Extending \TeX\ and METAfont with Floating-Point Arithmetic

Nelson H. F. Beebe

Department of Mathematics
University of Utah
Salt Lake City, UT 84112-0090
USA

Arithmetic in \TeX\ and METAfont

- Binary integer arithmetic with \( \geq 32 \) bits (\TeX\ \texttt{\count} registers)
- Fixed-point arithmetic with sign bit, overflow bit, \( \geq 14 \) integer bits, and \( 16 \) fractional bits (\TeX\ \texttt{\dimen}, \texttt{\muskip}, and \texttt{\skip} registers)
- Overflow detected on division and multiplication but not on addition (flaw (NHFB), feature (DEK))
- Gyrotations sometimes needed in METAfont to work with fixed-point numbers

Uh, oh. A little while ago one of the quantities that I was computing got too large, so I'm afraid your answers will be somewhat askew. You'll probably have to adopt different tactics next time. But I shall try to carry on anyway.

Arithmetic in METAfont

METAfont restricts input numbers to \( 12 \) integer bits:

- \% \texttt{mf expr}
- \texttt{gimme an expr: 4095} \quad \texttt{>> 4095}
- \texttt{gimme an expr: 4096! Enormous number has been reduced.} \quad \texttt{>> 4095.99998}
- \texttt{gimme an expr: infinity} \quad \texttt{>> 4095.99998}
- \texttt{gimme an expr: \texttt{epsilon}} \quad \texttt{>> 0.00002}
- \texttt{gimme an expr: 1/\texttt{epsilon}} \quad \texttt{>> 0.33333}
- \texttt{gimme an expr: 3*\texttt{(1/3)}} \quad \texttt{>> 0.99998}
- \texttt{gimme an expr: 1.2 - 2.3} \quad \texttt{>> -1.1}
- \texttt{gimme an expr: 1.2 - 2.4} \quad \texttt{>> -1.2}
- \texttt{gimme an expr: 1.3 - 2.4} \quad \texttt{>> -1.09999}
Historical remarks

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

*Programming the Mark I: Early Programming Activity at the University of Manchester*

Annals of the History of Computing

2(2) 130–168 (1980)

Floating Point Arithmetic . . . The subject is not at all as trivial as most people think, and it involves a surprising amount of interesting information.

Donald E. Knuth


Computer hardware designers can make their machines much more pleasant to use, for example by providing floating-point arithmetic which satisfies simple mathematical laws.

The facilities presently available on most machines make the job of rigorous error analysis hopelessly difficult, but properly designed operations would encourage numerical analysts to provide better subroutines which have certified accuracy.

Donald E. Knuth

*Computer Programming as an Art ACM Turing Award Lecture* (1973)

Why no floating-point arithmetic?

- System dependence in *precision, range, rounding, underflow, overflow*
- Base varies: 2, 3 (Setun), 4 (Iliiac II), 8 (Burroughs), 10, 16 (IBM S/360), 256 (Iliiac III), 10000 (Maple)
- Bizarre behavior when TE\(\hat{\text{x}}\) was developed:
  - \(x \times y \neq y \times x\) (early Crays)
  - \(x \neq 1.0 \times x\) (Pr1me)
  - \(x + x \neq 2 \times x\) (Pr1me)
  - \(x \neq y\) but \(1.0/(x - y)\) gets zero-divide error
  - wrap between underflow and overflow (PDP-10)
  - job termination on overflow or zero-divide (most)
- No standardization: almost every vendor had unique floating-point system
Why no floating-point … [cont]?

Language dependence:
- Algol, Pascal, and SAIL (real)
- Fortran (REAL, DOUBLE PRECISION, and sometimes REAL*16)
- C/C++ (double, float added in 1989, long double in 1999)
- Java and C# (only float and double, but arithmetic system is badly botched: see Kahan and Darcy’s *How Java’s Floating-Point Hurts Everyone Everywhere*)

Compiler dependence: multiple precisions mapped to just one
- BSD compilers still provide no 80-bit format after 27 years in hardware

IEEE 754 binary standard (1985)
- Preliminary version first implemented in Intel 8087 chip (1980)
- Three formats defined: 32-bit, 64-bit, and 80-bit. 128-bit format available on some Alpha, IA-64, and SPARC systems.
- Nonzero normal numbers are *rational*: $x = (-1)^s f \times 2^p$, where $f \in [1, 2)$
- Signed zero
- Largest stored exponent represents Infinity when $f = 0$, quiet and signaling NaN (Not-a-Number) when $f \neq 0$
- Smallest stored exponent allows $f$ to have leading zeros with *gradual underflow* to subnormal values

