A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages

by

Samuel A. Figueroa del Cid

A dissertation submitted in partial fulfillment
of the requirements for the degree of Doctor of Philosophy
Department of Computer Science
New York University
January, 2000

Robert B. K. Dewar
Dissertation Advisor

Contents

List of Tables

Chapter 1

Introduction

1.1 The IEEE Standard for Floating-Point Arithmetic

The floating-point units of most general-purpose processors in use today conform to the IEEE Standard for Binary Floating-Point Arithmetic [47], from hereon referred to as simply "the IEEE Standard." (There is a corresponding international standard [46].) One of the reasons this standard has become so widely adopted is that its designers restricted its applicability to floating-point arithmetic engines (perhaps at least partially for pragmatic reasons). Accordingly, the IEEE Standard specifies such things as data formats and what values are representable in them, basic operations that can be performed on representable values, and detailed descriptions on how these operations are to be carried out. These descriptions stipulate how results are to be rounded to fit within the constraints of these data formats, what the behavior should be when special (or exceptional) computational situations arise, and in which areas control can be effected on a conforming engine's behavior at the time a computation is proceeding. However, the IEEE Standard stops short of specifying what high-level language constructs are to be used to take advantage of the features it describes.

In actuality, there is a second, more general standard for floating-point arithmetic: the IEEE Standard for Radix-Independent Floating-Point Arithmetic [48]. This second standard can apply to either binary or decimal floating-point systems, for example. Also, whereas the former specifies for at least some of the data formats it defines what bit patterns are to be used to encode representable values, the latter does not go to this extent, and in fact does not even give a specific width for any of the data formats it describes.

Although it is theoretically possible to devise implementations of [47] that do not conform to [48], in practice, many binary floating-point arithmetic engines conform to both standards. In addition to specifying the behavior in edge cases more completely in a few situations, the latter standard requires the inexact exception to be signaled appropriately during certain conversions, and adds two more functions to the list of recommended functions in the appendix—see chapter 7.

Instead of exhaustively describing the IEEE Standard here, most subsequent chapters explore one aspect of the IEEE Standard in detail, noting how much of that aspect is typically

implemented in hardware, and to what extent that aspect is or could be supported in high-level languages. The penultimate chapter outlines how to support the IEEE Standard in two popular high-level languages, Ada and Java, and the appendix contains a more detailed description showing how the IEEE Standard could be supported in Ada.

1.2 What does it mean to support the IEEE Standard?

Some language definitions, especially recent ones, now make reference to the IEEE Standard. Haskell 98 [13] is an example (picked at random) of such a definition. On target platforms in which IEEE-Standard-conforming floating-point arithmetic engines are available, a couple of data formats defined by the IEEE Standard are available, as are most, if not all, the operations described therein. Presumably, special computational situations are handled as the IEEE Standard requires as well. One could ask, "What more could one want beyond this (level of support)?" The next chapter mentions specific criteria that language designers should keep in mind if they truly wish to fully support the IEEE Standard, and will make clear that providing full support for the IEEE Standard is not as trivial as many language definitions, including that of Haskell 98, would make it appear.

1.3 Considerations in deciding how to support the IEEE Standard in high-level languages

In deciding how to support the IEEE Standard in programming languages, it is important to become familiar with how this standard is implemented in various floating-point arithmetic engines, as well as what features tend to be implemented in hardware, and what features are commonly implemented in software. This way, one can avoid making access to the IEEE Standard's features costly. Otherwise, the effort in figuring out how to support the IEEE Standard can be in vain, since programmers will be unlikely to frequently use features that cause their programs to run more slowly. This is especially true for many who write programs that intensively use floating-point arithmetic, since these programs need to perform as efficiently as possible.

Perhaps the most widely used floating-point arithmetic engine today is the one incorporated in Pentium processors from Intel [6], as well as other processors conforming to this architecture. Subsequent chapters cover this processor's support for IEEE Standard in detail. Other interesting or commonly-used architectures, such as the Motorola 68000 series [1] and PowerPC architecture [8], Sun's SPARC [70], Hewlett-Packard's PA-RISC [5], DEC Alpha [2], and Inmos T9000 [9], are also examined. (Although Intel's and Hewlett-Packard's upcoming IA-64 architecture is not discussed, it is similar to Intel's x86 architecture [3].)

Floating-point arithmetic engines that support floating-point formats with differing precision will often round results more than once—first to fit some intermediate destination, and then again to fit the final destination. (Chapter 6 discusses this in greater detail.) This double rounding can cause results to differ from what would be obtained if results were rounded just

once, so understanding when double rounding matters and when it does not, and what the consequences are in avoiding double rounding (especially in terms of a program's performance) is important from the context of programming language design.

It is also important to consider how programmers may want to use the features of the IEEE Standard. Subsequent chapters discuss these matters as well.

Finally, current languages' support (or lack thereof) for the IEEE Standard needs to be assessed, and different ways of supporting the IEEE Standard's features need to be considered. Hundreds of languages have been designed over the years. Writing about each of them would make this work excessively long, so only a few languages are discussed in any detail. These include languages in which numerically intensive programs have traditionally been written, such as Fortran, C, C++, and Ada. In addition, some extensions to these languages are mentioned, as well as how the IEEE Standard's features are (or are not) made available in implementations of these languages. Some newer languages are also mentioned, such as Java and variants thereof, Limbo, and the author's own μln.

1.4 Prior Related Work

There have been several attempts to support the IEEE Standard in high-level languages. As previously mentioned, some newer languages, like Haskell 98 [13], Modula-3 [63], and Java [38], tend to support those features that are fairly trivial to support with traditional language facilities, but which many older languages don't even support explicitly. (This is not to say that it is impossible to largely achieve the same level of support in implementations of some of these older languages, such as what the Optimizing Oberon Compiler [64] does for the Oberon-2 language [62], only that these older languages generally do not mention how conforming implementations should support the IEEE Standard.)

More complete and serious attempts to support the IEEE Standard in (existing) high-level languages include: the Standard Apple Numeric Environment (SANE) [12], which includes bindings for C and Pascal; a set of extensions to the C language from the Numeric C Extensions Group (NCEG), whose work is being incorporated into the upcoming revision of the C language, popularly known as C9X [17]; and RealJava [21] and Borneo [22], both of which are dialects of Java. These efforts are discussed in subsequent chapters.

In all of these efforts, and in contrast to this thesis' proposal for how to support the IEEE Standard in Ada [11], the language itself has been modified in some way to achieve some satisfactory level of support for the IEEE Standard. It was this lack of elegance that motivated this author to design the μln language [30], which was a study on how well the IEEE Standard could be supported in a high-level language if there were no constraints on what the characteristics of the language were. μln is also discussed in subsequent chapters.

Another problem with some of these efforts, and with Dewar's binding for Ada 83 [10, 24] (which provides reasonably complete support for the IEEE Standard), is that making use of the features the IEEE Standard describes is not always as convenient and natural as one might wish: access to many of these features is often available only through function or procedure calls, rather than via higher-level abstractions, as proposed in the next chapter.

