Fclass: a Proposed Classification of Standard Floating-Point Operands

Abstract
Occasionally some programs will have to discover and manipulate attributes of floating-point numbers and their standard formats. This document offers names for those attributes. They need support from programming languages. Normally, attributes like byte-widths of floating-point variables' *formats* are needed at compile-time; attributes such as signs of floating-point variables' *values* are needed at run-time. Both kinds of attributes are discussed here; the values' attributes are revealed by a function here called "Fclass" intended to simplify and speed up programs that must filter out peculiar floating-point operands like NaNs before performing arithmetic upon them.

Contents

Compile-time Attributes

To use EQUIVALENCE declarations in FORTRAN, or `struct` and `union` declarations in C, programmers who assemble data-structures with fields that vary at run-time must discover the byte-widths of floating-point formats first. Most formats specified by IEEE Standard 754 for Binary Floating-Point Arithmetic have standard widths known in advance regardless of platform; but the *Extended* formats' widths differ on different hardware and sometimes for different compilers on the same hardware. This is why compilers must provide a compile-time function Width(X) or Sizeof(X) that returns the width in bytes of memory occupied by floating-point variables of the same declared type as X . (Register-widths are not relevant here.) The Width(…) function returns integer values …

Name of IEEE 754 Format	Width in Bytes
Single-Precision (`float`)	4
Single-Extended	≥ 6
Double-Precision (`double`)	8
Double-Extended (`long double`)	10, 12 or 16
Quadruple-Precision (`long double`)	16
(*Doubled-double* is NOT standardized)	(16)

This tabulated list will lengthen as other floating-point formats come into use, if they ever do.

Languages must provide locutions that permit Width(…) to figure in structure declarations that are portable in the sense that a program recompiled elsewhere will still run correctly even if the data-structures it creates may not be read correctly when copied verbatim from one computer's memory to another. Of course, such programs are somewhat like suicide:– something to be discouraged but not prevented. Still, the necessary locutions, like conditional compilation, have many important applications.

Another attribute needed at compile-time is the "Little-Endian" or "Big-Endian" affiliation of a floating-point format. That is one of a few attributes, like whether the significand's leading bit is explicit (as it is for all current implementations of Double-Extended formats) or not (Single, Double and Quadruple precisions), needed for bit-twiddling of floating-point numbers. This practice too is something best discouraged but not prevented, especially not if a language fails to supply built-in run-time functions of the kind to be discussed next. Bit-twiddling is forced upon a programmer who must implement her own versions of these functions either because they are lacking in her chosen language or because its versions run too slowly. Bit-twiddling is also necessary in software that tests floating-point hardware upon specially patterned operands created using only integer hardware. The specifications of fields for sign, exponent and significand are the province of another standard, IEEE 1596.5 on Data Communication, so they need not be explored here.

Run-Time Attributes

Programs frequently filter operands before performing operations that may malfunction on peculiar values; examples include subnormal numbers and zeros that may be troublesome divisors, and infinities that spoil matrix multiplications, and signed zeros and infinities that mark the ends of open or closed intervals for Interval Arithmetic. At issue now are not the comparisons against numerical thresholds accomplished by the usual order predicates like " x < y " but the filtering out of peculiar operands like NaNs that might complicate such comparisons. When filtering is simple and fast enough it may well be preferred to the detection of malfunctions after they occur in lengthy formulas.

Speed is crucial here. It is achieved, when filtering requires more than one test per floating-point operand, by moving the tests from floating-point to integer and logical hardware, and by consolidating several tests into a few with the aid of precalculated masks whenever possible. This is possible when the tests concern attributes like the ones tabulated below … .

The nine attributes named below can be associated in our minds with nine bits each 0 or 1 according to whether the operand in question falls into the category described. These bits may well be copied from tag bits already generated when an operand is loaded into a tagged floating-point register. So long as they are generated fast, their provenance doesn't matter. Note that an operand can cause more than one bit to be set to 1 only if its sign bit or its last bit is 1 .

