent ranges in decimal than binary: \([-101, 97]\), \([-398, 385]\), \([-6176, 6145]\), and \([-1572863, 1572865]\).
- \(cf\) (combination field), \(ec\) (exponent continuation field), \((cc)\) (coefficient combination field).
- Infinity and NaN recognizable from first byte (not true in binary formats).
Need *much* more than ADD, SUB, MUL, and DIV operations

- mathcw library provides full C99 repertoire, including printf and scanf families, plus hundreds more [but not functions of type complex]

- code is portable across all current platforms, and several historical ones (PDP-10, VAX, S/360, ...)

- supports *six* binary and *four* decimal floating-point datatypes

- separate algorithms cater to base variations: 2, 8, 10, and 16

- pair-precision functions for even higher precision

- fused multiply-add (FMA) via pair-precision arithmetic

- programming languages: Ada, C, C++, C#, Fortran, Java, Pascal

- scripting languages: gawk, hoc, lua, mawk, nawk
Virtual platforms

MMIX STATION: NEW AND IMPROVED FOR 2009!

MMIX INSIDE!

Whatever your figurework requirements, there's a MMIX Station exactly suited to your needs. Designed by Prof. D. E. Knuth of Stanford, this ingenious all electric machine has more than two hundred registers and is the fastest producer of useful, accurate answers just when business is needing more and more figures. Available in a broad color range.
Fused multiply-add

- $a \times b + c$ is a common operation in numerical computation (e.g., nested Horner polynomial evaluation and matrix/vector arithmetic)
- `fma(a, b, c)` computes $a \times b + c$ with exact double-length product and addition with one rounding
- `fma(a, b, c)` recovers error in multiplication:

\[
\begin{align*}
    d & \leftarrow \text{fl}(a \times b) \\
    \text{err} & \leftarrow \text{fma}(a, b, -d) \\
    a \times b & = \text{fl}(a \times b) + \text{err}
\end{align*}
\]

- `fma()` in some native hardware [IBM PowerPC (32-bit and 64-bit only), HP/Intel IA-64 (32-bit, 64-bit, 80-bit), and some HP PA-RISC and MIPS R8000]
- `fma()` is a critical component of many algorithms for accurate computation
Markstein’s book shows how fma() leads to accurate and compact elementary functions, as well as provably-correctly-rounded software division and square root.

Nievergelt [TOMS 2003] proved that fma() leads to matrix arithmetic provably accurate to the penultimate digit.

See fparith.bib for many other applications of fma().
The MathCW Library

\[ \times \times 10 \approx 3660 \text{ functions} \approx 2M \text{ lines} \]
MathCW design goals

- Complete C99 and POSIX support with many enhancements
- Portable across past, present, and future platforms
- Binary *and* decimal arithmetic fully supported
- Ten floating-point formats, including single (7D), double (16D), quadruple (34D), and octuple precision (70D)
- IEEE 754 (1985 and 2008) and 854 feature access
- Free software and documentation under GNU licenses
- Documented in manual pages and forthcoming treatise
- Interactive access in *hoc*
- Interfaces to Ada, C, C++, C#, Fortran, Java, Pascal
- Replace native binary arithmetic in all scripting languages with high-precision decimal arithmetic
MathCW design goals [cont.]

- Separate data from code
- Abstract data types: \texttt{fp\_t} and \texttt{hp\_t}, and \texttt{FP()} and \texttt{FUNC()} wrappers
- Make algorithm files base-, precision-, and range-independent when feasible
- No platform software configuration needed
- Offer static, shared, fat (multi-architecture), and wrapper libraries
- Provide high relative accuracy: target is two \textit{ulps} (units in the last place), but exponential, log, root, and trigonometric families return results that are (almost) always correctly rounded (and much better than Intel IA-32 rounding of 80-bit to 64-bit results)
- Provide \textit{exact} function argument reduction [Payne/Hanek at DEC (1982) and Corbett at Berkeley (1983)]
Error plots

Errors in `expm1f()`

Errors in `expm1df()`

Errors in `expm1()`

Errors in `expm1d()`

Error plots

![Error plots for bks0f()](image1)

![Error plots for bks0df()](image2)

![Error plots for bks0()](image3)

![Error plots for bks0d()](image4)
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- complete real C99 mathematical library repertoire
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- complete real C99 mathematical library repertoire
- complete IEEE 754/854 feature access
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- complete real C99 mathematical library repertoire
- complete IEEE 754/854 feature access
- printf() formatting extended for binary, octal, hexadecimal, based, plus digit grouping
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- complete real C99 mathematical library repertoire
- complete IEEE 754/854 feature access
- printf() formatting extended for binary, octal, hexadecimal, based, plus digit grouping
- (potentially) multilingual
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- complete real C99 mathematical library repertoire
- complete IEEE 754/854 feature access
- printf() formatting extended for binary, octal, hexadecimal, based, plus digit grouping
- (potentially) multilingual
- extensive help facility
hoc live demo

- hoc ≈ interactive C without declarations, datatypes, or data structures
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- *complete* real C99 mathematical library repertoire
- *complete* IEEE 754/854 feature access
- `printf()` formatting extended for binary, octal, hexadecimal, based, plus digit grouping
- (potentially) multilingual
- extensive help facility
- *support for external libraries written in hoc*
hoc live demo

- hoc \approx \text{interactive C without declarations, datatypes, or data structures}
- hoc32, hoc36, hoc64, hoc72, hoc80, hoc128, hocd32, hocd64, hocd128
- \textit{complete} real C99 mathematical library repertoire
- \textit{complete} IEEE 754/854 feature access
- \texttt{printf()} formatting extended for binary, octal, hexadecimal, based, plus digit grouping
- (potentially) multilingual
- extensive help facility
- support for external libraries written in hoc
- (possibly) dynamic load library support
Q&A and discussion