A somewhat different strategy for "supporting" the IEEE Standard is that of the Language Independent Arithmetic Standard (LIAS) [45]. One way to view LIAS is as a guide as to how to define and document (possibly without explicitly mentioning the IEEE Standard) the characteristics and semantics of languages' floating-point facilities, which is a serious deficiency in many language definitions. (For example, a Fortran language processor may insist that the sum of 2 and 2 is 5, and still fully conform to the Fortran standard! Among other things, this is the kind of nonsense LIAS purports to eliminate.) Brown's model for floating-point arithmetic [14], which Ada adopted in a modified form, can also be viewed as an earlier example of this kind of strategy.

One problem with this kind of strategy is that, in an attempt to be applicable to a wide variety of machines, both real and imagined, the model of arithmetic these standards provide is not sufficiently specific [34]. Consequently, one has to program defensively, even to the point of making one's programs work on imaginary machines that exhibit highly ludicrous behavior that will never occur in practice [53]. Indeed, it is sometimes quite challenging to devise algorithms that will work on machines that could possibly conform to these models of floating-point arithmetic. The reality is that actual machines' floating-point behavior, especially that of the most current computers, does not differ nearly as much as these models allow. That is, these models of floating-point arithmetic are not sufficiently specific, and thus they fail, for example, to make available all the features of the IEEE Standard.

In addition, these approaches do not satisfactorily answer the question of what it means to fully support the IEEE Standard, though some have mentioned some of the issues relevant to supporting the IEEE Standard in a high-level language [28, 27, 26]. Now, some might argue that a higher level view of floating-point arithmetic is desirable in high-level programming languages. After all, they may say, once optimizing language processors start tweaking the code, the details of floating-point behavior become somewhat fuzzy. However, a hallmark of good programming languages is allowing for a high level of abstraction, while at the same time appropriately providing control over the small details that, at least for numerical programs, can make the difference between code that works reliably every time, and code that provides satisfactory results as long as all the astrological bodies are aligned just so.

Chapter 2

What Does It Mean to Support the IEEE Standard?

As was mentioned in the previous chapter, some recent language definitions (Java [38] among them) seem to equate supporting the IEEE Standard with providing access to one or two data formats and to most of the operations described therein. Words to the effect of "all operations are performed as described in the IEEE Standard" may even appear in some of these language definitions. As will be seen in this chapter, such support falls short of what one should expect from a high-level programming language that truly supports the IEEE Standard.

Subsequent chapters explain in detail what the features of the IEEE Standard are, but very briefly, this standard gives specifications for: four different (but not necessarily distinct) data formats and the set of representable values in each of these formats; a set of operations on these values; how the results of these operations are to be rounded to fit their destinations (if applicable); what happens when special computational situations arise; and, though not strictly part of the Standard, a set of recommended functions (described in an appendix) that ought to be available.

Though typically implemented as a combination of hardware and software, and sometimes even purely in software, the IEEE Standard is written in such a way as to allow it to be implemented purely in hardware. Little guidance is provided as to how its features should be made available in a high-level programming language. While it could be argued that some features are more appropriate for low-level programming, the fact remains that some of the IEEE Standard's useful features are not generally available in the programming languages that are commonly in use today.

The question, then, is "What does it mean to support the IEEE Standard in a high-level programming language?" The answer this work proposes is as follows: A high-level programming language properly supports the IEEE Standard if it:

1. Provides access to all the floating-point data formats described in the IEEE Standard supported by machines that might be executing programs written in the given language. (It is not good enough to give the excuse that programs written in a given language can

only run on some imaginary machine that only supports a certain subset of the data formats the IEEE Standard describes, and that therefore the language only provides access to those data formats.) See chapter 3.

2. Provides access to all the operations described in the IEEE Standard (including all 26 predicates related to the comparison of two floating-point values), as well as to the functions described in the appendix of the IEEE Standard for Radix-Independent Floating-Point Arithmetic [48]. See chapter 7.

3. Provides representations for all special values, such as infinities, NaNs, and negative zero, in such a way that they can be used as compile time constants, and without incurring side effects (such as setting status flags, invoking traps, or otherwise signaling that some special computational situation has occurred) by virtue of simply mentioning them in a program (that is, without actually using them in some computation). See chapter 8.

4. Provides the ability to specify what rounding mode (and rounding precision mode, if appropriate) should be used when performing arithmetic at compile time, as well as the ability to arbitrarily change at run time the rounding mode (and rounding precision mode, if appropriate) in effect for floating-point operations. See chapters 4 and 5.

5. Provides access to the status flags described in the IEEE Standard, and, if feasible, the ability to associate and disassociate trap handlers with the various exceptional situations described in this standard. See chapter 8.

6. Confers upon programmers the ability to write code in such a way that, were the values of the various operands appearing in any code fragment known, along with enough of the context in which the given fragment appears, and the basic implementation-defined characteristics of the language processor being used, one would be able to determine what the result of the computation would be, regardless of how aggressively the language processor in question applies various transformations in an attempt to improve the quality of the generated code. (In other words, results should be deterministic, and not depend on whether certain optimizations or code improvements were or were not performed.) See chapter 9.

7. Provides all the above using the natural syntax and facilities of the language. (It is not enough, for example, to provide access to the operations described in the IEEE Standard solely through normal function calls, if one would typically use infix syntax to denote the same kinds of operations when writing programs in a given language. Nor is it good enough to provide access to the data formats described in the IEEE Standard using composite types, such as records or structures, if the language already provides primitive floating-point data types.)

8. Does not gratuitously reduce performance in general, especially when none of the specific features of the IEEE Standard are being used at a given point in a program. (An example of unnecessary performance degradation is requiring the status flags to be checked after

every floating-point operation in order to be able to detect, say, exactly where [if ever] the invalid operation exception is signaled during the course of a computation. Another such example is requiring every floating-point result to be explicitly checked in an effort to avoid double rounding at all costs—Java's traditional floating-point semantics basically requires the latter when certain processors simulate the Java Virtual Machine's behavior [38]. See chapters 6 and 9.)

Note that strictly speaking, the last three requirements do not emanate directly from the IEEE Standard. However, going through the exercise of making the features of the IEEE Standard available in a high-level language would clearly be pointless if it were not possible to predict what the results of a computation would be (see requirement 6 above), since this would conflict with "the desiderata that guided the formation of [the] standard," as reported in the Standard's foreword [47]. Also, if the IEEE Standard's features were made available in such a way that they were either cumbersome to use or imposed a significant performance penalty, they would be highly unlikely to be used, except perhaps for a few cases in which there were very clear benefits to their use, again defeating the purpose of the IEEE Standard.

As will be seen in subsequent chapters, very few language definitions fully support the IEEE Standard as outlined above. It will be shown that, depending on the given language definition, it is possible in some, though not all, cases to supplement a language in such a way as to fully support the IEEE Standard, without amending the language definition itself. It will also be shown how a language can superficially appear to support the IEEE Standard, but yet require fundamental changes in order to truly support this standard, as outlined above.

Chapter 3

Supporting Data Formats in High-Level Languages

3.1 IEEE Standard requirements in regards to data formats

The IEEE Standard defines four data formats: two basic formats and two extended formats. The basic formats are the single and double formats, and the extended formats are the single extended and double extended formats. The single format is the only format whose support is required. Support for the other formats is optional, though the IEEE Standard does recommend support for the extended format corresponding to the widest supported basic format.

