Writing robust IEEE recommended functions in “100% Pure Java”™

Joseph D. Darcy[†]

1. Abstract

In addition to specifying details of floating point formats and arithmetic operations, the IEEE 754 and IEEE 854 standards recommend conforming systems include several utility functions to aid in writing portable programs. For example, floating point constants associated with a format, such as the largest representable number, can be generated in a format-independent manner using the recommended functions and a handful of floating point literals. The functions perform useful tasks such as returning the floating point number adjacent to a given value and scaling a floating point number by a power of two. However, the descriptions of the functions are very brief and the two related standards give slightly different specifications for some of the functions.

This paper describes robust implementations of the IEEE recommended functions written in “100% Pure Java.” These functions work under all the dynamic rounding and trapping modes supported by IEEE 754. Writing such functions in Java presents a challenge since Java does not include methods to set or query the IEEE 754 floating point state. Most previous software implementations of the IEEE recommended functions, such as analogous functions included in the C math library, are intended to be run under only the IEEE default modes, round to nearest with trapping disabled.

2. Introduction

The IEEE 754 [1] and IEEE 854 [3] standards recommend conforming systems included ten utility functions to aid program portability. (The IEEE 754 standard is for binary floating point numbers and specifies bit-level layout while IEEE 854 covers both binary and decimal floating point numbers but does not specify encodings for floating point values.) In both standards the recommended functions perform a variety of useful tasks such as scaling a floating point number, returning adjacent floating point values, and extracting a floating point number’s exponent. Before discussing options for implementing these functions, relevant features of the floating point standards are summarized. Unless otherwise indicated, in the remainder of this document “the standard” refers to IEEE 754, whose details are used instead of IEEE 854.

2.1. IEEE 754/854 Floating Point Arithmetic

Certain capabilities in the standard are required while others are optional but recommended. For our implementation of the recommended functions, we assume widely implemented recommended features (specifically trapping mode) are both available and in use. However, our programs cannot directly use these features since they are not supported in Java. The standard’s arithmetic operations include base conversion, comparison, addition, subtraction, multiplication, division, remainder, and square root. IEEE 754 defines three relevant floating point formats of differing sizes. The 32 bit single format is mandatory and the optional 64 bit double format is ubiquitous on conforming processors. Some architectures, such as the x86, include support for a third optional, but recommended, 80 bit double extended format. (Two other formats, the single extended format used on some DSP chips and the 128 bit quad format, will not be discussed.) Wider formats have both a larger range and more precision than narrower ones.

2.1.1. IEEE 754 Floating Point Values

The finite values representable by an IEEE number can be categorized by an equation with two integer parameters k and n, along with two format-dependent constants N and K [15]:

$$\text{finite value} = n \cdot 2^{k+1-N}.$$

The parameter k is the unbiased exponent with K+1 bits. The value of k ranges between $E_{min} = -(2^K-2)$and $E_{max} = 2^K-1$. The parameter n is the N-bit *significand*, similar to the mantissa of other floating point standards. To

[†] This material is based upon work supported under a National Science Foundation Graduate Fellowship.

Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

avoid multiple representations for the same value, where possible the exponent k is minimized to allow n to have a leading 1. When this "normalization" process is not possible a *subnormal* number results. Subnormal numbers have the smallest possible exponent. Unlike other floating point designs, IEEE 754 uses subnormal numbers to fill in the gap otherwise left between zero and the smallest normalized number. Including subnormal numbers permits *gradual underflow*.

Floating point values are encoded in memory using a three field layout. From most significant to least significant bits, the fields are sign, exponent, and significand:

$$\text{finite value} = (-1)^{sign} \cdot 2^{exponent} \cdot significand\,.$$

For a normalized number, the leading bit of the significand is always one. The single and double formats do not actually store this leading *implicit bit*. Subnormal values are encoding with a special out-of-range exponent value, $E_{min} - 1$.

Besides the real numbers discussed above, IEEE 754 includes *special values* NaN (Not a Number) and $\pm\infty$. The special values are encoded using the out-of-range exponent $E_{max} + 1$. The values ± 0.0 are distinguishable although they compare as equal. Together, the normal numbers, subnormals, and zeros are referred to as finite numbers. Except for NaNs, if the bits of an IEEE floating point numbers are interpreted as signed-magnitude integers, the integers have the same lexicographical order as their floating point counterparts.

NaN is used in place of various invalid results such as 0/0.[1] Actually, many different bit patterns encode a NaN value; the intention is to allow extra information, such as the address of the invalid operation generating a NaN, to be stored into the significand field of the NaN. These different NaN values can be distinguished only through non-arithmetic means. For assignment and IEEE arithmetic operations, if a NaN is given as an operand the same NaN must be returned as the result; if a binary operation is given two NaN operands one of them must be returned.

The standard defines the comparison relation between floating point numbers. Besides the usual, >, =, and <, the inclusion of NaN introduces an *unordered* relation between numbers. A NaN is neither less than, greater than, nor equal to any floating point value (even itself). Therefore, `NaN > a`, `NaN == a`, and `NaN < a` are all false. A NaN is unordered compared to any floating point value.

By default, an IEEE arithmetic operation behaves as if it first computed a result exactly and then rounded the result to the floating point number closest to the exact result. (In the case of a tie, the number with the last significand bit zero is returned.) While rounding to nearest is usually the desired rounding policy, certain algorithms, such as interval arithmetic, require other rounding conventions. To support these algorithms, the IEEE standard has three additional rounding modes, round to zero, round to $+\infty$, and round to $-\infty$. The rounding mode can be set and queried dynamically at runtime.

The implementation of the recommended functions occasionally slightly perturb a floating point value. This perturbation is measured in ulps. An ulp (unit in the last place) of a real number is the difference between that number and its nearest representable floating point value. More generally, the error in a calculated result is often measured in ulps; a finite error in a result is "correctly rounded" if its error cannot exceed half an ulp.

2.1.2. IEEE 754 Exceptional Conditions

When evaluating IEEE 754 floating point expressions, various exceptional conditions can arise. The conditions indicate events have occurred which may warrant further attention. The five exceptional events are, in decreasing order of severity,

1. **invalid**, NaN created from non-NaN operands; for example 0/0 and $\sqrt{-1.0}$.
2. **overflow**, result too large to represent; depending on the rounding mode and the operands, infinity or the most positive or most negative number is returned. Inexact is also signaled in this case.
3. **divide by zero**, non-zero dividend with a zero divisor yields a signed infinity exactly.
4. **underflow**, a subnormal value has been created.
5. **inexact**, the result is not exactly representable; some rounding has occurred (actually a very common event).

