New directions in floating-point arithmetic

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

26 September 2007

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 $2^{\pm 63} \approx 10^{\pm 19}$

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

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)

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

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

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:

- $u \neq 1.0 \times u$

Historical flaws on some systems

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

- $u \neq 1.0 \times u$
- $u + u \neq 2.0 \times u$

Historical flaws on some systems

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

- $u \neq 1.0 \times u$
- $u + u \neq 2.0 \times u$
- $u \times 0.5 \neq u / 2.0$

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

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

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
- $u \times v \neq v \times u$

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

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

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

	s	exp	significand		
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)

IEEE 754 binary floating-point arithmetic

	s	exp	significand		
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)
- *exp* is unsigned biased exponent field

IEEE 754 binary floating-point arithmetic

	s	exp	significand		
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for +, 1 for −)
- *exp* is unsigned biased exponent field
- smallest exponent: zero and *subnormals* (formerly, *denormalized*)

IEEE 754 binary floating-point arithmetic

s	exp	significand

bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *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)

IEEE 754 binary floating-point arithmetic

	s	exp	significand		
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *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

IEEE 754 binary floating-point arithmetic

s	exp	significand

bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *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
- ± 0, $\pm\infty$, signaling and quiet NaN

IEEE 754 binary floating-point arithmetic

- NaN from 0/0, $\infty - \infty$, $f(\mathrm{NaN})$, x op NaN, ...

IEEE 754 binary floating-point arithmetic

- NaN from 0/0, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{~op~NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{\ op\ NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers
- $\pm\infty$ from big/small, including nonzero/zero

IEEE 754 binary floating-point arithmetic

- NaN from 0/0, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{~op~} \mathrm{NaN}$, …
- 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

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{~op~NaN}$, ...
- 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

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{~op~NaN}$, ...
- 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): $[-45, 38]$, $[-324, 308]$, $[-4951, 4932]$, $[4966, 4932]$, $[-315\,723, 315\,652]$

IEEE 754 binary floating-point arithmetic

- 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

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

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

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

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
- for a fixed storage size, granularity (the spacing between successive representable numbers) increases as the base increases

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
- 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.cccccccccccccccccccccccc...p-1`)

Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal `0x1.cccccccccccccccccccccccc...p-1`)
- 5% sales-tax example: binary arithmetic: $0.70 \times 1.05 = 0.734999999\ldots$, which rounds to 0.73; correct decimal result 0.735 may round to 0.74

Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal `0x1.cccccccccccccccccccccccc...p-1`)
- 5% sales-tax example: binary arithmetic: $0.70 \times 1.05 = 0.734999999\ldots$, 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

Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal `0x1.cccccccccccccccccccccccc...p-1`)
- 5% sales-tax example: binary arithmetic: $0.70 \times 1.05 = 0.734999999\ldots$, 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 11 500 decimal digits for binary-to-decimal conversion of 128-bit format,

Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal `0x1.cccccccccccccccccccccc...p-1`)
- 5% sales-tax example: binary arithmetic: $0.70 \times 1.05 = 0.734999999\ldots$, 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 11 500 decimal digits for binary-to-decimal conversion of 128-bit format,
- base-conversion problem not properly solved until 1990s

Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal `0x1.cccccccccccccccccccccccc...p-1`)
- 5% sales-tax example: binary arithmetic: $0.70 \times 1.05 = 0.734999999\ldots$, 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 11 500 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 ($10^{\pm 999\,999\,999}$)

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 ($10^{\pm 999\,999\,999}$)
- IBM `decNumber` library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (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 ($10^{\pm 999\,999\,999}$)
- 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

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 ($10^{\pm 999\,999\,999}$)
- 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

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 ($10^{\pm 999\,999\,999}$)
- 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

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 ($10^{\pm 999\,999\,999}$)
- 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

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 ($10^{\pm 999\,999\,999}$)
- 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.

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 ($10^{\pm 999\,999\,999}$)
- 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

	s	**cf**	**ec**	**cc**		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- 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

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- 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 $[-1\,572\,863, 1\,572\,865]$

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- 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 $[-1\,572\,863, 1\,572\,865]$
- *cf* (combination field), *ec* (exponent continuation field), (cc) (coefficient combination field)

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- 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 $[-1\,572\,863, 1\,572\,865]$
- *cf* (combination field), *ec* (exponent continuation field), (cc) (coefficient combination field)
- Infinity and NaN recognizable from first byte (not true in binary formats)

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- code is portable across all current platforms, and several historical ones (PDP-10, VAX, S/360, ...)

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- 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

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- 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

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- 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

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- 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

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- 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

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more
- 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