IEEE 754 binary standard [cont]
- Nonstop computing model: sticky flags record exceptions
- Four rounding modes:
  - to nearest with ties to even (default)
  - to $+\infty$
  - to $-\infty$
  - to zero (historical chopping)
- $\pm\infty$ generated from large/small and finite/0
- NaN generated from 0/0, $\infty - \infty$, $\infty/\infty$, and any operation with a NaN operand
- NaN returned from functions when result is undefined in real arithmetic (e.g., $\sqrt{-1}$)
IEEE 754R Precision and range

<table>
<thead>
<tr>
<th>Binary</th>
<th>32-bit</th>
<th>24b (≈ 7D)</th>
<th>1e-45</th>
<th>1e-38</th>
<th>3e+38</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64-bit</td>
<td>53b (≈ 15D)</td>
<td>4e-324</td>
<td>2e-308</td>
<td>1e+308</td>
</tr>
<tr>
<td></td>
<td>80-bit</td>
<td>64b (≈ 19D)</td>
<td>3e-4951</td>
<td>3e-4932</td>
<td>1e+4932</td>
</tr>
<tr>
<td></td>
<td>128-bit</td>
<td>113b (≈ 34D)</td>
<td>6e-4966</td>
<td>3e-4932</td>
<td>1e+4932</td>
</tr>
<tr>
<td></td>
<td>256-bit</td>
<td>234b (≈ 70D)</td>
<td>2e-315723</td>
<td>5e-315653</td>
<td>3e+315652</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>32-bit</th>
<th>7D</th>
<th>1e-101</th>
<th>1e-95</th>
<th>1e+96</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64-bit</td>
<td>16D</td>
<td>1e-398</td>
<td>1e-383</td>
<td>1e+384</td>
</tr>
<tr>
<td></td>
<td>128-bit</td>
<td>34D</td>
<td>1e-6176</td>
<td>1e-6143</td>
<td>1e+6144</td>
</tr>
<tr>
<td></td>
<td>256-bit</td>
<td>70D</td>
<td>1e-1572932</td>
<td>1e-1572863</td>
<td>1e+15782864</td>
</tr>
</tbody>
</table>

Remark on floating-point arithmetic

Contrary to popular misconception, even in some books and compilers, floating-point arithmetic is not fuzzy.

- Results are exact if they are representable
- Multiplication by power of base is always exact, in absence of underflow and overflow
- Subtraction of numbers of like signs and exponents is exact

Binary versus decimal

- Humans less uncomfortable with decimal arithmetic
- Sales tax: 5% of 0.70 = 0.0349999... in all binary precisions, instead of exact decimal 0.035. Thus, significant cumulative rounding errors in businesses with many small transactions (food, telephone, ...)
- Financial computations need fixed-point decimal arithmetic
- Hand calculators use decimal arithmetic
- Additional decimal rounding rules (8 instead of 4)
- Decimal arithmetic eliminates most base-conversion problems

Binary versus decimal [cont]

- Older Cobol standards require 18D fixed-point
- Cobol 2002 requires 32D fixed-point and floating-point
- Proposals to add decimal arithmetic to C and C++ (2005, 2006)
- 25 years of Rexx and NetRexx scripting languages give valuable experience in arbitrary-precision decimal arithmetic
- Excellent IBM *decNumber* library provides open source decimal floating-point arithmetic with a billion ($10^9$) digits of precision and exponent magnitudes up to 999 999 999
**Binary versus decimal [cont]**

- Preliminary support in gcc for +, −, ×, and / (late 2006) based on subset of IBM decNumber library
- mathcw package provides C99-compliant run-time library for binary, and also for decimal, arithmetic (NHFB 2005–2007)
- Three sizes defined for IEEE 754R: 32-bit (7D), 64-bit (16D), and 128-bit (34D)
- IBM PowerPC chips add hardware decimal arithmetic (2007)
- Hardware support likely in future Intel IA-32 and EM64T

**Problems with IEEE 754 arithmetic**

- Language access to features slow: 27+ years and still waiting!
- Programmer unfamiliarity, ignorance, and inexperience
- Deficient educational system
- Partial implementations by some vendors (e.g., subnormals flush to zero, IA-32 has only one NaN, IA-32 and IA-64 have imperfect rounding, Java and C# lack rounding modes and higher precisions)
- Long internal registers generally beneficial, but also produce many computational surprises and double rounding, compromising portability
- Rounding behavior at underflow and overflow limits unspecified, and vendor dependent

**How decimal arithmetic is different**

- Nonzero normal numbers $x = (-1)^s f \times 2^p$, where $f$ is an *integer*: can simulate fixed-point arithmetic
- Lack of normalization means multiple storage forms, but 1., 1.0, 1.00, 1.000, … compare equal
- Quantization detectable (e.g., for financial computations, 1.00 differs from 1.000)
- Signed zero and Infinity, plus quiet and signaling NaNs detectable from first byte (binary formats require examination of all bits)
- Eight rounding modes (legal and tax mandates)
- Compact storage formats — Densely-Packed Decimal (DPD) [IBM] and Binary-Integer Decimal (BID) [Intel] — need fewer than BCD’s four bits per decimal digit