The standard has two mechanisms for dealing with these exceptional conditions: sticky flags and traps. Flags are mandatory. Although optional, trapping mode is widely implemented on processors conforming to IEEE 754. The mechanism used for each exceptional condition can be set independently. When sticky flags are used,

[1] The standard actually defines two kinds of NaNs, quiet NaNs and signaling NaNs. Since they cannot be generated as the result of an arithmetic operation, this document ignores signaling NaNs and assumes all NaNs are quiet.

arithmetic operations have the side effect of ORing the conditions raised by that operation into a global status flag. The sticky flag status can also be cleared and set explicitly. When a condition's trap is enabled, the occurrence of that condition causes a hardware trap to occur and a trap handler to be invoked. The standard requests that the trap handler be able to behave like a subroutine, computing and returning a value as if the instruction had executed normally. Even though not explicitly required in the standard, our implementation assumes that trapping status, like rounding mode, can be dynamically set at runtime. In this document, signaling a condition refers to either setting the corresponding flag or generating the appropriate trap, depending on the trapping status.

Trapping mode requires some additional information beyond notification of an exceptional event. For trapping on overflow and underflow, the trap handler is to be able to return an exponent-adjusted result, a floating point number with the same significand as if the exponent range were unbounded but with an exponent adjusted up or down by a known amount so the number is representable. The trapping model used in this paper maps traps to Java exceptions so it is not possible to return to the location which caused the exception. (A method of the exception provides the exponent-adjusted result for overflow or underflow.)

Underflow is signaled differently depending on the trapping status. If trapping on underflow is enabled, any attempt to create a subnormal number will cause a trap. In non-trapping mode, only a subnormal result that is also inexact will raise the underflow flag.

2.2. IEEE Recommended functions in Java

One of the main design criteria in Sun's Java programming language is "write once, run anywhere" portability [9]. For arithmetic, Java's portability comes from rigorously defining the sizes of and operations on integer and floating point numbers. Unlike previous languages that did not require a specific floating point standard, Java's `float` and `double` types are mandated to be IEEE 754 single and double precision numbers. Unfortunately, Java either omits or explicitly disallows required features of IEEE 754, including rounding modes and sticky flags. Several language proposals (such as RealJava [5] and Borneo [6]) add more complete IEEE 754 support to Java. Part of a complete IEEE 754 environment is a full set of the IEEE recommended functions. The Java core API already includes two simple recommended functions (`isNaN` and `isInfinite`[2]); this paper describes how to implement robust versions of the remaining functions in "100% Pure Java" [14], Java code that uses methods in the core API and refrains from calling any `native` methods. Our primary design goal is robustness, not speed. While our implementation is not gratuitously slow, handling exceptional conditions properly can be costly, especially since some processors operate much more slowly on special floating point values than on normal numbers.

Writing IEEE recommended functions that stay within Java's limited floating point model would not be very difficult. For example, since Java does not provide access to the sticky flags and disallows floating point exceptions, the signaling behavior of a method would not have to conform to the standard's constraints. However since these functions are intended to be used in a full IEEE 754 environment, our implementation returns the correct value and generates the proper signal(s) for every given input. Moreover, our implementation works under all rounding modes and under any trapping status even though Java does not provide mechanisms to set or query these pieces of floating point state. Functioning under all dynamic IEEE modes without the ability to directly control or sense that state significantly complicates the implementation. For example, the `nextAfter` method must perform extra floating point operations to work under all rounding modes; all methods must avoid premature computation on subnormals in case trapping on underflow is enabled.

Creating a Java implementation of the recommended functions that satisfies the above conditions is somewhat perverse; the implementation must worry about details neglected by Java. Our implementation is intended to be used in a Java-derived environment like Borneo where IEEE 754 semantics are respected. Until such an environment exists, a pure Java substitute must be used. As discussed in the remainder of the paper, the complexity of respecting IEEE 754 details even in the relatively simple recommended functions argues for improved language support to make writing such codes more tractable. Simply adding library routines that sense and change IEEE 754 floating point state does *not* provide sufficient support since compiler optimizations can unpredictably invalidate reliance on delicate floating point semantics.

To write code that works under all dynamic modes, one approach (promoted by the `ProcEntry`/`ProcExit` routines in SANE [2]) is to save the floating point state at the start of a method, install a known default state (such as round to nearest and non-trapping mode), perform the calculation to signal any new

[2] The standard calls for `isFinite` instead of Java's `isInfinite`. `isInfinite` is not the logical negation of `isFinite` since `isFinite(NaN)` and `isInfinite(NaN)` are both `false`.

conditions, and finally return the computed result and restore the original rounding mode and trapping status while merging any new signals with the old. This methodology cannot be used in our implementation since Java does not include methods to set or query the dynamic rounding modes, trapping status, or sticky flags. Instead, many methods of our implementation share the following general logical structure:

1. Screen for special cases such as NaN, Infinity, and ±0.0.
2. Bitwise convert the floating point argument(s) to integer.[3]
3. Separate the integer representation(s) into sign, exponent, and significand.
4. Build a new integer representing the floating point return value.
5. Bitwise convert the integer answer to floating point.
6. If signaling is involved, perform an operation that simultaneously yields the proper value and signals the appropriate condition(s). For overflow and underflow, an operation that merely sets the appropriate flags cannot be used due to the possibility of trapping on that condition (an incorrect value would be reported to the trap handler). A method that explicitly sets the flags would not be a suitable substitute for evaluating an appropriate expression since the method would not necessarily have the right trapping behavior on all processors.[4]

This structure prevent spurious sticky flags from being set during the computation. Before returning, only floating point operations that do not raise any flags can be executed inside a method. These operations include scaling a floating number by a power of two in the absence of overflow and underflow. The possibility of trapping on underflow precludes performing any intermediate operations that might generate subnormal numbers. This inconvenient restriction often forces operating on integer representations of floating point numbers.

2.2.1. Avoiding Potential Compiler Difficulties

While generating signals by evaluating an expression is the proper general technique, language definitions and optimizing compilers prevent its straightforward usage. The definitions of most programming languages, including Java, do not include the IEEE 754 sticky flags as part of the language semantics. Their absence licences optimizing compilers to perform constant folding at compile time. For example, the simplest expression that can be used to signal divide by zero is

```
1.0/0.0
```

(this expression is used by Java to create the infinity fields in the `Float` and `Double` classes). Even when run without optimization flags, a Java compiler will most likely replace such an expression by a floating point constant, preventing the runtime signal. Inhibiting this behavior requires making it less obvious to the compiler that the expression's operands are compile-time constants. For example, by assigning one of the operands to a variable, a compiler must perform constant propagation before constant folding can be done.

```
double zero=0.0;
1.0/zero;
```

However, such a simple program transformation does not guarantee constant folding will be avoided. With Sun's JDK 1.1.3, this alternation prevents constant folding when `javac` is run without the optimization flag but the quotient is replaced by ∞ when the `-O` option is used. To present a greater challenge to the compiler's constant folding efforts, our implementation uses decimal to binary conversion.

```
1.0/Double.valueOf("0.0").doubleValue()
```

To make this technique more concise, we employ a small `static` wrapper function, `atof`.

```
1.0/atof("0.0")
```

To defeat this obfuscation, a compiler would have to perform partial evaluation; a JIT could also thwart this code by using memoization. However, future optimizing compilers are not likely to optimize away this base conversion since a programmer is unlikely to unintentionally call a conversion method instead of simply writing a literal. Analogous

[3] Bitwise converting between floating point and integer means creating an integer with the same bit pattern as a given floating point number (or vice versa). Java supports this conversion with the methods `Float.floatToIntBits`, `Float.intBitsToFloat`, `Double.doubleToLongBits`, and `Double.longBitsToDouble`.