Each of these formats must be capable of representing a certain set of values, including certain nonzero finite values, at least one signaling and one quiet NaN (Not a Number), and positive and negative zeros and infinities. (Chapter 8 discusses NaNs, zeros, and infinities in greater detail.)

The nonzero finite values that must be representable in a given format are all those of the form $(-1)^s 2^E i$, where s is a finite integer, E is an integer between $E_{\min}$ and $E_{\max}$, both of which are format dependent, and i is a positive integer less than 2^p, where p is a positive integer that is also format dependent. The values of $E_{\min}$, $E_{\max}$, and p are specified for the basic formats; an upper bound for $E_{\min}$ and a lower bound for $E_{\max}$ and p are specified for the extended formats[1]. Table 3.1 lists the approximate range of representable finite magnitudes and decimal digits of accuracy for the various formats[2].

The width of the single format is 32 bits, and that of the double format is 64 bits. Encodings in both of these formats consist of three fields: the sign, the power of two, and the significand,

[1]The presentation here is slightly different (and hopefully slightly simpler) from that given in the IEEE Standard. In particular, instead of the integer i, the presentation in the Standard is based on a significand strictly between 0 and 2 consisting of at most p binary digits, with the bounds for E adjusted accordingly.

[2]The ranges in Table 3.1 do not take into account denormalized numbers, which allow smaller magnitudes to be represented at the expense of less precision. (Denormalized numbers are discussed in greater detail in Chapter 8.) If denormalized numbers were taken into account, the lower bound of the ranges would be extended approximately as follows: 10^{-45}, at least 10^{-317}, 10^{-324}, and at least 10^{-4951} for the single, single extended, double, and double extended formats, respectively.

Format	Approximate Range of Normalized Magnitudes	Approximate Decimal Digits of Accuracy	Binary Digits of Accuracy
Single	10^{-38} – 10^{38}	7	24
Single Extended	at least 10^{-308} – 10^{308}	at least 9	at least 32
Double	10^{-308} – 10^{308}	almost 16	53
Double Extended	at least 10^{-4932} – 10^{4932}	at least 19	at least 64

Table 3.1: Format ranges

with the most significant bit of the significand being implied (that is, not stored). The encodings are fully specified in such a way that, except for NaNs, every representable value in a given format corresponds to exactly one bit string, and vice-versa.

Special values (that is, infinities, NaNs, denormalized numbers, and zeros) are encoded by having the bit string in the exponent (that is, power of two) field be all ones or all zeros. (Using the notation above, a denormalized number is any number which can be written in the form $(-1)^s 2^E i$, and for which no value of E between $E_{\min}$ and $E_{\max}$ would allow the given number to be written in this form with $2^{p-1} \leq i < 2^p$. In the encodings for such numbers, the most significant [implied] bit of the significand is zero.)

Neither the width nor how values are to be encoded is specified for the extended formats, although Table 3.1 of the Standard seems to imply that encodings should not be more compact than is possible using only binary digits. (Of course, in practice, encodings are unlikely to consist of anything other than groups of binary digits.) Note that the constraints on both the double and double extended formats also fit the constraints on the single extended format, that is, both the double and double extended formats qualify as single extended formats.

3.2 Hardware facilities supporting data formats

The question of whether a given floating-point arithmetic engine supports the data formats described in the IEEE Standard demands in general a complex answer, particularly if one takes into account the operations the engine is capable of performing on values represented in these formats. Even without considering any operations which the arithmetic engine provides, it is not obvious what it means for an arithmetic engine to support a given data format. For the purposes of this chapter, however, the discussion will mostly be concerned with whether the given arithmetic engine is capable of moving an arbitrary value in one of these formats from primary memory to one of its internal registers with a single instruction, and then move (that is, store), again with a single instruction, that value to either a different internal register (assuming the engine has more than one internal register), or back to primary memory, without any change whatsoever to the value. That is, if the destination is another register, the value in that register must be an identical copy of that in the original register. If the destination is

primary memory, the value must be an identical copy of that in the original location in primary memory.

The floating-point units in implementations of the Intel x86 architecture support the single and double formats, as well as the double extended format. The width of the latter is 80 bits, and is subdivided into the same three fields as the other two formats. The format-dependent parameters corresponding to the double extended format barely fulfill the requirements of the Standard, and values are encoded similarly to how they are encoded for the other formats, except that all bits in the significand are stored explicitly. As explained in Chapter 8, this leads to some illegal encodings, which, if ignored, allow one to affirm that the double extended format is fully supported in the sense described above.

Support for these data formats is similar in the Motorola 68000 architecture, except that the width of the double extended format is 96 bits, since it includes an extra 16 bits of unused padding. Also, there are no illegal encodings in this format.

Most RISC architectures support the single and double formats. Some of them, such as DEC's Alpha and Motorola's PowerPC architectures do not support any other formats. Other RISC architectures, such as Sun's SPARC and HP's PA-RISC, define a 128-bit extended double format that is similar to the double format, except that the exponent is 15 bits wide and the significand occupies 112 bits. However, this format is mostly of academic interest, since implementations of these architectures do not include hardware support for operations on data in this format. Implementations of the Motorola 88000 architecture included hardware support for an 80-bit wide extended double format very similar to that of Intel's x86 extended double format.

3.3 Making data formats available in high-level languages

One of the most common ways of making a data format available in a high-level language is by means of a special data type, perhaps associated with a special keyword or identifier. Such a keyword or identifier provides a means to conveniently refer to values of this data format.

A less convenient, but still common, way of making a data format available is through a user-defined data type, which is a method one may be tempted to use in order to support a relatively uncommon data format, such as the double extended format, in an implementation of a high-level language that lacks support for such a format. Using such a method poses several challenges. Firstly, a data type must be defined (most likely using one or more predefined data types) in such a way that the amount of storage associated with objects of the type is not less than that required by the associated data format, regardless of which implementation of the language is used. Ideally, the amount of storage should be exactly that required by the associated data format, and objects of that type should be aligned strategically in primary memory so as to not impose an undue performance penalty when used.

Secondly, in order for performance to be tolerable, some sort of knowledge would need to be built into the language processor so maximum advantage of the features of the floating-point arithmetic engine can be taken. Even for operations as simple as assigning a value to a variable, the appropriate kind of registers should be used, for example. It would be very inefficient to,

say, keep all variables of a given type in memory, as some language processors do when dealing with structures or records, loading to and storing from floating-point registers when operating on such data, and using (possibly narrower) general purpose registers when copying values from one variable to another.

Since parameters whose type is user defined may sometimes be passed in memory rather than, say, in floating-point registers, there may be some issues related to the calling convention. For example, suppose one associates a record consisting of three 32-bit floating-point quantities with an extended double format that is 96 bits wide. If such records are passed in memory, one may want them aligned on a 128-bit address boundary in order to improve performance, which may be difficult to achieve—the calling convention may only guarantee 64-bit alignment. Alternatively, the calling convention could require passing such a record using one floating-point register for each field, or, if such records consisted of three 32-bit integers instead, the calling convention could require passing them using general purpose registers. In either case, some trickery may be required in order to force such a parameter to memory so arithmetic can be performed on it after loading it in a (single) floating-point register.