**Software floating-point arithmetic**

- \TeX{} and METAFONT must continue to guarantee identical results across platforms
- Unspecified behavior of low-level arithmetic guarantees *platform dependence*
- Floating-point not associative, so instruction ordering (e.g., compiler optimization) affects results
- Long internal registers alter precision, and results
- Multiply-add computes $x \times y + z$ with *exact* product and single rounding, getting different result from separate operations

**Conclusion:** only a *single* software floating-point arithmetic system in \TeX{} and METAFONT can guarantee *platform-independent results*
Side excursion

What if you could provide a seamlessly integrated, fully dynamic language with a conventional syntax while increasing your application’s size by less than 200K on an x86? You can do it with Lua!

Keith Fieldhouse

Software floating-point [cont.]

No need to modify \TeX beyond what has already been done: Lua\TeX interfaces \TeX to a clean and well-designed scripting language — just need to change arithmetic and library inside \texttt{lua}

Scripting languages usually offer a single floating-point datatype, typically equivalent to IEEE 754 64-bit double (that is all C used to have)

qawk and dnawk: awk for 128-bit binary and decimal

Machines are fast and memories are big: adopt 34D 128-bit format, or better, 70D 256-bit format, instead as default numeric type.

\texttt{mathcw} has highly-portable open-source library support for \textit{ten} floating-point precisions, including 256-bit binary and decimal.

How much work?

Changing scripting languages from binary to decimal floating-point arithmetic took two to four hours each, with relatively few modifications:

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Deleted</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>dgawk</td>
<td>40717</td>
<td>109</td>
<td>165</td>
</tr>
<tr>
<td>dlua</td>
<td>16882</td>
<td>25</td>
<td>94</td>
</tr>
<tr>
<td>dmawk</td>
<td>16275</td>
<td>73</td>
<td>386</td>
</tr>
<tr>
<td>dnawk</td>
<td>9478</td>
<td>182</td>
<td>296</td>
</tr>
<tr>
<td>\texttt{METAFONT} in C</td>
<td>30190</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\TeX in C</td>
<td>25215</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Floating-point arithmetic and typesetting

\TeX’s smallest dimension is \(2^{-16}\text{pt} = 1\text{sp}\), while wavelength of visible light is about 100\text{sp} (\TeXbook, p. 58): rounding errors are \textit{invisible}

\TeX’s largest dimension is \(2^{14}\text{pt} = 5.75\text{m}\), not quite billboard size

macro notation painful (\texttt{layout.tex}):

\begin{verbatim}
% MARGINNOTEYA = 0.75 * TEXTHEIGHT + FOOTSKIP
T = \TEXTHEIGHT
\multiply \T by 75 \ percent possible overflow!
\divide \T by 100
\advance \T by \FOOTSKIP
\xdef \MARGINNOTEYA {\the \T}
\end{verbatim}

reduction \(75/100 \rightarrow 1/4 \times 3\) possible here, but not in general
Floating-point arithmetic [cont]

- Overflow detection unreliable in TeX (integer arithmetic in most programming languages is worse!)
- No elementary functions available in TeX, not even square root
- METAfont offers \+, \-, \abs, \angle, \ceiling, \cosd, \dir, \floor, \length, \mexp, \mlog, \normaldeviate, \round, \sind, \sqrt, and \uniformdeviate
- Floating-point simplifies computation of fractions, scaling, and rotation: see \LaTeX{} calc package for horrors of fixed-point arithmetic
- Interface from TeX to scripting language allows conventional numeric programming

MMIX and NNIX

Whenever anybody has asked if I will be writing a book about operating systems, my reply has always been “Nix.” Therefore the name of MMIX’s operating system, NNIX, should come as no surprise.

Donald E. Knuth

**MMIXware: A RISC Computer for the Third Millenium**

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.
MMIX and NNIX [cont]

- MMIX is a modern virtual machine used in recent volumes of DEK’s famous series *The Art of Computer Programming*
- MMIX is written as literate program in published books
- Arithmetic is IEEE 754 64-bit binary in software using only 32-bit unsigned integers
- 17 floating-point instructions: FADD, FCMP, FCMPE, FDIV, FEQL, FEQLE, FINF, FIX, FIXU, FLOT, FLOTU, FMUL, FREM, FSQRT, FSUB, FUN, and FUNE
- gcc version 3.2 can be built for MMIX system
- NNIX is a (still virtual) Unix-like O/S for MMIX
- mathcw library port to MMIX: 10 new + 12 header bug workaround, out of 250,000 lines

THE EATLES
JULY/AUGUST 1969
[1969 = YEAR OF FIRST EDITION OF DONALD KNUTH’S SEMINUMERICAL ALGORITHMS (TAOCP VOLUME 2)]