[4] The standard does not specify the interaction of flags and traps. Processors have used different policies and languages have not provided a uniform interface to these features.

code in C survived under the highest optimization level on multiple compilers. Using decimal to binary conversion also keeps the source code more readable than if other, more indirect, techniques were used to suppress the spurious optimizations.

No non-exceptional IEEE arithmetic operation can discriminate between `±0.0`. In most computations the sign of zero does not have much noticeable affect, determining the sign of the result only if the result is zero. But, the expression `1.0/0.0` can be used to differentiate between positive and negative zero since `1.0/(±0.0)` yields $\pm\infty$ with the same sign as the denominator.

2.2.2. Incomplete specification of the recommended functions

The descriptions of the recommended functions in the IEEE 754 and 854 standards are terse and do not fully specify the desired behavior in all situations. For example, `scalb` "returns $y \times 2^N$ for integral values N without computing 2^N" [1]. Effectively, `scalb` attempts to return a number having a different exponent than the input but the same significand. If `scalb` returns a subnormal result, precision may be lost to rounding.

The semantics of IEEE 754 arithmetic operations are described according to their behavior, not their implementation. For example, the answers produced by IEEE 754 arithmetic operations are succinctly required to act as if they "first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format" subject to the dynamic rounding mode [1]. The standard makes no mention of or reference to known implementation strategies or potential difficulties.[5] In contrast to the rest of IEEE 754, the description of `scalb` implies properties of its implementation, namely that 2^N should not be computed. This comment can be interpreted to obliquely require that the range of the scaling factor should exceed the largest power of two that can be stored by a floating point number of the same format. If that is the standard's intention, it would be considerably clearer simply to state the acceptable range of N. Another way to interpret the requirement of not computing 2^N is that `scalb` should be a bit manipulation operation instead of an arithmetic operation. In other words, the computation of 2^N is to be avoided simply by changing the exponent bits of the floating point argument. If `scalb` is defined as a bit manipulation operation, the result should not depend on the dynamic rounding mode and round to nearest should always be used.

While the authors of the standard intend for `scalb` to act as an arithmetic, not bitwise operation [16], any ambiguity could be remedied by replacing the incongruous specification of `scalb` with a slightly longer specification that stated the desired behavior.

Since the standard does not discuss the maximum allowed scaling range, the overflow behavior of `scalb` is also not fully specified. Depending on the rounding mode, if a result is too large in magnitude to represent, either infinity or `±MAX_VALUE` is returned by an untrapped floating point operation. When trapping on overflow, the trap handler returns a result with the same significand as the overflowing value but with an exponent adjusted by a known amount so the number is representable. With an arbitrary integer scale factor, the true result may be a number so large in magnitude that it cannot be represented even with an adjusted exponent. The standard does not describe how `scalb` should behave in these extreme circumstances.

The `logb` function returns the unbiased exponent of its floating point argument. The IEEE 754 and IEEE 854 standards specify different behavior for `logb` on subnormal values (cf. §3.4).

The treatment of NaN values presents a minor issue for implementing the recommended functions. Often when a recommended function takes a NaN argument, NaN is returned as a result. Patterned after the arithmetic operations, our implementations of the IEEE recommended functions return the input NaN when a NaN argument causes the output to be NaN. This behavior cannot be guaranteed by returning the Java's "canonical" NaNs, the `Float.NaN` and `Double.NaN` class variables.

The remainder of this paper describes each recommended function individually in §3, discusses a software implementation of bitwise conversion between floating point and integer types in §4, and concludes with comparisons to other implementations of the IEEE recommended functions. The appendix presents a method to determine the dynamic rounding mode based on evaluating floating point expressions.

[5] Since its adoption, numerous processors claiming IEEE 754 compliance have shipped with faulty floating point units, most notably the infamous Pentium divide bug [19].

3. IEEE 754/854 Recommended functions

The debugging of floating point subroutines is usually a difficult job,
since there are so many cases to consider.
—Donald Knuth
The Art of Computer Programming, vol. 2, 2nd edition

The following sections discuss each IEEE recommended function, including the overall behavior, implementation issues, and alternative implementation techniques. Java has two floating point types, `float` and `double`, and our implementation provides methods acting on both these types. In principle, our implementation could be ported to work on `double extended` values if it were possible to bitwise convert double extended to some integer type. Since our implementation is intended to be used in an environment sensitive to sticky flags and floating point exceptions, the programmer interface to a method includes not only the types of the parameters, the type of the return value, and the exceptions a method may throw, but also the sticky flags a method inspects and modifies. This information is documented with a Borneo-style method signature; the `admits` list for the flags a method inspects and the `yields` list for the flags a method sets. For a given function, the description of the implementation applies equally to the `float` and `double` versions since the versions share the same logical structure; therefore, only the `double` versions are discussed. For our implementation, given a function for `double`, the corresponding `float` version can be fairly mechanically synthesized by replacing `double` local variables with `float`, `long` variables with `int`, and changing bit masks and loop bounds accordingly. In some cases, using `double` calculation inside `float` methods could simplify and clarify the method. For example, since `float` subnormals are normal numbers in `double`, intermediate operations on `float` subnormals that could cause an underflow exception can be performed unexceptionally in `double` arithmetic. The implementation assumes that the floating point trapping status and rounding mode are constant throughout the execution of a method. The functions are discussed in roughly order of increasing implementation complexity. Figure 1 shows the dependency relationships within our implementation of the IEEE recommended functions; the variants of `logb` are explained in §3.4.