Chapter 7 has a more detailed discussion of how operations on floating-point values can be made available in high-level languages, but in order to avoid the overhead of calling ordinary functions when operating on data of a given type, special "intrinsic" or generic functions may be provided. A language processor could translate these intrinsic functions directly into assembly or machine language, possibly based on the data type of the argument or arguments.

In any case, resorting to user defined types is not as elegant as having a built in data type to specifically support a given data format.

A third way of making a data format available in a high-level language is to basically combine the two approaches above, and use a preprocessor to translate programs using special syntax into a form that more closely conforms to the constructs of a given language, or into a combination of high-level language constructs and (pseudo) assembly language. In an extreme case, one could even envision such a preprocessor making transformations to make use of the floating-point arithmetic engine in ways that are not apparent from glancing at the original program, in much the same way as a vectorizer program may transform, say, a Fortran program so as to make it easier for a Fortran compiler to recognize opportunities to vectorize a code.

Of course, such an approach would likely have the usual drawbacks associated with the use of any preprocessor: the process of translating from a high-level language to machine language is more complicated, as is debugging a program. However, this approach does not require one to modify every language processor one is interested in using, and avoids many problems inherent with creating yet another language whose sole reason for existence is to support a new data type.

If one were to develop a preprocessor and is not constrained by having to produce output that strictly conforms to a standard language definition or to a subset of a language that more than one language processor accepts, one could develop a preprocessor that takes advantage of the peculiar features of a given language processor. The given language processor might, for example, have partial support for the data format in which one is interested, perhaps including special intrinsic functions or the ability to specify what kind of register should be used to hold

a particular piece of data.

3.4 Associating floating-point literals with data formats

Many different ways can be devised to associate floating-point literals with particular data formats. One way is to implicitly associate floating-point literals with data formats, perhaps based on the magnitude of the value being represented, or how many digits make up the literal. Of course, using schemes like these would make programming more of an error prone task than it already is.

One could resort to calling an appropriate function in order to do the conversion from the external representation to the internal one, but then the literal would probably not be usable in places in which a constant expression is required. In addition, this approach adversely affects performance, and requires one to depend on the quality of the library containing such a function (or functions). However, unlike some other approaches, this would not likely require any changes to the definition of the language.

A very similar approach is to make conversion attributes available, the advantages being that conversion could take place at translation time, and no libraries would be involved.

Perhaps the most common way is to make the data format used depend on the suffix at the end of the floating-point literal. For example, the suffix d could be appended to a floating-point literal to indicate that it should be represented internally in (a format not narrower than) the double format. A disadvantage of this approach is that it can be tedious (and error prone) to change the format associated with each literal in a section of code, should one desire to do so. In addition, depending on how the language is designed, it may be difficult to write a program portably this way, since some implementations may not support certain data formats.

Another common way is to give literals names, and then associate these names with appropriate data types. This could make it easier to change the data format used to represent the literals in a program, especially if these literals are associated with a user-defined data type, since this would involve simply changing the definition of that data type.

A slight variation on this approach is to allow suitable prefixes, such as names of data types, to be prepended to floating-point literals. This would have the advantage of not requiring one to name every floating-point literal, though it could make it cumbersome to write code when a particular floating-point literal is needed in more than one place in a program.

Finally, a combination of the above approaches could be provided.

3.5 Other issues related to numeric literals

In programs, numeric literals are often represented in decimal notation. Some literals may be exactly representable in any data format, in which case it is not critical how that literal is actually stored internally. In general, however, a given value can only be approximated to varying degrees of accuracy, depending on what data format is used.

Given this, one may be tempted to store numeric literals using the widest data format available so as to minimize the loss of information in the conversion from the external to

the internal representation of a number. This could lead, however, to double rounding (on rare occasions) if, for example, the numeric literal needs to later be converted to a narrower format. That is, the actual value used in the computation could differ from the value that is representable in the narrower format and nearest to the one specified in the original program. (See Chapter 6 for more information on double rounding.) The programmer may have specified what data format is to be used to represent a given numeric literal, but if not, in general it is better to avoid converting numeric literals to their internal representations until it is clear in what format the associated value will actually be required to be represented during the course of a computation.

One easy mistake to make is to convert a given numeric literal to some data format only once, and then use that internal representation in every place in which that numeric literal appears, even if different data formats are appropriate in different places in the program. Another easily overlooked mistake is to keep track of fewer digits in a numeric literal than are required in order to be able to correctly round the specified value to the format or formats in which the literal is used.

3.6 Issues related to mixed-language programming

The most obvious issue related to mixed-language programming is whether data formats used when passing parameters between functions or procedures processed by different language processors, or to store the return value of a function match the expectations of the receiver of these values.

Also, if processed by different language processors, functions or procedures which access global variables need to make the same assumptions as far as what data formats are used for these variables.

A more subtle issue has to do with numeric literals, and can be illustrated with the following example. Suppose one were to compute some multiple of π, and subsequently pass this value to a library function whose source code was processed by a different language processor. Suppose that library function divides the value passed into it by π. Even if the numeric literal used in both functions to represent π were written identically, if this literal is represented internally using different formats, the quotient computed in the library function is unlikely to be an integer, thus possibly leading to unexpected results. In order to avoid such surprises, the same format needs to be used for such literals, even when processed by different language processors[3].

[3]Actually, this problem can occur even when the same language processor is used to process the entire program, since some language processors can use different data formats for different numeric literals, even if they are written identically.

3.7 What support exists in high-level languages

3.7.1 Traditional language designs

Many language definitions do not specify what data formats must be supported. Instead, language definitions are usually more concerned with the concept of data types. Discussion about data types may refer to sets of values that must be representable, but such discussion usually stops short of explaining how such values are to be encoded, or claiming to exhaustively enumerate the full set of values that an object of a given data type may have. An implementation is usually free to map data types to the data formats supported by the implementation, subject to a few minimal constraints.

The definition of the C language [16] is typical of the approach that many language definitions take. It defines three floating-point data types: float, double, and long double. There are few constraints on what values must be representable and how values are to be encoded, among them that the set of values that objects of type long double can have must include all those that objects of type double can have, which in turn must include all those that objects of type float can have. Thus, the same data format can be associated with all three of these data types.

Constraints on the standard header file float.h, which characterizes some salient features of the implementation's floating-point system, require values whose magnitude is between 10^{-37} and 10^{37} to be representable as normalized numbers. In addition, the precision of representable numbers must be such that one must be able to convert arbitrary 6- or 10-decimal-digit numbers (depending on whether one is referring to the float or double data type) to their internal representation, and then back again to decimal notation, with the result being the original number. Thus, if the radix for floating-point numbers were two, and the encoding were similar to that of the basic formats as specified in the IEEE Standard, the exponent range would have to at least include integers between -123 and 123, and the number of bits in the significand would have to be at least 21 and 35 for the float and double data types, respectively. The constraints on the long double data type are the same as for the double data type.

