Floating Point Numbers in Java[1]

by Michael L. Overton

Virtually all modern computers follow the *IEEE*[2] *floating point standard* in their representation of floating point numbers. The Java programming language types *float* and *double* use the *IEEE single format* and the *IEEE double format* respectively.

Floating Point Representation

Floating point representation is based on *exponential* (or *scientific*) notation). In exponential notation, a nonzero real number x is expressed in decimal as

$$x = \pm S \times 10^E, \quad \text{where } 1 \leq S < 10,$$

and E is an integer. The numbers S and E are called the *significand* and the *exponent* respectively. For example, the exponential representation of 365.25 is 3.6525×10^2, and the exponential representation of 0.00036525 is as 3.6525×10^{-4}. It is always possible to satisfy the requirement that $1 \leq S < 10$, as S can be obtained from x by repeatedly multiplying or dividing by 10, decrementing or incrementing the exponent E accordingly. We can imagine that the decimal *point floats* to the position immediately after the first nonzero digit in the decimal expansion of the number: hence the name floating point. For representation on the computer, we prefer base 2 to base 10, so we write a nonzero number x in the form

$$x = \pm S \times 2^E, \quad \text{where } 1 \leq S < 2. \tag{1}$$

Consequently, the binary expansion of the significand is

$$S = (b_0.b_1b_2b_3\ldots)_2, \quad \text{with } b_0 = 1. \tag{2}$$

For example, the number 11/2 is expressed as

$$\frac{11}{2} = (1.011)_2 \times 2^2.$$

Now it is the *binary point* that *floats* to the position after the first nonzero bit in the binary expansion of x, changing the exponent E accordingly. Of course, this is not possible if the number x is zero, but at present we are considering only the nonzero case. Since b_0 is 1, we may write

$$S = (1.b_1b_2b_3\ldots)_2.$$

[1]Extracted from *Numerical Computing with IEEE Floating Point Arithmetic*, published by the Society for Industrial and Applied Mathematics (SIAM), March 2001. Copyright ©SIAM 2001.

[2]Institute for Electrical and Electronics Engineers. IEEE is pronounced "I triple E". The standard was published in 1985.

Table 1: IEEE Single Format

$\pm$	$a_1a_2a_3\ldots a_8$	$b_1b_2b_3\ldots b_{23}$

If exponent bitstring $a_1\ldots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{23} = 0$, NaN otherwise

The bits following the binary point are called the *fractional* part of the significand.

A more complicated example is the number 1/10, which has the nonterminating binary expansion

$$\frac{1}{10} = (0.0001100110011\ldots)_2 = \frac{1}{16}+\frac{1}{32}+\frac{0}{64}+\frac{0}{128}+\frac{1}{256}+\frac{1}{512}+\frac{0}{1024}+\cdots. \tag{3}$$

We can write this as

$$\frac{1}{10} = (1.100110011\ldots)_2 \times 2^{-4}.$$

Again, the binary point *floats* to the position after the first nonzero bit, adjusting the exponent accordingly. A binary number that has its binary point in the position after the first nonzero bit is called *normalized.*

Floating point representation works by dividing the computer word into three fields, to represent the sign, the exponent and the significand (actually, the fractional part of the significand) separately.

The Single Format

IEEE single format floating point numbers use a 32-bit word and their representations are summarized in Table 1. The first bit in the word is the sign bit, the next 8 bits are the exponent field, and the last 23 bits are the fraction field (for the fractional part of the significand).

Let us discuss Table 1 in some detail. The $\pm$ refers to the sign of the number, a zero bit being used to represent a positive sign. The first line shows that the representation for zero requires a special zero bitstring for the exponent field *as well as* a zero bitstring for the fraction field, i.e.,

$$\begin{array}{|c|c|c|}\hline \pm & 00000000 & 00000000000000000000000 \\ \hline \end{array}.$$

No other line in the table can be used to represent the number zero, for all lines except the first and the last represent normalized numbers, with an initial bit equal to one; this bit is said to be *hidden*, since it is not stored explicitly. In the case of the first line of the table, the hidden bit is zero, not one. The 2^{-126} in the first line is confusing at first sight, but let us ignore that for the moment since $(0.000\ldots0)_2 \times 2^{-126}$ is certainly one way to write the number zero. In the case when the exponent field has a zero bitstring but the fraction field has a nonzero bitstring, the number represented is said to be *subnormal*. Let us postpone the discussion of subnormal numbers for the moment and go on to the other lines of the table.