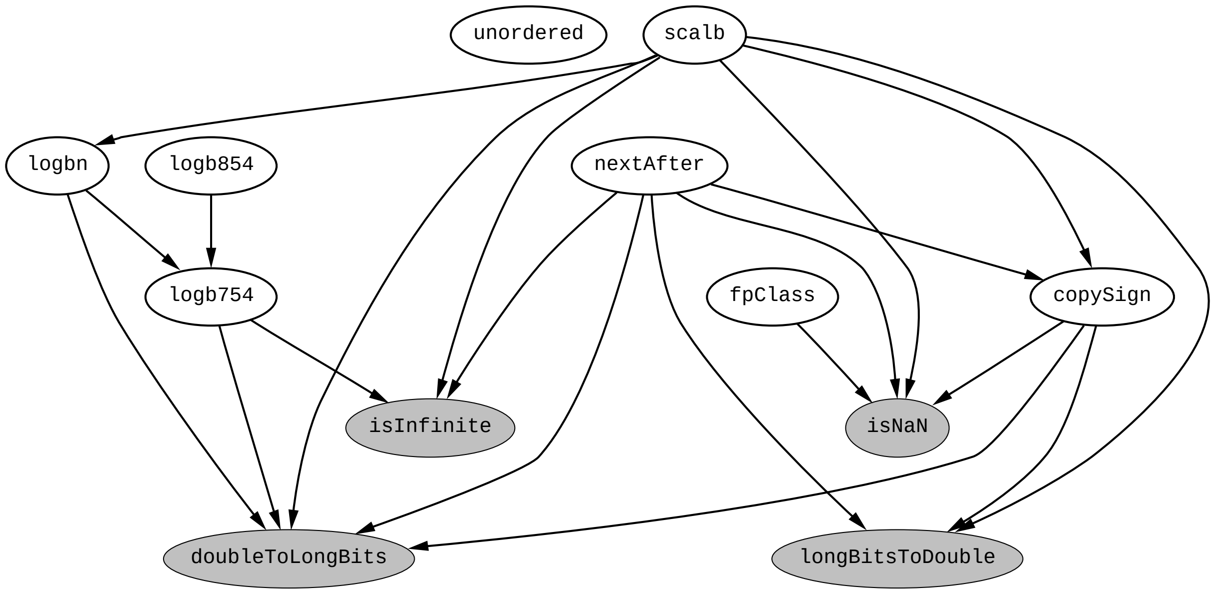

Figure 1 — Call graph among IEEE recommended functions. Shaded methods are already in the Java API.

3.1. copySign

```
public static double copySign(double value, double sign)
      admits none yields none
```

The functionality of `copySign` is used to implement complex log and complex square root at their discontinuities as well as for Sturm sequence and eigenvalue computations [7].

The `copySign` method returns the first argument with the sign of the second argument without signaling divide by zero if `sign` is `±0.0`. The implementation is very straightforward; see if either argument is NaN, if so return the arguments' sum, otherwise convert `sign` to integer and isolate the sign bit, convert `value` to integer and isolate the exponent and significand, logically OR the two integers together, convert the resulting bit pattern to `double` and return.

The standard ignores the sign bit of a NaN, so copying another number's sign bit into a NaN would not cause any problems, but our implementation checks for either argument being NaN to avoid unnecessarily creating a NaN different than the input NaN.

3.2. unordered

```
public static boolean unordered(double comparand1, double comparand2)
      admits none yields none
```

For the unordered relation to be true between two floating point numbers, at least one of the numbers must be a NaN. Therefore, unordered is logically equivalent to

```
isNaN(comparand1) || isNaN(comparand2).
```

Since a NaN value compares unequal to itself, `isNaN` can be implemented by

```
(comparand1 != comparand1).
```

(Java semantics correctly preclude the above comparison being replaced by `true`.) While this version of `isNaN` would be fine in a true IEEE 754 environment, it cannot be used in our implementation because of the possibility of an unwanted invalid signal.

The standard recommends that the comparison operators other than `==` and `!=` signal invalid when given a NaN argument. Processors typically have separate comparison instructions (or instructions with condition codes) to indicate whether or not signaling on NaN is desired. However, these capabilities are lacking in Java and the JVM.

Java semantics do not include the IEEE sticky flags. The JVM has two floating point comparison instructions per floating point type [17]. Instead of returning a boolean value, both floating point instructions return –1, 0, or 1 if the first argument is less than, equal to, or greater than the second. The unordered relation is not consistently indicated. One instruction returns –1 for unordered, the other 1. Within the JVM there is no way to determine whether one, both, or neither of these JVM comparison instructions raises invalid on a NaN argument. Therefore, we provide our own `isNaN` method that converts its argument to an integer to prevent any spurious signals.

Another expression to that could potentially be used to calculate unordered relies on NaN arguments breaking the equivalence between (`a > b`) and (`a <= b`). The expression

```
(comparand1 < comparand2) != (comparand1 >= comparand2)
```

returns the unordered relation between two values. But even when following the standard this expression would signal an unwanted invalid condition when the values are unordered. Assuming `!=` does not signal invalid (so `isNaN` does not need bitwise conversion), the expression based on an inlined `isNaN` is more efficient; sometimes executing only one comparison and at most two while the alternative expression always performs three comparisons.

3.3. fpClass

```
public static int fpClass(double value)
      admits none yields none
```

There are nine different "kinds" of IEEE floating point numbers; NaNs, positive and negative infinities, positive and negative zero, positive and negative subnormal numbers, and positive and negative normal numbers. The `fpClass` method takes a floating point number and classifies it by returning its kind. First, NaN is filtered out to avoid spurious invalid signals. Next, `value` is compared against numbers at the boundaries between different kinds of numbers, such as the smallest normal number. To avoid a divide by zero exception, the sign bit of zero is determined with `copySign`. Between one and four comparisons are necessary to classify a floating point number.

3.4. logb family

```
public static double logb(double value)
      throws DivideByZeroException
      admits none yields divideByZero
```

The `logb` family of methods, `logb754`, `logb854` and `logbn`, all extract the unbiased exponent from a floating point number. Several variants are necessary to conform to both IEEE 754 and 854 and to provide other useful functionality. The variants differ in their treatment of subnormal numbers. Otherwise the variants return the same values and have the same exceptional behavior, signaling divide by zero when given a zero argument.

First, the floating point argument is converted to integer and the exponent field is isolated. For normal numbers, the unbiased exponent is returned. Identified by the value of the extracted exponent, infinity, NaN, and zero are treated separately. For zero the expression `1.0/0.0` is evaluated to create the proper signal. (As discussed earlier, `1.0/0.0` itself cannot be used directly since the compiler would likely constant fold such an expression.) The `logb754` variant returns the unbiased exponent stored in a subnormal number's representation, namely $E_{min} - 1$.

For subnormals, `logb854` returns E_{min} instead of $E_{min} - 1$. This allows subnormals to be identified as numbers where `|scalb(`*`number`*`, -logb(`*`number`*`))|` < `1.0`. Under the IEEE 754 scheme, this condition holds for some but not all subnormal values. Given an implementation of `logb754`, `logb854` can easily be constructed using one additional conditional statement. Since it conforms to the newer standard, `logb854` is named `logb` in our implementation.

The third variation is a normalizing `logb`; instead of returning a fixed exponent for subnormals, the number is treated as if it were normalized. Given `logb754`, implementing the normalizing `logb` amounts to finding the bit position of the leading one in the significand of a subnormal number. Although the most straightforward way to find the leading one is to keep doubling the subnormal value until it is greater than or equal to the magnitude of the minimum normal number, this approach cannot be used in our implementation because an underflow exception would be thrown if trapping on underflow were enabled. To avoid this problem, our implementation uses the integer representation of the floating point number to find the leading bit.

