

From UCB to IEEE: How Could There Possibly Be Anything More to Say About Computer Arithmetic?

In 1977 I finally graduated from UCB and, a few months later, what became the IEEE 754 standardization effort began to coalesce. IEEE 754 was my principal technical preoccupation from about 1978-1982. Its successor effort 754R has kept me even busier since 2001 - seven years and counting. In between, I implemented parts of 754 and found out what hadn't been done very well. I studied the right balance between hardware and software implementation, I studied how to test hardware and software implementations, I studied extending its principles to additional functions, I studied getting language standards interested in supporting it, and I wrote down some of what I had learned. **Today I'll outline the progress that happened and the progress that didn't happen in the last 30 years.**

Bait and Switch!

What could possibly be worse than a talk on
Floating-Point Arithmetic?

$$(x + y)(1 + \delta_1) \dots (x + y)/(1 + \delta_2) \dots$$

a source of convenience for the analyst
and confusion for the student

NOTE - This blue and gold color scheme highlights precepts taught at Berkeley.

A student has the right to
make a mistake.

When the student makes a mistake, it's up to the student to correct it.

When the professor makes a mistake, it's up to the student to correct it.

- An error bound is different from an error
-
- *And the only thing less exciting than a talk about error bounds on floating-point arithmetic is...*

IEEE 754R

Progress and Prospects

David Hough
Sun Microsystems
8 April 2003

<http://754r.ucbtest.org>

<http://camino.oakapple.net> *I was gone March-June*

What does a thesis have to do with a standard?

1968-1970-1976-1977 graduate career *Explaining and Ameliorating the Ill Condition of Zeros of Polynomials*

1977-1978-1982-1985 754-1985 *IEEE Standard for Binary Floating-Point Arithmetic*

2000-2001-2006-2008-? 754R *Standard for Floating-Point Arithmetic*

What motivates participation?

English Wisdom

For he himself has said it,
And it's greatly to his credit,
That he is an Englishman!
Though he says AN Hermitian

English wisdom about theses and standards

- There must be a beginning of any great matter, but the continuing until the end, until it be thoroughly finished, yields the true glory.

English wisdom about theses and standards

- Every day you may make progress. Every step may be fruitful. Yet there will stretch out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.

Floating Point?

- Virgule flottante?
- Gleitkomma?
- ...
- ...
- Scaled integers! *decimal preserves scale factor*
- + exception handling

IEEE 754: Situation in 1977

- Mainframe/mini diversity
 - DEC VAX – not enough exponent in double
 - IBM 360 – almost chopped hex! - killed single
 - Cray – fast hardware
- Micros to be different?

754 Binary Formats

- Signed zeros
- Subnormal numbers
- Normal numbers
- Infinity
- Quiet NaNs
- Signaling NaNs

754 Exceptions

- Inexact
- Underflow/Subnormal
- Overflow
- Division by Zero (Pole $1/0$)
- Invalid Operation/Signaling NaN Operand

754 Success Mostly

- Binary Formats – single, double, extended
- Default Rounding for common arithmetic operations
- Default Nonstop Exception Handling for 5 exception groups **but Sun Fortran recanted**

754 Problems

Dynamic exception handling

Rounding modes expensive to use

Global state inhibits optimization

No language binding – language support is just becoming available

NaNs not portable

Binary \Leftrightarrow Decimal conversion unpredictable

Expression evaluation unpredictable, especially with extended precision **and optimizing compilers**

Simple Expression Evaluation – Typical RISC or SSE2

- double x, y, z ;
- $z = x * y$;
- One arithmetic instruction, one rounding error. Max error **0.5** ulp.
- $z = x * y * z$;
- Two instructions, two rounding errors. Max error about **1.5** ulp. Chance of gratuitous intermediate over/underflow.

IA32 Extended Temporary

- `double x, y, z ;`
- `z = x * y ;`
- Four arithmetic instructions, two rounding errors.
Max error **$0.5 + \epsilon$** ulp
- `z = x * y * z ;`
- Six instructions, three rounding errors. Max error **$0.5 + 2 * \epsilon$** ulp – and no chance of gratuitous intermediate over/underflow
- Complicated expression evaluation unpredictable due to limited registers

The Problem with Expression Evaluation

- Programmers can't say what they mean or mean what they say in portable program text
 - Fast ?
 - Reproducible ?
 - Accurate ?
 - Don't care – don't ask don't tell?

754R Original Goal: Merge 3 Existing Standards

- 754 binary floating point
- 854 “radix and word-length independent” decimal floating point
- 1596.5 SCI formats and language
- *extendable*
- *reproducible*

My 2003 Goal

- Revisit areas that have proved to be problematic to see which require revision: is cost too high relative to value ?
- Move some design aspects to a higher level, closer to user programming than to system implementation

My 2007 goals

- Enhance performance of programs
- Enhance performance of programmers
- Enhance portability of program source codes
- Enhance portability of program results

2001-2007 Other participants goals – *what motivates paying for participation*

- Canonize what they've already done
- Canonize what they are planning to do that they have disclosed
- Canonize what they are planning to do but have not disclosed
- Support chip implementers
- Support ABI software (kernel, libc, libm)
- Support above-ABI libraries (LAPACK)
- Support ISV's and end-user application programmers

28 March 2008 Goal

- Submit to IEEE Microprocessor Standards Committee on April ~~1~~ 2!