All the lines of Table 1 except the first and the last refer to the normalized numbers, i.e., all the floating point numbers that are not special in some way. Note especially the relationship between the exponent bitstring $a_1a_2a_3\ldots a_8$ and the actual exponent E. This is *biased representation*: the bitstring that is stored is the binary representation of $E + 127$. The number 127, which is added to the desired exponent E, is called the *exponent bias*. For example, the number $1 = (1.000\ldots0)_2 \times 2^0$ is stored as

$$\begin{array}{|c|c|c|}\hline 0 & 01111111 & 00000000000000000000000 \\ \hline \end{array}.$$

Here the exponent bitstring is the binary representation for $0 + 127$ and the fraction bitstring is the binary representation for 0 (the fractional part of 1.0). The number $11/2 = (1.011)_2 \times 2^2$ is stored as

$$\begin{array}{|c|c|c|}\hline 0 & 10000001 & 01100000000000000000000 \\ \hline \end{array}.$$

The number $1/10 = (1.100110011\ldots)_2 \times 2^{-4}$ has a nonterminating binary expansion. If we truncated this to fit the fraction field size, we would find that $1/10$ is stored as

$$\begin{array}{|c|c|c|}\hline 0 & 01111011 & 10011001100110011001100 \\ \hline \end{array}.$$

However, it is better to *round*[3] the result, so that $1/10$ is represented as

$$\begin{array}{|c|c|c|}\hline 0 & 01111011 & 10011001100110011001101 \\ \hline \end{array}.$$

[3]The IEEE standard offers several rounding options, but the Java language permits only one: rounding to nearest.

The range of exponent field bitstrings for normalized numbers is 00000001 to 11111110 (the decimal numbers 1 through 254), representing actual exponents from $E_{\min} = -126$ to $E_{\max} = 127$. The smallest positive normalized number that can be stored is represented by

$$\begin{array}{|c|c|c|}\hline 0 & 00000001 & 00000000000000000000000 \\ \hline\end{array}$$

and we denote this by

$$N_{\min} = (1.000\ldots 0)_2 \times 2^{-126} = 2^{-126} \approx 1.2 \times 10^{-38}. \tag{4}$$

The largest normalized number (equivalently, the largest finite number) is represented by

$$\begin{array}{|c|c|c|}\hline 0 & 11111110 & 11111111111111111111111 \\ \hline\end{array}$$

and we denote this by

$$N_{\max} = (1.111\ldots 1)_2 \times 2^{127} = (2 - 2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}. \tag{5}$$

The last line of Table 1 shows that an exponent bitstring consisting of all ones is a special pattern used to represent $\pm\infty$ or NaN, depending on the fraction bitstring. We will discuss these later.

Subnormals

Finally, let us return to the first line of the table. The idea here is as follows: although 2^{-126} is the smallest normalized number that can be represented, we can use the combination of the special zero exponent bitstring and a nonzero fraction bitstring to represent smaller numbers called subnormal numbers. For example, 2^{-127}, which is the same as $(0.1)_2 \times 2^{-126}$, is represented as

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 10000000000000000000000 \\ \hline\end{array},$$

while $2^{-149} = (0.0000\ldots 01)_2 \times 2^{-126}$ (with 22 zero bits after the binary point) is stored as

$$\begin{array}{|c|c|c|}\hline 0 & 00000000 & 00000000000000000000001 \\ \hline\end{array}.$$

This is the smallest positive number that can be stored. Now we see the reason for the 2^{-126} in the first line. It allows us to represent numbers in the range immediately below the smallest positive normalized number. Subnormal numbers cannot be normalized, since normalization would result in an exponent that does not fit in the field. Subnormal numbers are *less accurate*, i.e., they have less room for nonzero bits in the fraction field, than normalized numbers. Indeed, the accuracy drops as the size of the

Table 2: IEEE Double Format

$\pm$	$a_1a_2a_3\ldots a_{11}$	$b_1b_2b_3\ldots b_{52}$

If exponent bitstring is $a_1\ldots a_{11}$	Then numerical value represented is
$(00000000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1021}$
$(00000000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1020}$
$\downarrow$	$\downarrow$
$(01111111111)_2 = (1023)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{0}$
$(10000000000)_2 = (1024)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111111100)_2 = (2044)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1021}$
$(11111111101)_2 = (2045)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1022}$
$(11111111110)_2 = (2046)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1023}$
$(11111111111)_2 = (2047)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{52} = 0$, NaN otherwise

subnormal number decreases. Thus $(1/10) \times 2^{-123} = (0.11001100\ldots)_2 \times 2^{-126}$ is truncated to

0	00000000	11001100110011001100110

,

while $(1/10) \times 2^{-135} = (0.11001100\ldots)_2 \times 2^{-138}$ is truncated to

0	00000000	00000000000011001100110

.