To find the normalizing exponent of a subnormal number, it would also be possible to scale a subnormal by an amount guaranteed to result in a normal number and to then find the exponent of this new value. However, since the integer representation is already available, this option was not used. Another way to determine the exponent of a subnormal value is to compare it against a table of subnormal powers of two. In general, after filtering for NaN, `logb` could be implemented by constructing a (possibly implicit) table of the floating point powers of two and performing binary search to find the largest power of two less than or equal to the argument, but this method seems too slow for further consideration.

3.5. scalb

```
public static float scalb(double value, int scale_exponent)
      throws OverflowExeption, UnderflowException, InexactException
      admits none yields overflow, underflow, inexact
```

The intention of including `scalb` as an IEEE recommended function was to provide a quick way to scale entire arrays [16]. One technique to avoid floating point exceptions when calculating on an array is to first scan the array to determine a scaling factor that prevents floating point exceptions, scale the array elements, and finally perform the calculation in question. In this application of `scalb`, overflow does not occur since the scaling factor is chosen to avoid that exception. In this restricted case, `scalb` can be implemented with a few multiplications per array element. Since our implementation of `scalb` must behave properly for all scaling values, it can be considerably slower than several multiplications.

The main idea in our implementation of `scalb` is to convert the floating point argument to integer, determine what exponent the resulting floating point answer should have, and return a reconstituted floating point value with the adjusted exponent. Getting proper exception handling for extreme scaling values complicates the implementation. The scaling factor is $2^{\texttt{scale\_exponent}}$.

First, zeros, infinities, and NaNs are screened out (`scalb` is the identity function for these values regardless of scaling factor). Next, extreme scaling factors are tested for and handled separately. If the magnitude of `scale_exponent` exceeds $E_{max} - E_{min} + (N-1)$, no precision of the input would be left after scaling. After extracting the floating point argument's exponent with `logb`, the final exponent of the result is determined by summing `scale_exponent` and the argument's exponent. Testing for such extreme `scale_exponent`s prevents misleading integer overflow when the scaling factor is near ±`Integer.MAX_VALUE`.[6] If the exponent indicates a normal number, the proper floating point value is created and returned. No exceptional conditions arise if the result is a normal number; if the exponent indicates a subnormal or overflow, additional handling occurs.

A multiplication is used to generate the proper signals and proper value for the trap handler in the case of overflow and underflow. With unbounded range, the exponent of the product of a floating point multiply is (approximately) the sum of the exponents of the two operands. However, extreme scaling exponents can result in numbers too large or too small to be generated by a floating point multiplication. To maximize the range of overflowed numbers that can be created by a multiply, the significand of the `value` argument is first scaled to be E_{max}. Then a multiply by an exact power of two generates and returns the proper value sensitive to the current rounding mode. Such a multiplication signals overflow and inexact. If the exponent of the answer is outside of the range that can be generated, an `ArithmeticException` is thrown.

If the trapping status could be sensed without causing an exception, `scalb` could handle extreme scaling factors more gracefully. In non-trapping mode, `scalb` could return infinity and raise the appropriate flags. When operating in trapping mode, instead of throwing an `ArithmeticException`, `scalb` could construct and throw an exception with the proper scaled value instead of relying on trap handlers to spawn the exception. The exponent adjust used on overflow and underflow is larger in magnitude than the exponents of the largest and smallest representable numbers. Therefore, the wrapped exponent scheme can encode numbers that cannot be generated by any IEEE 754 arithmetic operation on numbers of the given format. If the result were outside of this larger bias-adjusted range, then zero or infinity, as appropriate, could be returned for the exception's scaled value.

Because of subnormal numbers, dealing with very small scaled values is slightly more complicated than dealing with very large ones. Subnormal numbers have less precision bits than normal numbers. A final result whose exponent could be generated by multiplying two subnormal numbers will not necessarily deliver a full complement of significand bits to the trap handler. Therefore, in our implementation, `value` is first scaled to have the exponent of the minimum normal number before being multiplied by a subnormal power of two to generate the answer and signals. Resulting exponents too small to generate cause an `ArithmeticException` to be thrown.

If `value` has trailing zeros in its significand, it is possible to "subnormalize" the number without losing precision information. In this way, values with smaller exponents could be properly generated for the trap handler. Our implementation does not use this technique because of its limited applicability. As with overflow, the ability to sense the trapping status would allow more informative handling of `scalb` underflow.

Another strategy for implementing `scalb` uses a sequence of multiplications by powers of two. At most three multiplies are needed to scale from the smallest subnormal to the largest normal value.[7] Testing to determine the exact number of multiplies to use takes additional work. Instead, the multiplies can be "strip mined;" for scaling up, multiply by $2^{\texttt{scale\_exponent} \bmod E_{max}}$ and then keep multiplying by E_{max} until all of the scaling factor has been accounted for. Such an approach may give a better average case performance but will perform unnecessary work when very large scaling factors are given (and will be further slowed if a processor operates on infinities considerably slower than normal values).

3.6. nextAfter

```
public static double nextAfter(double base, double direction)
      throws OverflowExeption, UnderflowException, InexactException
      admits none yields overflow, underflow, inexact
```

For a given argument `base`, `nextAfter` returns either the next larger or next smaller floating point value in the direction of `direction`. This functionality is useful for a variety of tasks. To test for numerical sensitivities, `nextAfter` can be used to slightly perturb input data. By taking the quotient of a floating point number and the

[6] `Integer.MAX_VALUE` == 2^{31} –1 and (`Integer.MAX_VALUE` +1) == -2^{31}. Therefore, if a positive unbiased exponent is added to a very large integer scaling factor, the resulting sum may be of the wrong sign, erroneously indicating an extremely small answer instead of an extremely large one (and vice versa).

[7] When scaling down, if subnormal factors are used, at most two multiplications are needed to go from numbers with the largest exponent to the smallest subnormal. However, on many processors operations on subnormals can be considerably slower than operations on normal floating point numbers; any time saved by performing one multiply fewer will be dwarfed by the increased latency of subnormal operations.

difference between that floating point number and its neighbor, the precision of a floating point format can be measured. By using similar relations, other format-dependent constants, such as the minimum normal value, can be expressed in a format-independent manner in terms of `nextAfter`, a few floating point literals, and IEEE arithmetic operations.

The `nextAfter` method is tricky to implement robustly due to its signaling behavior on overflow and underflow. The usual behavior of `nextAfter` is easy to understand and implement; return the value adjacent to `base` toward `direction`. (It is useful to be able to perturb a `float base` towards both a `float` and a `double direction`. Therefore, the `direction` argument of the `float` version of `nextAfter` is declared `double`, not `float`.) If the parameters compare as equal, return `base` (this preserves the sign of zero properly). If `base` is finite but the result is infinite, overflow and inexact are signaled. If the result is a subnormal, underflow and inexact are signaled.

