IEEE 754R

Progress and Prospects

David Hough
Sun Microsystems
8 April 2003
http://754r.ucbtest.org
http://camino.oakapple.net

Floating Point?

- Virgule flottante?
- Gleitkomma?
- ...
- ...
- Scaled integers!
- **+** 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

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 ⇔ Decimal conversion unpredictable
- Expression evaluation unpredictable, especially with extended precision

Simple Expression Evaluation – Typical RISC

- 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 almost 2 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 + ε ulp
- z = x * y * z ;
- Six instructions, three rounding errors. Max error 0.5 + 2 * ε ulp – and no chance of gratuitous intermediate over/underflow
- Complicated expression evaluation unpredictable due to limited registers

Diversion: Sun Math Libraries

- Multiple (SPARC) libraries for multiple goals:
- libm - Full IEEE support – modes and exceptions – part of Solaris
- fdlibm - Portability of C source code – also Java
- libmcr - Correct rounding – C source
- libm - Almost correct rounding
- libmopt - Not quite so correct rounding
- libmvec - Vectorizability

libmvec vector library

- For loops, generally Fortran, usually invoked by compiler
- Default rounding and exception handling
- Inexact flag unpredictable
- ...
- 2-4X faster than libmopt for vectors of length 100-1000
- But 1.1-2X slower for vectors of length 1

fdlibm portable library

- C or Java source code to provide “similar” results on all platforms
- libmopt more accurate (0.65 vs 0.85 ulps)
- libmopt faster 1.3-2X
- Fdlibm compiled with GCC instead of Sun compilers, can be 1.1X faster but with greater ulp errors and more test vector failures

libm vs libmopt

- libmopt almost always faster (often 2X) and more accurate!
- libmopt shows many more exception flag errors for pow(x,y)
- libmopt less accurate and less careful about log10(10^n)

libmcr – *correctly rounded*

- research work in progress
- Gaston Gonnet tests
- 2-5X performance penalty vs libmopt; 2-3X vs fdlibm

What price correct rounding?

- Relative Cost = *increase in execution time / decrease in error bound*
- *Transcendental* functions are easier – exp(x)
- Algebraic functions are harder – pow(x,y) and especially sqrt($x^2 + y^2$) and above all x^3
- x*y+z is easier than x*x*x !

Standards for Transcendental Functions

- Require public implementations which do everything right and do not cost too much more. Vectorizable is good.
- Which functions to standardize?
- **Static declarations** of limited contexts in which accuracy or performance is more important? How much less accuracy is tolerable?

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

Current Goal

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

Contentious areas unlikely to change

- Gradual underflow and subnormal numbers
- Nonstop default exception handling
- Binary format
- Signed zeros

PURPOSE

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.

SCOPE

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.

Hardware or Software?

It is intended that 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. It is the environment the user of the system sees that conforms or fails to conform to this standard.

Important Additions

- 32-, 64-, and 128-bit decimal formats in dense 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 (in principle)
- quiet functions and predicates promoted

Important Rewrites

- Tables of comparison predicates
- Format descriptions use integers rather than fractions

Incompatible Changes

- Quiet/Signaling NaN bit defined (doesn't work for HPPA)
- Extended precision rounding control **should** affect exponents (Java on IA32)

Meta-issues

- Undefined behavior, e.g. (int) ∞
- Implementor choices, e.g. underflow
- Upward performance compatibility 754 $\Rightarrow$ 754R
- Commutativity: signed zeros and quiet NaNs
- **Static specification**
- Names of operators

Base Conversion

- Correctly-rounded in principle, but no agreed text
- Can any NaN, quiet or signaling, be converted in one direction and then in reverse without changing its bit pattern, on one system?
- Among systems?

Quiet NaNs

- NaN1 + NaN2 = ?
- 754 says “a quiet NaN”
- NaN2 + 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
- What about double converted to single?

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?

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, ∞/∞, sqrt(–1), ... to facilitate alternate exception handling

Alternate Exception Handling

- 754's hardware trap mechanism moved to appendix, but... no replacement text yet
- Facilities to support:
- Presubstitution
- Counting mode
- Break
- Longjump

Break/jump on Exception

- for (i = 0, i < n, i++) {
- z(i) = z(i) * x(i) ;
- }→overflow: goto retry←
- ...
- retry: ...

Counting Mode

- count = 0;
- for (i = 0, i < n, i++) {
- z(i) = z(i) * x(i) ;
- }→overflow, underflow: z(i), count = scaledprod(z(i),x(i),count);←
- z(i) = scalb(z(i), − count);
- …

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 removed
- Effect can be obtained through expression evaluation rules

Types of expressions

- 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.
- Predictability: Identical results on all platforms. Dynamic arithmetic parameter determination; double-double precision.
- Accuracy: as much as possible without crossing a performance boundary. Residuals.

Canonical/Standard/Normal/Boring 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 **user**'s destination precision
- Correctly-rounded transcendental functions could fit here
- Completely portable and predictable results for a limited class of expressions

Kinds of Extended Precision Expression Evaluation

- All operands are promoted to a specific higher precision; operations are performed in that precision; results converted to destination precision.   Fast and accurate on IA32.
- Widest-needed
- Fused multiply add
- What do these have in common?

An Extended Evaluation Paradigm

- Use any higher intermediate precision for anonymous temporaries as long as the error bound for each operation and the exponent range is >= that implied by the operands.
- Encompasses IA32-style.   Avoids need for specifying extended precision types.
- Reproducible results available with round-to-odd-unless-exact rounding mode.

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; this proposal is for staticly-compiled languages.

And now for something completely different?

- 754 design for intervals as pairs of points was intended to make interval arithmetic *possible to implement with point arithmetic* by providing rounding modes and infinity points
- That design is not *efficient*: information must be recomputed constantly
- Exceptions might be completely rethought: Bill.Walster@sun.com
- A format specific to interval arithmetic: Guy.Steele@sun.com

Participation

- Anybody may join email list
- Anybody may attend meetings by conference phone, starting at 22:00 $\Rightarrow$ 02:00
- Special need to hear from people using obscure aspects of standard e.g. Signaling NaNs