Exercise 1 *Determine the IEEE single format floating point representation for the following numbers: 2, 1000, 23/4, $(23/4) \times 2^{100}$, $(23/4) \times 2^{-100}$, $(23/4) \times 2^{-135}$, $(1/10) \times 2^{10}$, $(1/10) \times 2^{-140}$. (Make use of (3) to avoid decimal to binary conversions).*

Exercise 2 *What is the gap between 2 and the first IEEE single number larger than 2? What is the gap between 1024 and the first IEEE single number larger than 1024?*

The Double Format

The single format is not adequate for many applications, either because more accurate significands are required, or (less often) because a greater exponent range is needed. The IEEE standard specifies a second basic format, *double*, which uses a 64-bit double word. Details are shown in Table 2. The ideas are the same as before; only the field widths and exponent bias are

Table 3: Range of IEEE Floating Point Formats

Format	$E_{\min}$	$E_{\max}$	$N_{\min}$	$N_{\max}$
Single	-126	127	$2^{-126} \approx 1.2 \times 10^{-38}$	$\approx 2^{128} \approx 3.4 \times 10^{38}$
Double	-1022	1023	$2^{-1022} \approx 2.2 \times 10^{-308}$	$\approx 2^{1024} \approx 1.8 \times 10^{308}$

different. Now the exponents range from $E_{\min} = -1022$ to $E_{\max} = 1023$, and the number of bits in the fraction field is 52. Numbers with no finite binary expansion, such as $1/10$ or π, are represented more accurately with the double format than they are with the single format. The smallest positive normalized double number is

$$N_{\min} = 2^{-1022} \approx 2.2 \times 10^{-308} \tag{6}$$

and the largest is

$$N_{\max} = (2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}. \tag{7}$$

We summarize the bounds on the exponents, and the values of the smallest and largest normalized numbers given in (4), (5), (6), (7), in Table 3.

Significant Digits

Let us define p, the precision of the floating point format, to be the number of bits allowed in the significand, including the hidden bit. Thus $p = 24$ for the single format and $p = 53$ for the double format. The $p = 24$ bits in the significand for the single format correspond to *approximately* 7 *significant decimal digits*, since

$$2^{-24} \approx 10^{-7}.$$

Here $\approx$ means *approximately equals*[4]. Equivalently,

$$\log_{10}\left(2^{24}\right) \approx 7. \tag{8}$$

The number of bits in the significand of the double format, $p = 53$, corresponds to *approximately* 16 *significant decimal digits.* We deliberately use the word *approximately* here, because defining *significant digits* is problematic. The IEEE single representation for

$$\pi = 3.141592653\ldots,$$

is, when converted to decimal,

$$3.141592741\ldots.$$

[4]In this case, they differ by about a factor of 2, since 2^{-23} is even closer to 10^{-7}.

To how many digits does this approximate π? We might say 7, since the first 7 digits of both numbers are the same, or we might say 8, since if we round both numbers to 8 digits, rounding π up and the approximation down, we get the same number 3.1415927.

Representation Summary

The IEEE single and double format numbers are those that can be represented as

$$\pm(b_0.b_1b_2\ldots b_{p-1})_2 \times 2^E,$$

with, for normalized numbers, $b_0 = 1$ and $E_{\min} \le E \le E_{\max}$, and, for subnormal numbers and zero, $b_0 = 0$ and $E = E_{\min}$. We denoted the largest normalized number by $N_{\max}$, and the smallest positive normalized number by $N_{\min}$. There are also two infinite floating point numbers, $\pm\infty$.

Correctly Rounded Floating Point Operations

A key feature of the IEEE standard is that it requires correctly rounded arithmetic operations. Very often, the result of an arithmetic operation on two floating point numbers is *not* a floating point number. This is most obviously the case for multiplication and division; for example, 1 and 10 are both floating point numbers but we have already seen that 1/10 is not, regardless of where the single or double format is in use. It is also true of addition and subtraction: for example, 1 and 2^{-24} are IEEE single format numbers, but $1 + 2^{-24}$ is not.