Since the bits of floating point numbers (excluding NaNs) are lexicographically ordered when treated as signed magnitude integers, the adjacent floating point value can be gotten by incrementing or decrementing the integer representation of `base` and converting it back to floating point. Since Java's integers are 2's complement, negative values must be augmented in the opposite direction (incrementing a negative 2's complement value corresponds to decrementing the same bits interpreted as a signed magnitude integer). This logical core of `nextAfter` is implemented with only a few lines of code; the complexity comes from the exceptional cases.

Before any further processing, the arguments are determined to be non-NaN. To return an input NaN if at least one of the arguments is a NaN, the sum of the parameters is returned. Cases where infinity is one of the arguments do not require special handling. Having `base` equal to zero causes a slight complication. The bit pattern of negative zero does not behave properly as a signed magnitude number, but positive zero does. Therefore, if `base` is `–0.0`, it is replaced by `+0.0`. One idiom to accomplish this task without branching could be

```
base = base + 0.0;
```

which is intended to produce a positive zero if `base` is zero [15]. Unfortunately, while this trick works under three of the four rounding modes, it does not work under round to negative infinity. Therefore, since our `nextAfter` is meant to be run under all rounding modes, an explicit test for zero must be used.

To generate the proper signals when `nextAfter` returns an infinity from a finite base (such as `nextAfter(MAX_VALUE, INFINITY)`), one ulp is added to `MAX_VALUE`, which also gives the trap handler the proper significand. However, an explicit infinity must be returned for the code to work properly under all rounding modes. Under the round to zero rounding mode, no addition operation returns infinity from finite operands. More generally, under round to zero the only arithmetic operation that returns infinity from finite operands is division by zero.

When the result of `nextAfter` is subnormal, underflow and inexact must be signaled. As with the behavior on overflow discussed above, to cope with dynamic rounding modes `nextAfter` uses separate expressions to trigger the underflow trap handler (if present) and to return the floating point answer for non-trapping mode.

Trapping on overflow/underflow is used to provide the appearance of floating point numbers with unbounded exponents [16]. The overflow/underflow trap handler is supposed to provide an exponent-adjusted result. This result can be used for subsequent computation as long as the number of over/underflows is recorded.

Since trapping on underflow is meant to present the appearance of unbounded exponents, when `nextAfter` tries to return the next smaller or larger value, it should increment/decrement `base` as if `base` had all the precision bits of that format. As described above, a fixed, representable value is added to `base` to cause an overflow with the proper significand bits. However, a similar feat is not directly possible for subnormal numbers since subnormal numbers have less precision bits than normal numbers; the increment/decrement needed to perturb a subnormal number at full precision is not representable as a number in the same floating point format. To overcome this limitation, the subnormal number is first scaled into normal range.

After a subnormal number is scaled into normal range, at least one of the trailing significand bits is 0. Therefore, an ulp can be exactly added or subtracted. The last bit of this perturbed value will always be 1; if an ulp is added, no carry will ripple out since the least significant significand bit is 0; if an ulp is subtracted, a borrow will ripple down from the first 1 in the significand (if `value` is `0.0` subtracting an ulp also yields a 1 in the last bit). Therefore, scaling this perturbed number back into subnormal range signals both inexact and underflow. Inexact is signaled since at least one bit of precision is lost and underflow is signaled since the result is subnormal. This procedure delivers the proper bits to the underflow trap hander if trapping on underflow is enabled. If trapping on underflow is not enabled, the proper signals are also generated since the operation is inexact. However, if underflow is not being trapped on, due to different rounding modes, the result of the above calculation may not be the proper subnormal result. But, since the proper signals have already been generated, the integer representation of the answer can be converted to floating point and returned.

As an alternative to converting to integer, the adjacent number can be calculated exactly in floating point by adding or subtracting an ulp. The difference between two adjacent normal floating point numbers is always a representable power of two and the magnitude of the ulp is determined by a number's exponent. However, this approach does not have the proper exceptional behavior for subnormals; as discussed above, a more complicated expression would need to be evaluated. (An addition or subtraction resulting in a subnormal answer is always exact, for a proof see [10]).

4. Bitwise conversion between integer and floating point types

The Java API includes methods to bitwise convert between floating point and integer types of the same width (`Double.doubleToLongBits`, etc.). The JVM has no instructions to perform this bitwise conversion nor does the JVM have any untyped load/store instructions. Therefore, it is not immediately obvious how a Java environment can implement bitwise conversions between floating point and integer types.

Java allows `native` methods written in other languages to be called from a Java program. Sun's JDK calls C functions to bitwise convert between floating point and integer. Since C has untyped unions, an expedient way to bitwise convert between integer and floating point values is to have an untyped union between an integer and floating point type of the same size. To operate on the bitwise representations of floating point numbers, the relative "endianess" between the integer and floating point types must be known. If the C compiler does not support a 64 bit data type, the address of a `double` floating point number can be taken and, via casting, accessed as an array of two 32 bit integers. However, some Java environments (such as Java processors [18]) have no native code other than the JVM bytecode. Therefore, without additional JVM instructions, such a Java environment must implement these bitwise conversions in a program expressible in the Java language. The next two sections outline how to compute these conversions in pure Java using integer operations, existing value conversions between integer and floating point types, and floating point comparison, addition, subtraction, multiplication, and divide.

The desired signaling behavior of the bitwise conversion functions is `admits none yields none`. Without a mechanism to clear the sticky bits, it is not possible to implement `yields none` for floating point to integer conversion since the only arithmetic way to determine the sign of `0.0` raises the divide by zero flag (it is assumed `copySign` is not available). The integer to floating point conversion cannot create arbitrary NaNs; it must create the canonical NaN generated from evaluating invalid floating point expressions such as `0.0/0.0`. Likewise, an arithmetic implementation of integer to floating point conversion cannot preserve special NaN values.

A pure Java implementation of the bitwise conversion methods would allow our IEEE recommended functions to be "absolutely pure." Using a pure bitwise conversion avoids an API call that may indirectly employ an "impure" `native` method. (However, using `native` methods to implement core API functionality does not taint the purity of a Java library).

4.1. integer to floating point

```
public static double pureLongBitsToDouble(long bits)
        admits none yields none
```

Clearly it is feasible (but impractical) to implement a pure version of `longBitsToDouble` by using a *very* large `switch` statement. However, it is relatively straightforward to create the floating point doppelgänger of an integer value. Using bitwise logical operations, the bit positions encoding different fields of a floating point number can be isolated. From these isolated fields, corresponding floating point numbers can be created and then multiplied together to give the final floating point result. The existing value conversion between integer and floating assists in constructing the significand of the result.