754 Contentious areas that 754R has not eliminated

- Anonymous extended-precision temporaries and double rounding
- Gradual underflow and subnormal numbers
- Nonstop default exception handling
- Signed zeros

Abstract

This standard specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. This standard specifies exception conditions and their default handling.

An implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware. For operations specified in the normative part of this standard, numerical results and exceptions are **uniquely determined** by the values of the input data, sequence of operations, and destination formats, all under user control.

If only it were true.

PURPOSE

2003: This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two. For operations specified in this standard, numerical results and **exceptions** are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control.

2008: This standard provides a method for computation with floating-point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two. The results of the computation will be identical, independent of implementation, given the same input data. **Errors, and error conditions**, in the mathematical processing will be reported in a consistent manner regardless of implementation.

An exception is not an error unless it is handled badly.

SCOPE

2003: This standard specifies formats and methods for binary and decimal floating-point arithmetic in computer programming environments: standard and extended functions in 32-, 64-, and 128-bit formats and extended precision formats, and recommends formats for data interchange. Exception conditions are defined and default handling of these conditions is specified.

2008: This standard specifies formats and methods for floating-point arithmetic in computer systems: standard and extended functions with single, double, extended, and extendable precision, and recommends formats for data interchange. Exception conditions are defined and standard handling of these conditions is specified.

Important Additions

- 64- and 128-bit decimal formats in dense decimal encodings developed and donated by IBM
- 128-bit binary format (quad)
- fused multiply-add
- min/max (but much debate remains)
- correctly-rounded conversion between binary and decimal

More additions since April 2003

- optional Expression evaluation
- optional Reproducibility
- optional Alternate Exception Handling
- optional Elementary Transcendental Functions
- Static specification of attributes
- 64- and 128-bit decimal formats in different encodings developed and donated by Intel

Meta-issues

- Undefined behavior, e.g. (int) ∞
- Implementor choices, e.g. underflow
- Design for implementation on old hardware or new?
- Commutativity: signed zeros and quiet NaNs
- Names of operators

The value of a theorem is
the extent to which it
enables us to know more
while remembering less.

False theorems

- Many generalizations about exceptions (underflow)
- Many generalization about reproducible results for required functions (min/max, quiet NaNs)
- Definition of inexact exception

NO VOTES

- 2006 – not finished
- 2008 – not right
 - Arithmetic operation definition “provide”
 - Require delayed alternate exception handling
 - Reorthogonalize expression evaluation
 - Integrate reproducibility into expression evaluation

STOP

Time's up! Hand in your blue books!

My overview of the last 754R draft, ballot, and
comments:

<http://754r.ucbtest.org/msc-ballots/ballot6.html>

END

Additional slides

Quiet NaNs

- $\text{NaN1} + \text{NaN2} = ?$
- 754 says “a quiet NaN”
- $\text{NaN2} + \text{NaN1}$ might be different
- But if NaN contains or points to debug information, that information should be preserved according to *some* rule
- Implies extra comparison circuitry

Signaling NaNs

- Useful for uninitialized storage, but 754 implementations guessed wrong – all 1's is quiet not signaling.
- Intended to be a *symbolic link* to further information. Not completely supported in 754 negate/abs.
- Most operations should operate on the linked value, by a trap – but how to specify at a higher level so software might be portable?
- Not useful enough, not optional, but upward compatible

Subdividing exceptions

- Underflow and exact subnormal result – very confusing for implementors
- Signaling NaN operand vs other invalid results
- Naming individual invalid results: $0*\infty$, $0/0$, ∞/∞ , $\text{sqrt}(-1)$, ... to facilitate alternate exception handling
- Used only 5/8 of a byte

Alternate Exception Handling Facilities to support

- *Abrupt underflow*
- *Presubstitution*
- *Immediate transfer of control*
- *Delayed transfer of control (determinate)*
- *Don't slow down the normal case*

Control transfer on Exception

- for ($i = 0, i < n, i++$) {
- $z(i) = z(i) * x(i)$;
- } \rightarrow overflow: goto retry \leftarrow
- ...
- retry: ...

Presubstitution

-
- for (i = 0, i < n, i++) {
- z(i) = (sin(c * x(i)) / x(i)) ;
- } → zero-div-zero: c; ←
- ...

Exception Handling Implementations

- 754 asynchronous traps
- Numerical operands tested before each operation
- Numerical result tested after each operation
- Exception flags tested after each operation
- Exception flags tested outside loop
- Doesn't matter unless you care about performance! Let user specify behavior, let compiler specify implementation

Extended Precision

- Will anybody implement any extended precision other than IA32? If so then it is specified by IA32, not 754R
- Standard can be conveniently simplified if extended is **generalized**

Expression Evaluation

- Normal: none of accuracy, predictability, or performance are particularly important
- Performance: as fast as possible. Order of evaluation, extra precision can be sacrificed. Most dot products, FFT's.
- **Reproducibility**: Identical results on all platforms. Dynamic arithmetic parameter determination; double-double precision.
- Accuracy: as much as possible without crossing a performance boundary. Residuals.

Reproducible Expression Evaluation

- An operation specified by the standard, on operands all of the same type, produces the result specified by the standard rounded correctly to the destination precision
- Correctly-rounded transcendental functions fit here
- Completely portable and predictable results for a limited class of expressions

Static Declarations

Attach static declarations to a specific operator, expression, or block of code.

Declare expression evaluation, rounding direction, alternate exception handling.

Explicit declarations override inherited dynamic modes.

Problematic because they imply a language binding; some languages are dynamic.