Let x and y be floating point numbers, let $+$,$-$,$\times$,$/$ denote the four standard arithmetic operations, and let $\oplus$,$\ominus$,$\otimes$,$\oslash$ denote the corresponding operations as they are actually implemented on the computer. Thus, $x+y$ may not be a floating point number, but $x \oplus y$ is the floating point number which is the computed approximation of $x+y$. When the result of a floating point operation is not a floating point number, the IEEE standard requires that the computed result is the rounded value of the exact result. It is worth stating this requirement carefully. The rule is as follows: if x and y are floating point numbers, then

$$x \oplus y = \text{round}(x+y),$$

$$x \ominus y = \text{round}(x-y),$$

$$x \otimes y = \text{round}(x \times y),$$

and

$$x \oslash y = \text{round}(x/y),$$

where round is the operation of *rounding* to the nearest floating point number in the single or double format, whichever is in use. This means that the result of an operation with single format floating point numbers is accurate to 24 bits (about 7 decimal digits), while the result of an operation with double format numbers is accurate to 53 bits (about 16 decimal digits).

The Intel Pentium chip received a lot of bad publicity in 1994 when the fact that it had a floating point hardware bug was exposed. For example, on the original Pentium, the floating point division operation

$$\frac{4195835}{3145727}$$

gave a result with only about 4 correct decimal digits. The error occurred only in a few special cases, and could easily have remained undiscovered much longer than it did; it was found by a mathematician doing experiments in number theory. Nonetheless, it created a sensation, mainly because it turned out that Intel knew about the bug but had not released the information. The public outcry against incorrect floating point arithmetic depressed Intel's stock value significantly until the company finally agreed to replace everyone's defective processors, not just those belonging to institutions that Intel thought really needed correct arithmetic! It is hard to imagine a more effective way to persuade the public that floating point accuracy is important than to inform it that only specialists can have it. The event was particularly ironic since no company had done more than Intel to make accurate floating point available to the masses.

Exceptions

One of the most difficult things about programming is the need to anticipate exceptional situations. Ideally, a program should handle exceptional data in a manner as consistent as possible with the handling of unexceptional data. For example, a program that reads integers from an input file and echoes them to an output file until the end of the input file is reached should not fail just because the input file is empty. On the other hand, if it is further required to compute the average value of the input data, no reasonable solution is available if the input file is empty. So it is with floating point arithmetic. When a reasonable response to exceptional data is possible, it should be used.

Infinity from Division by Zero

The simplest example of an exception is *division by zero*. Before the IEEE standard was devised, there were two standard responses to division of a positive number by zero. One often used in the 1950's was to generate the largest floating point number as the result. The rationale offered by the manufacturers was that the user would notice the large number in the output and draw the conclusion that something had gone wrong. However, this often led to confusion: for example, the expression $1/0 - 1/0$ would give the result 0, which is meaningless; furthermore, as 0 is not large, the user might *not* notice that any error had taken place. Consequently, it was emphasized in the 1960's that division by zero should lead to the interruption or termination of the program, perhaps giving the user an informative message such as "fatal error — division by zero". To avoid this, the burden was on the programmer to make sure that division by zero would never occur.

Suppose, for example, it is desired to compute the total resistance of an electrical circuit with two resistors connected in parallel. The formula for the total resistance of the circuit is

$$T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}. \tag{9}$$

This formula makes intuitive sense: if both resistances R_1 and R_2 are the same value R, then the resistance of the whole circuit is $T = R/2$, since the current divides equally, with equal amounts flowing through each resistor. On the other hand, if R_1 is very much smaller than R_2, the resistance of the whole circuit is somewhat less than R_1, since most of the current flows through the first resistor and avoids the second one. What if R_1 is zero? The answer is intuitively clear: since the first resistor offers no resistance to the current, *all* the current flows through that resistor and avoids the second one; therefore, the total resistance in the circuit is zero. The formula for T also makes sense mathematically, if we introduce the convention that $1/0 = \infty$ and $1/\infty = 0$. We get

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

Why, then, should a programmer writing code for the evaluation of parallel resistance formulas have to worry about treating division by zero as an exceptional situation? In IEEE arithmetic, the programmer is relieved of that burden. The standard response to division by zero is to produce an infinite result, and continue with program execution. In the case of the parallel resistance formula, this leads to the correct final result $1/\infty = 0$.

NaN from Invalid Operation

It is true that $a \times 0$ has the value 0 for any *finite* value of a. Similarly, we adopt the convention that $a/0 = \infty$ for any *positive* value of a. Multiplication with ∞ also makes sense: $a \times \infty$ has the value ∞ for any *positive* value of a. But the expressions $0 \times \infty$ and $0/0$ make no mathematical sense. An attempt to compute either of these quantities is called an *invalid operation*, and the IEEE standard response to such an operation is to set the result to NaN (Not a Number). Any subsequent arithmetic computation with an expression that involves a NaN also results in a NaN. When a NaN is discovered in the output of a program, the programmer knows something has gone wrong and can invoke debugging tools to determine what the problem is.