Before what will become finite non-zero numbers are processed, the bit patterns of infinities and zeros are tested for and handled separately, as are the bit patterns ranging over NaNs. For finite values, the overall task is to construct three floating point numbers, one with the sign, another with the exponent, and a third with the significand of the target floating point number. Multiplying these three numbers together generates the answer. The sign component is set to `±1.0` depending on the high order bit of the integer. The exponent is extracted from the integer value via masks and shifts. Starting with `1.0 == ` 2^0, a loop keeps multiplying by `2.0==`2^1 (or by `0.5==`2^{-1}

depending on the sign of the exponent) until a value with the final exponent is reached. (For faster average execution, the floating point powers of two present in the exponent field can be created by repeated squaring and then multiplied together.) The significand is created by isolating the appropriate bits from the integer representation, ORing in the implicit bit for normal numbers, and assigning that integer to a float. This number is then scaled to lie between 0 and 2 by dividing by a fixed quantity (alternatively, the exponent value could be adjusted instead). Afterwards, the three components are multiplied together. For a practical implementation, trapping on underflow must not be turned on since generating subnormals would become impossible.

4.2. floating point to integer

```
public static long pureDoubleToLongBits(double value)
            throws DivideByZeroException    // unwanted exception
            admits none yields divideByZero // should be yields none
```

Bitwise converting from floating point to integer is surprisingly about as direct as converting in the other direction. The implementation first tests for NaN, infinity, and zero values. The sign bit of `±0.0` is determined by dividing `±0.0` into `1.0`. For other finite values, the sign bit can be found by comparing against `0.0`. Next, the exponent is determined by finding the largest power of two less than or equal to the absolute value of `value`. After finding the exponent, the significand can be ascertained.

Once the exponent is found, the number can be scaled to be an integer floating point value between 0 and 2^N. This value can then be exactly rounded to integer using existing language features. Finally, the three pieces of the integer representation are bitwise ORed together to form the final result.

5. Related Work

5.1. FDLIBM

The software package FDLIBM [8] (Freely Distributable LIBM) was developed and released by SunSoft as a public domain reference implementation of C's `libm` math library. Precursors to FDLIBM were distributed with versions of BSD UNIX. FDLIBM includes implementations of the more interesting IEEE recommended functions. FDLIBM assumes IEEE 754 arithmetic and only includes `double` precision versions of the functions. The functions in FDLIBM are, naturally, written in C and extensive use is made of interpreting the bits of a `double` as an integer using pointer arithmetic and casting. For exceptions, FDLIBM evaluates a set of standard expressions; for example, overflow is generated by MAXFLOAT*MAXFLOAT. This uniform treatment of exceptions does not conform to the exception model presented in §2.2 since the wrong bits would be given to the trap handler. In other respects, FDLIBM uses many of the techniques discussed in this paper. Circumlocution is used to undermine the compiler's optimization efforts; for example, to signal divide by zero FDLIBM uses`(x-x)/(x-x)` instead of `0.0/0.0`. FDLIBM's `nextafter` appropriately passes argument NaNs back out again by returning the sum of the parameters.

5.2. Algorithm 772

Approximately contemporaneously with FDLIBM, a separate C implementation of the recommended functions was released and described by Cody and Coonen [4]. Cody and Coonen provide portable versions of `logb`, `scalb`, `nextafter`, `copysign`, `isnan`, and `finite` for both `float` and `double` precision. A test suite accompanies their software. They use an approach similar to ours; screen for special values, perform some bitwise conversions, calculate the answer, etc. Cody and Coonen assume non-trapping mode is in effect (they normalize a subnormal by repeatedly multiplying by `2.0`) and they do not attempt to deliver the proper bits to the trap handler. However, their code is designed to be run under dynamic rounding modes. Additionally, their code behaves properly under both gradual underflow and (non-IEEE 754 compliant) flush to zero.

Cody and Coonen report having to work around several compiler bugs. For example, the decimal to binary conversion of `DLB_MAX` often gave an incorrect value (or even overflowed). To work around this deficiency, after screening out zero and NaN, their `finite` implementation returns `x * 0.5 != x` which is `false` only for infinity and zero. However, other compiler bugs, such as optimizing away `x != x` are not accommodated and they assume `x != x` has the proper non-signaling behavior when `x` is a NaN. A compiler may be tempted to eliminate as dead code floating point expressions that are only evaluated for their exceptional behavior. Cody and Coonen guard against this by assigning the value of the exceptional expression into a variable that is used in a conditional statement with a side effect. The conditional is constructed to always be false in a manner opaque to the compiler. Therefore, the compiler will not elide code for the exception-generating expression. For example, this technique can be used to handle the generation of infinity and signaling of overflow in `nextAfter`:

```
result = Float.MAX_VALUE + atof("1.0141205e31"); // signal overflow
if(Float.floatToIntBits(result) == 0x0)          // use result (but the test is always false)
  return 0.0;                                    // this return statement never executes
return copySign(Float.POSITIVE_INFINITY, base);  // return proper value (cannot use calculated
                                                 // value of result because of dynamic
                                                 // rounding modes)
```

Ironically, to prevent some statements from being discarded erroneously as dead code, unrecognizable dead code is introduced.

5.3. Instructions in the x86 architecture

Going back to the 8087, the x86 floating point architecture has included explicit instructions implementing some of the IEEE recommended functions. On the 8087, the FSCALE instruction implements `scalb` but assumes that the `scale exponent` is an integer between $\pm 2^{15}$; if the given `scale exponent` is outside of that range, the result is undefined and no exceptions are signaled [11]. The FSCALE instruction on the 8087 is also fast; about three times the speed of a general multiply (35 clock cycles versus 100). Newer x86 family chips have redefined FSCALE to appropriately return infinity or zero if the now unrestricted scaling factor overflows or underflows the bias-adjusted range [12].

Pentium and later x86 generations have reversed the relative speed of FSCALE and FMUL. On a Pentium multiplication has a 3 cycle latency and can be pipelined while FSCALE has at least a 20 cycle latency and cannot be pipelined [13]. Therefore, since at most 3 multiplies are needed to scale the smallest subnormal to the largest normal, a sequence of FMUL's will be faster than FSCALE on a Pentium. While a single `scalb` operation does not just consist of multiplication but also some testing, a software implementation of `scalb` based on successive multiplications may take longer than the FSCALE instruction (even ignoring function call overhead). However, in some applications of `scalb`, such as scaling all the elements of an array, the scaling factor is pre-calculated such that no overflows will occur. In such situations, FSCALE would be considerably slower than a loop over the array elements performing one or two multiplies on each element since this loop could be aggressively optimized (unrolled, blocked, scheduled, etc.). In practice, scaling an array uses one multiply per element [16].