Proposed Name	Attribute of Floating-Point Operand
sign	Sign bit, either 0 (+) or 1 (–) .
qNaN	A “quiet” NaN ; it does not trap.
sNaN	A “signaling” NaN ; it traps when used.
infy	An infinity, either +∞ or –∞ .
finn	A finite nonzero number, not subnormal.
subn	A subnormal nonzero number.
zero	Either +0.0 or –0.0 .
oddf	The last significant bit stored is 1 .
notf	Not a (standard) floating-point number nor NaN.

(The last “*notf*” should not occur for operands loaded from memory nor for operands encountered by applications programs; it is an attribute to be encountered perhaps exclusively by operating systems software.)

The Function Fclass(x)

A function Fclass(x) is intended to return a bit-string that conveys the attributes of its floating-point operand x . In this document, Fclass(x) is treated as a *generic* function determined by the format of x as well as its value; some languages will require functions named Fclass_(x) in which the underscore “_” is replaced by a suffix like ...

Suffix	**Operand Format**
s	Single-Precision
sx	Single-Extended
d	Double-Precision
dx	Double-Extended
q	Quadruple-Precision
(dd)	(Doubled-double)

and so on. All versions of Fclass(x), generic or not, return the same bit-string for the same argument-value x regardless of format except for *finn*, *subn* and *oddf*, which depend upon x’s format.

The bit-string returned by Fclass(x) may be typed linguistically as an integer by some languages, or as a type of its own kind by better-protected languages. What matters most is that Fclass(x) return a quick classification of the value of its floating-point argument x that lends itself to logical masking in order to form predicates that test for infinities, NaNs, signed zeros, subnormal numbers, ..., and combinations thereof selected by masks of Fclass’ type built up at compile-time to speed the test demanded by the program. Of course, mask words deserve names like “signm”, “sNaNm”, “qNaNm”, … that programmers can combine by ANDing and ORing without having to memorize hexadecimal strings.

Here are examples in *Matlab*’s syntax, treating any nonzero integer as Boolean TRUE, zero as FALSE, and using & for AND , | for OR , ~ for NOT , and == for EQUALS :

```
isNaN(x)            (sNaNm | qNaNm) & Fclass(x)
isFinite(x)         ~( (infym | sNaNm | qNaNm) & Fclass(x) )   or
                       (zerom | finnm | subnm) & Fclass(x)
isInfinite(x)       infym & Fclass(x)
isPlusInfinity(x)   infym == Fclass(x)
isSubnormal(x)      subnm & Fclass(x)
isNormal(x)         ( zero | finn ) & Fclass(x)
isMinusZero(x)      Fclass(x) == (signm | zerom)
isPlusZero(x)       Fclass(x) == zerom
  x == 0.0          zerom & Fclass(x)
notZeroNorOdd(x)    ~( (zerom | oddfm) & Fclass(x) )
```

Note that these run fast as logical/integer operations because the compound masks are composed at compile-time.

SignBit

If Fclass(x) is a two-byte integer with *sign* as its leading bit, then one shift suffices to bring out the sign bit as an integer:

SignBit(x) := LogicalShift(–15, Fclass(x)) yields 0 or 1 ;

–SignBit(x) := IntegerShift(–15, Fclass(x)) yields 0 or –1 .

If the value of Fclass is not a word like that, the compiler writer should supply a fast inlined implementation of SignBit(x) because it is useful only if it is fast, and then it is useful in *Sturm Sequence* calculations in tight loops that count real roots of polynomial equations and isolate real eigenvalues of symmetric matrices.

Implementation Details

Implementors of floating-point hardware or firmware will find eight of Fclass(x)'s bits worth keeping along with perhaps more bits in a Tag field associated with each floating-point register. The Tag field's function is to speed up operations upon operands already classified as a by-product of their insertion into the registers. In conjunction with two extra exponent bits, the Tag field can also serve to speed the handling of over/underflowed intermediate results and subnormal operands without recourse to traps; that is a story for another day.