Addition with ∞ makes mathematical sense. In the parallel resistance example, we see that $\infty + \frac{1}{R_2} = \infty$. This is true even if R_2 also happens to be zero, because $\infty + \infty = \infty$. We also have $a - \infty = -\infty$ for any *finite* value of a. But there is no way to make sense of the expression $\infty - \infty$, which therefore yields the result NaN.

Exercise 3 *What are the values of the expressions $\infty/0$, $0/\infty$ and ∞/∞? Justify your answer.*

Exercise 4 *For what nonnegative values of a is it true that a/∞ equals 0?*

Exercise 5 *Using the 1950's convention for treatment of division by zero mentioned above, the expression $(1/0)/10000000$ results in a number very much smaller than the largest floating point number. What is the result in IEEE arithmetic?*

Signed Zeros and Signed Infinities

A question arises: why should $1/0$ have the value ∞ rather than $-\infty$? This is one motivation for the existence of the floating point number -0, so that the conventions $a/0 = \infty$ and $a/(-0) = -\infty$ may be followed, where a is a positive number. The reverse holds if a is negative. The predicate $(0 = -0)$ is true, but the predicate $(\infty = -\infty)$ is false. We are led to the conclusion that it is possible that the predicates $(a = b)$ and $(1/a = 1/b)$ have opposite values (the first true, the second false, if $a = 0$, $b = -0$). This phenomenon is a direct consequence of the convention for handling infinity.

Exercise 6 *Are there any other cases in which the predicates $(a = b)$ and $(1/a = 1/b)$ have opposite values, besides a and b being zeros of opposite sign?*

More about NaN's

The square root operation provides a good example of the use of NaN's. Before the IEEE standard, an attempt to take the square root of a negative number might result only in the printing of an error message and a positive result being returned. The user might not notice that anything had gone wrong. Under the rules of the IEEE standard, the square root operation is invalid if its argument is negative, and the standard response is to return a NaN.

More generally, NaN's provide a very convenient way for a programmer to handle the possibility of invalid data or other errors in many contexts. Suppose we wish to write a program to compute a function which is not defined for some input values. By setting the output of the function to NaN if the input is invalid or some other error takes place during the computation of the function, the need to return special error messages or codes is avoided. Another good use of NaN's is for initializing variables that are not otherwise assigned initial values when they are declared.

When a and b are real numbers, one of three relational conditions holds: $a = b$, $a < b$ or $a > b$. The same is true if a and b are floating point numbers in the conventional sense, even if the values $\pm\infty$ are permitted. However, if either a or b is a NaN none of the three conditions $a = b$, $a < b$, $a > b$ can be said to hold (even if both a and b are NaN's). Instead, a and b are said to be *unordered.* Consequently, although the predicates $(a \leq b)$ and $(\text{not}(a > b))$ usually have the same value, they have *different* values (the first false, the second true) if either a or b is a NaN.

The appearance of a NaN in the output of a program is a sure sign that something has gone wrong. The appearance of ∞ in the output may or may not indicate a programming error, depending on the context. When writing programs where division by zero is a possibility, the programmer should be cautious. Operations with ∞ should not be used unless a careful analysis has ensured that they are appropriate.

Casting in Java

In Java, a cast is required to convert an expression of one type to a variable of another type if substantial information loss is possible. Thus, for example, a cast is required to assign a *float* expression to an *int* variable or a *double* expression to an *int* or *float* variable. The question of whether a cast is required to convert an *int* expression to a *float* variable is more complicated. Not all *int*'s can be represented as *float*'s; for example, the integer 12345678 or the integer 22222222, which both can be represented exactly with the 32-bit integer format, have too many decimal digits to be represented exactly with the 23-bit significand used by *float.* Nonetheless, Java allows the assignment to be made without a cast, as the information loss is not considered sufficiently great to insist on a cast.

Java on a PC

Java was developed by Sun, whose machines all use 64-bit floating point registers. However, all PC's (Pentium's and earlier models) have 80-bit floating point registers. The IEEE floating point standard includes requirements for how to convert between the *double* 64-bit format and the extended 80-bit format, and the C language allows the use of *long double* variables which can exploit the accuracy of the 80-bit registers. However, the Java language does not allow use of the 80-bit format, because of its requirement that the same result be obtained on all machines. Consequently, it requires the use of a special floating point option on PC's that causes the results of all operations in the 80-bit floating point registers to be computed only to the accuracy used by the 64-bit double format, i.e., as if the significand were only 52 bits.