Fortran is slightly different in that implementations are strongly encouraged to use different data formats to encode values of type REAL and DOUBLE PRECISION, which are the only two floating-point data types that all implementations must support[4]. Fortran is otherwise similar to C, except that it provides a way for implementations to support any number of different floating-point data types[5].

Ada [11] also provides a way for implementations to support any number of different floating-point data types. Similarly to Fortran, one can select from among these, for example, by specifying how much precision is desired, possibly in combination with the desired range of

[4]Since the range (but not the precision) of these two data types can be the same, one could theoretically use the same data format, say, the double format, for both data types. On an implementation of the x86 architecture, the rounding precision mode could be altered depending on the precision corresponding to the data type of the result.

[5]Fortran 77 [31], an earlier version of Fortran, did not provide a way to support more than two floating-point data types.

representable numbers. Floating-point literals can be named or unnamed. They can be written in any base between two and sixteen. Unnamed floating-point literals are of type universal_real, whose set of values can be thought of as being the set of rational numbers. (Such values are typically converted to some other data type before being used in an expression.) Named floating-point literals can be associated with any floating-point data type. In addition, certain attributes can be used to associate a floating-point literal with a data type.

In C, C++, and Fortran, every floating-point literal is associated with some floating-point data type, depending on what the suffix, if any, at the end of the literal is. (In Fortran, the associated data type can also depend on what letter is used to separate the power of ten from the rest of the literal.) Conversely, for every predefined floating-point data type, there is a corresponding syntax for writing floating-point literals of that type.

3.7.2 Implementations of traditional languages

Although in the past this was not always the case, many language implementations now support the floating-point data formats implemented in hardware on the target processor. Thus, implementations targeting SPARC and PA-RISC processors tend to support the single and double formats, but not either of the extended formats, even though both of these architectures define a double extended format. On the other hand, implementations targeting Intel's x86 architecture usually support the single and double formats, as well as the double extended format, provided the language definition defines more than two floating-point data types. For example, Fortran 77 compilers, such as Microway's NDP Fortran compiler, may lack support for the double extended format. SunSoft's Fortran compiler only supports the double extended format when targeting SPARC processors; surprisingly, their C compilers support the double extended format, regardless of whether the target processor is an implementation of the SPARC, PowerPC, PA-RISC, or x86 architecture. Presumably, on RISC processors, software is used to perform arithmetic involving the double extended format.

3.7.3 Extensions to traditional languages

The main advantage that extensions to traditional languages typically offer in the area of data formats is standardization as to how one can refer to the data formats described in the IEEE Standard. For example, implementations of the Standard Apple Numerics Environment (SANE) support the single, double, and double extended formats, adding predefined data types as necessary for this purpose. The SANE does not support the single extended format. Floating-point literals are always converted to the double extended format in SANE Pascal and C.

The extensions to the C language described in [75] are somewhat less stringent in that, while support for the single and double formats is required, support for the double extended format is optional. Thus, implementations must provide access to at least two distinct data formats. In addition, there are aliases for the most efficient floating-point formats that are at least as wide, respectively, as the single and double formats. This set of extensions does not make any effort to provide access to the single extended format. Floating-point literals are allowed to

be represented internally in a format wider than that associated with their corresponding data types. Floating-point literals can optionally be written in hexadecimal notation.

3.7.4 New language designs

μln [30] provides three floating-point data types: one is associated with the single format, another with the widest format that the target processor supports. The third data type is associated with the double format if the target processor supports it; otherwise, this data type is associated with a format whose width is (strictly, when possible) between that of the other two data types. Floating-point literals are all of type decimal, which is a special composite data type consisting of two integers: one containing the significand, and another the power of ten. Unnamed floating-point literals are converted (possibly at translation time) to the widest supported data format before use; named floating-point literals are converted to the data format corresponding to the precision mode in effect. (Chapter 5 discusses precision modes.) Unnamed floating-point literals could suffer double rounding in certain cases.

Some languages support only one or two floating-point data types (or formats), even if the processor actually executing programs written in those languages supports two or three data formats in hardware. For example, Limbo [59] supports only the double format, and thus merits no more comment other than to remark that it fails to meet the IEEE Standard's requirement that the single format be supported. Java, on the other hand, supports both basic formats, but none of the extended formats. Every floating-point literal in Java is associated with one of these two formats, depending on how it is written.

3.8 What support should exist in high-level languages

In order for there to be some standardized means of referring to the data formats specified in the IEEE Standard, a language (or implementation thereof) intended to be used for numerical programming ought to make provision for at least three distinct floating-point data types. Two of these should correspond to the two basic formats, and one of them should correspond to the double extended format, and not be required to be implemented. Some accommodation ought to be provided for implementers wishing to support the single extended format, though in practice, support for this format is mainly of interest when targeting certain digital signal processors, some of which support the single extended format in hardware. If an implementation supports a single extended format that is distinct from the double format, the values that are representable in the single extended format should be required to be a subset of those that are representable in the double format. Implementations ought to be required to reject programs which make reference to data types they do not support.

Support for the single format ought to, of course, be mandatory. Although the IEEE Standard does not require support for any of the other formats, many implementations of the IEEE Standard do support at least one other data format. Few implementations, if any, support all four data formats; one or both of the extended formats are usually the ones lacking support. Thus, it is not critical for a language or implementation thereof to make provision for more

than three distinct data types.

In addition to these distinct data types, at least two aliases ought to be provided: one corresponding to the widest format supported (in hardware) by the implementation, and one corresponding to the most efficient format for the target processor, perhaps with a bias toward the widest format among those that are equally efficient, in order to benefit programmers who underestimate their need for precision [55]. In fact, in implementations which do not support a single extended format that is distinct from (and narrower than) the double format, the data type corresponding to the single extended format could also be an alias, in this case for the double format.

All these data types and aliases would allow one to precisely specify what format should be used to compute some value. For example, if some algorithm specifically required the use of a particular data format, there would be a way to specify this. On the other hand, if the final result of a computation were expressed in some basic format, one may wish to compute all the intermediate results in slightly greater precision in order to improve the accuracy of the final result, as well as the robustness of the algorithm. This could also be accommodated: The data type corresponding to the single extended format could be used for intermediate calculations when the final result is required in the single format. (The double format could actually be used instead in implementations that do not support a distinct single extended format.) On the other hand, if the final result is required in the double format, intermediate calculations can be performed using the widest format supported in hardware, which might be the double or double extended format, depending on the target processor. (If an algorithm actually requires the double extended format to be used, then the data type corresponding to this format should be specified, and implementations which do not support this data type should reject such code.) Finally, if it really does not matter what format is used for a particular computation, or if speed is of the essence, then the data type corresponding to the most efficient format for the target processor could be specified. Note that this format is the most efficient format at least as wide as the single format.

If one wishes to use the most efficient format at least as wide as the double format, one should distinguish between "temporary" variables which are likely to be held in registers, and variables, such as arrays, whose use is likely to cause traffic on the processor's memory bus. In the former case, using the widest format supported in hardware (which is likely to be the double or double extended format) is appropriate. In the latter case, the double format is the better choice, since this may reduce the traffic on the processor's memory bus, and is likely to require less time for operations such as multiplication and square root.