The FXTRACT instruction decomposes a floating point number into a significand and exponent, useful for implementing `logb`. Later x86 chips slightly modified FXTRACT to conform to the IEEE 754 standard `logb` (the 8087 predates official adoption of IEEE 754). The FXAM instruction implements something comparable to `fpClassify`. On the 8087, FXAM executes in about half the time that a single comparison does (12-23 cycles compared to 42-52). On the Pentium FXAM cannot be pipelined and takes 21 cycles while FCOM takes 4 cycles and can be pipelined. As with FSCALE and `scalb`, the change in relative speed between FXAM and FCOM implies the common floating point instructions on the Pentium could yield a faster `fpClassify` core.

6. Conclusions

The IEEE recommended functions perform relatively simple but useful floating point tasks. Our implementation demonstrates the tediousness involved in writing robust versions of the IEEE recommended functions in "absolutely pure" Java. (Using the pure bitwise conversion routines would even eliminate transitive reliance on possibly impure API calls.) Lack of proper language support and the possibility of unwanted optimization exacerbates the difficulty of meeting the admittedly severe robustness goals. Merely testing these functions is also problematic since it is not immediately apparent how to access the relevant floating point features. A language having full IEEE 754 support would greatly ease writing floating point aware functions.

7. Acknowledgments

The initial versions of these robust IEEE recommended functions were written while the author was a summer intern in Gregory Tarsy's floating point group at Sun. Alex Liu, a member of Sun's floating point group, helped the author through some of the trickier aspects of coding these functions. Stuart McDonald and Gregory Tarsy gave good suggestions on improving the presentation. Professor William Kahan at Berkeley also provided useful help and feedback on both the implementation and this paper.

8. Appendix — Dynamic rounding mode detection by evaluating floating point expressions

Our implementation of the recommended functions are oblivious to the rounding mode; they operate under any rounding mode but do not know what rounding mode is being used. This ignorance occasionally leads to more complicated and subtle code. For example, if the rounding mode is known to be round to nearest, `nextAfter` could avoid returning an explicit infinity for overflow conditions and simply return `MAX_VALUE` plus an ulp. Also in `nextAfter`, as long as the rounding mode is not round to negative infinity, adding a positive zero to `base` can be used to avoid an explicit test for zero. In methods more complicated than the recommended functions, it may not be easy or convenient to write robust functions that work under all rounding modes while being unaware of the actual rounding mode. It would be convenient for such methods to branch and execute different code based on the rounding mode. While a full IEEE 754 environment would have an explicit "`getRound`" method for this purpose, Java does not. Fortunately, it is possible to infer the rounding mode by evaluating a few aptly chosen floating point expressions.

The "exact" result of a floating point operation before being rounding to its destination, is either exactly representable as a floating point number or falls between two representable floating point numbers. If the result is not exact, the dynamic rounding mode sets the policy for determining which of the bordering representable numbers to deliver. For given operands, an arithmetic floating point operation has at most two possible answers; which one is returned depends on the rounding mode. Therefore, by examining the results of evaluating a known set of expressions, the rounding mode can be found. Since there are four rounding modes, at least two operations are needed and, as it turns out, exactly two suffice. The rounding threshold is the smallest positive floating point number such that, under round to nearest, when the threshold is added to `1.0`, a number larger than `1.0` is returned. Positive numbers smaller in magnitude than the rounding threshold are rounded away when added to `1.0`. Under some other rounding modes, `1.0` plus the rounding threshold returns `1.0`. Using this expression and its negation, the dynamic rounding mode can be inferred, as shown in Table 1. The `float` rounding threshold is $2^{-24} + 2^{-47} \approx 5.960465 \cdot 10^{-8}$ and the `double` rounding threshold is $2^{-53} + 2^{-105} \approx 1.1102230246251568 \cdot 10^{-16}$. In general, the rounding threshold for a format is equal to `nextAfter((nextAfter(1.0, ∞) – 1.0)/2.0, ∞)` where the floating point literals are in the given format.

Table 1 — Determining the rounding mode by expression evaluation.

`1.0f + ROUND_THRESH == 1.0f`	`-1.0f - ROUND_THRESH == -1.0f`	Rounding Mode
`false`	`false`	to nearest
`false`	`true`	to positive infinity
`true`	`false`	to negative infinity
`true`	`true`	to zero

9. References

[1] ANSI/IEEE, New York, IEEE Standard for Binary Floating Point Arithmetic, Std 754-1985 ed., 1985.

[2] *Apple Numerics Manual, Second Edition*, Apple, Addison-Wesley Publishing Company, Inc., 1988.

[3] W.J. Cody et. al, "A Proposed Radix-and Word-length-independent Standard for Floating-Point Arithmetic," IEEE Micro vol. 4, no.4, August 1984, pp 86-100.

[4] W.J. Cody and Jerome T. Coonen, "Algorithm 772 Functions to Support the IEEE Standard for Binary Floating-Point Arithmetic," ACM Transactions on Mathematical Software, vol. 19, no. 4, December 1993, pp. 443-451.

[5] Jerome Coonen, "A Proposal for RealJava, Draft 1.0," July 4, 1997, Numeric Interest Mailing list, http://www.validgh.com/java/realjava.

[6] Joseph D. Darcy, *Borneo 1.0: Adding IEEE 754 floating point support to Java* , M.S. Thesis, University of California, Berkeley, 1998.

[7] James W. Demmel, Inderjit Dhilon, and Huan Ren, "On the Correctness of Parallel Bisection in Floating Point," Computer Sciences Division Technical Report UCB//CSD-94-805, 1994 (also LAPACK Working Note number 70, http://www/netlib.org/lapack/lawns/lawn70.ps).

[8] FDLIBM, http://www.netlib.org/fdlibm/index.html

[9] James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, 1996.

[10] John R. Hauser, "Handling floating-point exceptions in numeric programming," ACM Transactions on Programming Languages and Systems, vol. 18, no. 2, March 1996, pp. 139-174.

[11] Intel Corporation, *The 8086 Family User's Manual: Numerics Supplement*, July 1980.

[12] Intel Corporation, *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1997.

[13] Intel Corporation, *Pentium® Processor Family Developer's Manual Volume 3: Architecture and Programming Manual*, 1995.

[14] Roger Hayes, *100% Pure Java Cookbook*, Sun Microsystems, Inc., http://java.sun.com/100percent

[15] William Kahan, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/ieee754.ps.

[16] William Kahan, personal communication.

[17] Tim Lindholm and Frank Yellin, *The Java™ Virtual Machine Specification*, Addison-Wesley, 1996.

[18] J. Michael O'Conner and Marc Tremblay, "PicoJava-I: The Java Virtual Machine in Hardware," IEEE Micro March/April 1997.

[19] H. P. Sharangpani and M. L. Barton, "Statistical Analysis of Floating Point Flaw in the Pentium™ Processor (1994)," Intel Corporation, November 30, 1994, http://www.intel.com/procs/support/pentium/fdiv/fdiv.htm