An implementation of Fclass(x) in software can resort exclusively to integer and logical operations. It must be done this way on *Pentium*-like architectures whose floating-point registers forget where their contents came from, obscuring finn, subn and oddf.

At present, the uses contemplated for Fclass preponderantly use it just once per argument, embedding it in a boolean expression like the ones illustrated above in *Matlab*'s syntax. So long as this pattern of use persists, little is lost by implementing Fclass the way the *Itanium* architecture does: it combines the sensing of Fclass(x) and the masking in one operation performed in its floating-point registers. Thus, the expressions illustrated above could be converted at compile-time into single instructions containing the appropriate mask.

However, future uses may well test Fclass(x) with different masks at different times for the same argument x . This possibility deserves exploration; would Interval and Complex Arithmetic codes benefit if values of Fclass for two arguments could be stored and combined later when needed?

Whether Fclass is implemented as a separate operation or is mixed with a mask at every invocation will not matter to the programmer with an efficiently optimizing compiler except that repeated references to Fclass(x) with the same x may be rather slower on some architectures than on others.

Appendix: Silent Order Predicates

Fclass(x) can help compilers support silent order predicates on those machines whose order predicates all trap or signal INVALID OPERATION when an operand is NaN . The six conventional comparison predicates, plus a seventh, in three common notations are ...

Math:	=	≠	?	<	≤	≥	>
C:	==	!=	?	<	<=	>=	>
Fortran:	.EQ.	.NE.	.UN.	.LT.	.LE.	.GE.	.GT.

The last four of these order predicates were specified in IEEE 754 to signal an INVALID OPERATION exception (trap or raise its flag) and deliver a .FALSE. value when NaN was compared with itself or anything else. The signal was deemed necessary to protect legacy software recompiled for hardware with NaNs though designed for old arithmetics without them. A user who noticed the signal could perhaps trace it back to an event that had previously gone unnoticed or did not occur on older hardware. Protection was intentionally imperfect because the two predicates "NaN .EQ. NaN" (specified .FALSE.) and "NaN .NE. NaN" (specified .TRUE.) did not signal; they served to detect a NaN in a language that lacked the word "NaN". Unfortunately they are "optimized" away too often by compilers that replace " x .NE. x " by " .FALSE. " at compile time; this is why " isNaN(x) " should be used instead.

(The UNORDERED predicate " x ? y " or " x .UN. y " above is silently .TRUE. just when at least one of of x and y is NaN . Strictly speaking, this predicate is mathematically superfluous since there is no need for NaNs in mathematical proofs, which take *trichotomy* for granted. Only computers, unable to stop and revise their own programs in the light of unforseen circumstances, need NaNs and an UNORDERED predicate.)

Nowadays programmers aware of NaNs' peculiarities need silent order predicates that never signal. In the syntax I prefer for C , these should augment the predicates listed above thus:

Math:	=	≠	?	!?	<>	<	≤	≥	>
C with signal:					<>	<	<=	>=	>
C silent:	==	!=	?	!?	!=?	!>=	!>	!<	!<=

No comparison predicate whose symbol begins with "!" would ever signal.

(I have yet to need more than the fourteen predicates tabulated here.)

Note that silent " x !> y " differs from signaling " !(x > y) " when either x or y can be NaN . However, " x != y " matches " !(x == y) " and similarly for !? since none of these ever signal.

Whatever their syntax, all the silent predicates should run fast too. This is why a fast Fclass should be used to prevent traps by filtering operands before the hardware's signaling predicate is applied to get the desired silent result.

Appendix: What is *oddf* good for?

It may speed up some implementations of multi-double arithmetic. Or it may be superfluous. The subject deserves further investigation. Like the INEXACT exception, it is an idea that most current practitioners of floating-point arithmetic have never considered though, in a past now so distant that almost nobody alive remembers it, such things used to be part of the trickery in which skilled practitioners exulted. For instance, if ~(oddfm & Fclass(x)) then 3.0*x suffers no roundoff, which fact figures in a program that solves cubic equations.