It ought to be possible to determine which data format is actually the widest supported in hardware, or at least whether the widest data format supported in hardware is wider than some other data format, such as the single extended format or the double format. This would allow one to decide which algorithm to use based on which formats are supported in hardware.

As for floating-point literals, it ought to be possible to associate any of them with any of the formats described in the IEEE Standard, as well as with the widest data type supported in hardware. In addition, since some implementations may not support all formats, it ought to be possible to not specify what format should be used to represent a given numeric literal, but

rather leave it up to the language processor to decide what format to use, depending, perhaps, on the context in which the literal appears. (Such literals would roughly correspond to the "efficient" data type mentioned above, except that different data formats might actually be used in different places, even if the literals are written identically.) The actual data format used could depend, for example, on the data types of nearby operands. Thus, if the nearby operands are all associated with the single format, the language processor could use the single format to represent the literal. On the other hand, if at least some of the nearby operands are associated with the double format, the language processor might instead use the double format to represent the literal. The conversion from the external (decimal) representation to the internal (binary) representation should take place after the language processor decides which format to use to represent the literal in order to avoid the possibility of double rounding.

If one were to assume that no implementation of a given language will support all four data formats described in the IEEE Standard, one could provide four ways of specifying floating-point literals: one associated with the single format; one associated with the single extended format if supported in hardware, and otherwise associated with the widest format supported in hardware that is not wider than the double format; one associated with the widest format supported by the implementation, even if such support is only in software; and one in which the processor is allowed to pick which data format to use. Note that this scheme provides ways to conveniently ensure that if literals are written identically to each other, their internal representations will also be identical, since the only way this would not necessarily be the case is if the literal is written using special syntax to indicate that the language processor should choose what format to use on a case-by-case basis.

Chapter 4

Supporting Rounding Modes in High-Level Languages

4.1 IEEE Standard requirements in regards to rounding modes

The IEEE Standard defines four different rounding modes: round to nearest, round toward zero, round up (toward positive infinity), and round down (toward negative infinity). The user must be able to select which of these rounding modes is in effect for any of the following operations:

- addition
- subtraction
- multiplication
- division
- square root
- conversion to a narrower floating-point format
- conversion between a floating-point format and an integer format
- rounding a floating-point number to an integer value, and
- conversion between a decimal string and a floating-point number.

Except for the latter, whose accuracy requirements are somewhat weaker in some cases, the results of these operations are first computed as if to infinite precision, and then rounded according to the rounding mode in effect to fit the destination format[1]. (The remainder operation

[1]Actually, in certain arithmetic engines, the results of some of these operations may be rounded according to the rounding precision mode in effect, rather than the destination format. See chapter 5.

Processor	Clock cycles to transfer CR to GPR	Clock cycles to isolate bits	Total clock cycles
Intel 486	(fstcw, mov) 4	(and) 1	5
Intel Pentium	(fstcw, mov) 3	(and) 1	4
IBM PowerPC 601	(mcrfs, mfcr) 3	(andi) 1	4

Table 4.1: Number of cycles required to get the current rounding mode

does not appear in the list above, since the result of the remainder operation, as defined, is always exact and independent of the rounding mode.)

The default rounding mode is round to nearest. When this mode is in effect, and an infinitely precise (intermediate) result is equally near to two adjacent representable numbers, the infinitely precise result is rounded to the number whose least significant (significand) bit is zero.

The rounding mode in effect must be settable dynamically, since according to section 2 ("Definitions") of the IEEE Standard [47], a *mode* is "a variable that a user may set, sense, save, and restore. . . ." This may come as a surprise to some, since nowhere else does the IEEE Standard unequivocably require the rounding mode to be settable dynamically. For example, section 4 ("Rounding") merely describes how the rounding mode in effect affects the result of an operation, but says nothing about whether it might be possible for the rounding mode in effect to be unknown at compile time at a given point in a program.

4.2 Hardware facilities for accessing rounding modes

4.2.1 Processors with dynamic rounding modes

Most floating-point architectures have a special control register to which the user has access in order to, among other things, find out what rounding mode is currently in effect, and to change it if so desired. Modifying this register is usually the only means to control what rounding mode will be used when computing the result of floating-point operations.

The sequence of instructions to find out what the current rounding mode is typically involves transferring the contents of the control register to a general-purpose register (possibly by way of memory) and isolating the relevant bits. Table 4.1 shows how many clock cycles are required to accomplish this on several popular processors.

The clock counts in Table 4.1 and in other tables in this section are taken from the manufacturers' literature, and assume best-case conditions[2]. In particular, all instructions and instruction operands are assumed to be in the internal processor cache, and any needed memory addresses are assumed to be already available in appropriate registers or as immediate

[2]It is difficult to measure the exact number of clock cycles required to execute these instruction sequences, since several factors may affect the timing. For example, in some cases, the timing may be dependent on the bandwidth between the processor and the external cache or memory.

Processor	Clock cycles to retrieve CR	Clock cycles to isolate bits and store to memory	Total clock cycles
Intel 486	(fstcw) 3	(and) 3	6
Intel Pentium	(fstcw) 2	(and) 1	3
IBM PowerPC 601	(mcrfs, mfcr) 3	(andi, stw) 2	5

Table 4.2: Number of cycles required to save the current rounding mode

Processor	Clock cycles to retrieve CR	Clock cycles to modify bits	Clock cycles to set CR	Total clock cycles
Intel 486	(fstcw, mov) 4	(and, or) 2	(mov, fldcw) 5	11
Intel Pentium	(fstcw, mov) 3	(and, or) 2	(mov, fldcw) 8	13
IBM PowerPC 601	N/A	N/A	(mtfsb) 8	8

Table 4.3: Number of cycles required to set the rounding mode

constants.

The sequence of instructions to save (only) the rounding mode in some memory location is very similar in principle. However, instead of converting the representation of the rounding mode to an integer between 0 and 3, (only) the bits representing the rounding mode are simply saved in a memory location. Table 4.2 shows how many clock cycles are required to accomplish this on several popular processors.

The sequence of instructions to set the rounding mode to one that is statically known (at compile time, for example) typically involves transferring the contents of the control register to a general-purpose register or to some memory location, modifying the appropriate bits, and transferring the modified value back to the control register. Table 4.3 shows how many clock cycles are required to accomplish this on several popular processors.

As before, the clock counts in Table 4.3 assumes best-case conditions. In particular, the desired rounding mode is assumed to be representable as an immediate constant, that is, a constant forming part of the instruction stream. (The sequence of instructions on the PowerPC 601 is quite different, since this processor has a special instruction called *mtfsb*, which modifies a single specified bit in the control register. This instruction can be used twice in order to set the desired rounding mode.)

Finally, the sequence of instructions to restore a previously-saved rounding mode, or to set the rounding mode to one that is not known statically, is similar in principle to that of simply setting the rounding mode as described in the paragraph above, except that the desired rounding mode can no longer be represented as an immediate constant. Table 4.4 shows how many clock cycles are required to accomplish this on several popular processors.

In order to put these clock counts into perspective, Table 4.5 shows the maximum number of pairs of additions and multiplications these processors can issue in the amount of time it

Processor	Clock cycles to retrieve CR	Clock cycles to modify bits	Clock cycles to set CR	Total clock cycles
Intel 486	(fstcw, mov) 4	(and, or) 3	(mov, fldcw) 5	12
Intel Pentium	(fstcw, mov) 3	(and, or) 3	(mov, fldcw) 8	14
IBM PowerPC 601	(mcrfs, mfcr) 3	(andi, lwz, or) 3	(stw,lfd,mtfsf) 8	14

Table 4.4: Number of cycles required to restore a previously saved rounding mode

Processor	Get RM	Save RM	Set RM	Restore RM
Intel 8086/8087	0.19	0.32	0.42	0.51
Intel 80286/80287XL	0.47	0.45	1.2	1.3
Intel 386DX/387DX	0.72	0.76	1.5	1.7
Intel 486	0.26	0.32	0.58	0.63
Intel Pentium	2	1.5	6.5	7
IBM PowerPC 601	4	5	8	14
IBM PowerPC 604	7	8	6	14
DEC Alpha 21164	10	11	20	20

Table 4.5: Number of pairs of additions and multiplications that can be issued in the same amount of time it takes to manipulate the rounding mode

takes to access or modify the rounding mode[3].

As can be seen from Table 4.5, the overhead of accessing or modifying the rounding mode becomes more significant as the (maximum) floating-point performance of a processor increases, especially considering that the current rounding mode will often need to be saved before it is modified.

4.2.2 Processors with static rounding modes

The INMOS IMS T800 is one of a very few processors which does not have dynamic rounding modes implemented in hardware. Instead, each floating-point operation which needs to be performed using a rounding mode other than round to nearest must be preceded by a special instruction which specifies the rounding mode to be used. The overhead of this special instruction is 2 clock cycles, as compared to 6–9 clock cycles for an addition and 11–27 clock cycles for a multiplication. However, if the language requires rounding modes to be dynamic, they will have to be emulated in software. This means that any time the compiler is not able to determine what rounding mode should be used for a particular floating-point operation, the compiler will have to generate code which tests a variable to determine what rounding mode is

[3]In the case of the PowerPC 601 [7], the information in Table 4.5 is relative to the number of pairs of single precision additions and multiplications it can issue, which is twice the number of double precision additions and multiplications it can issued. In the case of the other processors, the information in this table is relative to the number of pairs of double or extended precision additions and multiplications these processors can issue.

in effect, and then executes the correct code sequence based on the result of this test. If one assumes that round to nearest will be by far the most common rounding mode used, the overhead when this mode is in effect would be a minimum of 5 clock cycles in addition to the clock cycles required to perform the floating-point operation itself[4]. The overhead when a different rounding mode is in effect would be at least 12 clock cycles. This overhead, while significant, is not overwhelming. However, if the floating-point performance of this chip were as good as that of some of the more recent processors, this overhead would be intolerable.

The DEC Alpha architecture [2] also defines static rounding modes in addition to dynamic rounding modes. In this architecture, the rounding mode to be used is part of the encoding for floating-point instructions. Unfortunately, current implementations of this architecture do not have this feature implemented in hardware. As a result, if one were to make use of this capability, floating-point performance would slow down quite significantly [57]. However, if this feature were implemented in hardware, the compiler could issue instructions with the appropriate rounding mode encoded in the opcode in cases in which it knows what the current rounding mode in effect is, and use the dynamic rounding mode feature otherwise. In particular, the ability to use static rounding modes could make it trivial, for example, to guarantee that a certain section of code will be evaluated using a certain rounding mode (perhaps because that section of code is not designed to work if a different rounding mode is in effect), regardless of what rounding mode is in effect in other parts of the program. Also, given how long it usually takes to change the rounding mode in effect, the use of the static rounding mode feature could significantly speed up a section of code in which a particular rounding mode is to be used when executing one or a few specific operations.

4.3 Making rounding modes available in high-level languages

Facilities available to the high-level language programmer for setting rounding modes can be classified according to the granularity over which the user can control the mode in effect, as measured by the amount of source code or the amount of run time. The section of source code over which one request to use a particular rounding mode applies might be restricted to:

- a single arithmetic operation,
- a subexpression,
- a sequence of one or more statements,

[4]It is possible to reduce this overhead to near zero clock cycles by performing the operation using the round-to-nearest mode, and concurrently checking to see if the mode in effect is indeed round to nearest. If not, a complicated sequence would have to be executed to examine the instruction stream prior to the most recent floating-point operation in order to determine what the operands were, and then re-execute the operation using the correct rounding mode. So whenever the rounding mode were not round to nearest, floating-point performance would be severely impacted.

Another way to reduce the overhead is to check what rounding mode is in effect once per basic block, and to have the compiler generate several variations of each basic block—one for each rounding mode. This scheme would be particularly effective for basic blocks in which floating-point operations predominate.

- a scope,
- a portion of a file,
- an entire file, or
- an entire program.

Ways of setting rounding modes can also be characterized by whether the rounding mode specified affects the rounding mode in effect for the caller or callee of a function or procedure, that is, the granularity over which the user can control the rounding mode in effect, as measured by the amount of run time. Whether the rounding mode specified affects the rounding mode in effect for the caller of a function or procedure is not necessarily related to whether the rounding mode specified affects the rounding mode in effect for the function or procedure called. There are several possibilities:

- A function or procedure cannot directly affect the rounding mode in effect of its caller.
- A function or procedure can directly affect the rounding mode in effect of its caller, but not that of the caller of its caller. This might happen, for example, if the caller's rounding mode is normally restored upon return, except when "special" functions are called.
- A function or procedure can directly affect the rounding mode in effect of its caller, and that of any function or procedure further up in its call chain. This might happen, for example, if the rounding mode in effect were implemented as a global variable.
- The caller of a function or procedure cannot directly affect the rounding mode in effect of the function or procedure it calls.
- The caller of a function or procedure can directly affect the rounding mode in effect of the function or procedure it calls, but not that of any function or procedure further down its call chain. This might happen, for example, if functions and procedures do not normally inherit the rounding mode from their callers, unless a special mechanism, such as special syntax, is used. For example, suppose that in a certain fictitious language, the syntax for ordinary function calls were:

  $$function\_name(parameter_1, parameter_2, \ldots, parameter_n)$$

  The rounding mode to be used in the function called might be specified using the following syntax:

  $$function\_name(parameter_1, parameter_2, \ldots, parameter_n : rounding\_mode)$$

- The caller of a function or procedure can directly affect the rounding mode in effect of the function or procedure it calls, and that of any function or procedure further down its call chain. This might happen, for example, if the rounding mode in effect were implemented as a global variable.

In addition, the availability of facilities for setting rounding modes may cause the rounding mode in effect to not be known at compile time at any point in a program, to be known at compile time at every point in a program, or to be known at compile time in some places, but not in others. Of course, regardless of whether the rounding mode in effect is actually known at a given point in a program, it is possible for the compiler or programmer to assume that a particular rounding mode is in effect at that point.

4.4 Handling static evaluation and numeric literals

Normally, when a (named or unnamed) numeric literal appears in the context of an expression, its representation as a binary value has to be determined. This process of determining the proper binary representation of a numeric literal is called evaluation.

Similarly, there are cases in which a compiler can feasibly or indeed must be able to find out what the binary representation of the result of an expression is. This process of determining the proper binary representation of the result of an expression at compile time is called static evaluation. In this section, such an expression is said to be static. Also, in this section, the term floating-point static expression refers to a static expression involving floating-point operations, or to a numeric literal whose representation as a binary floating-point value must be determined.

If the rounding mode (and precision mode—see chapter 5) is known at compile time at a point in a program in which a floating-point static expression appears, it may be possible to safely evaluate that expression at compile time[5]. If the rounding mode is not known at compile time at a point in a program in which a floating-point static expression appears, the compiler can either pick a particular rounding mode and evaluate the expression using this mode as if at compile time[6], or cause the evaluation to occur at run time using the rounding mode in effect at that point in the program.

Hopefully, evaluation of a program's floating-point static expressions will be consistent. That is, the programmer should hopefully be able to determine whether an expression (static or otherwise) will be evaluated as if at compile time or at run time, and whether the rounding mode in effect at that point in the program or some other particular rounding mode will be used, just by examining the program in which the expression appears.

A language definition will often specify certain cases in which the compiler must statically evaluate an expression. If that expression involves floating-point operations, then it is reasonable for the compiler to pick a rounding mode, possibly different from the one in effect at that point in the program, to use during evaluation of that expression. The rounding mode the compiler should pick for use during the evaluation of such expressions depends on several factors, such as:

[5] As explained below, whether a floating-point expression can be evaluated at compile time depends in part on the semantics of floating-point exceptions. Of course, if a language definition requires the expression to be evaluated at compile time, whether or not an exception might occur during its evaluation does not change the fact that the expression has to be evaluated at compile time.

[6] Whether evaluation actually occurs at compile time or at run time is an implementation issue.

- whether the language definition specifies the rounding mode to be used (this is very unlikely);
- whether the rounding mode in effect at that point in the program happens to be known at compile time; and
- the ease with which a human can determine, just by examining the program, what rounding mode the compiler will use.

Because of the latter, it would probably be best if the compiler were to consistently pick the same rounding mode in all such cases, provided this does not conflict with the language definition.

If a compiler can feasibly statically evaluate a floating-point expression which the language definition forbids the compiler to evaluate at compile time, the compiler could still conceivably evaluate the expression at compile time, provided the rounding mode is known at compile time, and the generated code is able to fully preserve the semantics of each floating-point operation in the expression. In general, the compiler will only be able to do this in very few cases, since a side effect of most floating-point operations is to raise one or more exceptions, the most common of which is the inexact exception (see chapter 8). Therefore, unless the compiler happens to know that no exceptions would occur during the evaluation of the expression, or that no trap handlers are enabled for whatever exceptions would occur and the status flags corresponding to these exceptions are guaranteed to already be set at that point in the program, the compiler will have to determine whether it would be faster to explicitly raise the exceptions that would occur if the expression were evaluated at run time, or to simply evaluate the expression at run time. Furthermore, if any exceptions are to be raised at run time, the compiler will have to determine whether raising these exceptions explicitly would confuse any trap handlers which may be enabled, for example, whether any enabled trap handlers for these exceptions are able to determine what operation caused these exceptions to be raised.

If a compiler can feasibly statically evaluate a floating-point expression (or subexpression) which the language definition does not require, but does not forbid, the compiler to evaluate at compile time, the compiler will need to pick a rounding mode to use during the evaluation of this expression if it chooses to statically evaluate the expression. This rounding mode could either be:

- the rounding mode that would have been used if the language definition had required the compiler to statically evaluate the expression;
- the rounding mode that would be used if the expression were evaluated at run time; or
- a rounding mode that happens to be convenient.

It is also possible to evaluate part of the expression as if at compile time, and the remainder of the expression at run time, so that different rounding modes could potentially be used for different parts of the expression, confusing as this may be. (It is not so uncommon to encounter situations in which it might be desirable to use potentially different rounding modes for different parts of an expression. For example, consider the expression $3.1415926 \times 2.7182818$.

The compiler could arrange for the numeric literals to be evaluated as if at compile time using a different rounding mode than that for the multiplication, which might not take place until run time.) Perhaps the overriding concern here in the choice of a rounding mode should be the ease with which a human reader can determine, just by examining the program, which rounding mode will be used.

4.5 Issues related to mixed-language programming

As far as rounding modes are concerned, the main difficulty in mixed-language programming is making sure there is no conflict in the way rounding modes are handled in the various languages and compilers involved. For example, one language or compiler may assume that when a function or procedure is called, the rounding mode in effect will remain the same upon returning from the function or procedure as it was before the function or procedure was called, that is, that the callee cannot affect the rounding mode in effect in the caller. A different language or compiler may make the opposite assumption: it may assume that the caller knows that the rounding mode in effect may change upon returning from a function or procedure, and that there is therefore no need to be sure to restore the rounding mode in effect before the function or procedure was called.

There are many kinds of possible conflicts related to how different languages and compilers handle rounding modes. If there is some way for the compiler to determine that a foreign function or procedure (that is, one written in some other language or processed by some other compiler) is about to be called, then the compiler can take some measures to guard against these conflicts. For example, if a compiler purports to know what rounding mode is in effect at every point in a program, upon returning from a foreign function or procedure, the compiler will quite likely need to restore the rounding mode that was in effect before calling that foreign function or procedure. On the other hand, if a compiler does not care to know what rounding mode is in effect at any point in a program, and procedures and functions are allowed to modify the rounding mode in effect in their callers, then, as far as rounding modes are concerned, the compiler need not take any special care in processing calls to foreign procedures and functions.

Another potential problem has to do with the startup code that is executed just before execution of the program itself begins. Different languages and compilers may have different conventions on what rounding mode should be in effect when the program itself begins execution. Other languages or compilers may not even initialize the floating-point unit at all in cases where such initialization is required! (A compiler might, for example, initialize the floating-point unit only if the program performs floating-point operations. But the only code in a program that involves floating-point operations might be processed by a different compiler, so the other compiler may mistakenly reach the conclusion that the program does not perform floating-point operations at all.) A conflict in the convention on what rounding mode should be in effect when the program itself begins execution could cause a program to behave incorrectly.

4.6 What support exists in high-level languages

4.6.1 Existing language designs

Among the very few language defining documents that mention anything at all about rounding modes is the Ada 95 Reference Manual [11], and then only indirectly. Ada implementations may optionally conform to the accuracy requirements in Appendix G of [11]. Basically, these accuracy requirements say that for predefined operations, which certainly include addition, subtraction, and multiplication, the error of the result must be less than one ulp (unit in the last place). (Division is allowed to be less accurate, since on some computers, division is performed by computing the reciprocal of the divisor, followed by multiplication by that reciprocal.) This implies that any rounding method which can guarantee this much accuracy is acceptable. In particular, any of the four rounding methods defined in the IEEE Standard are acceptable.

In addition to these accuracy requirements, [11] requires all implementations to provide several functions that round floating-point numbers to integers. The most common of these is described in section 4.6 ("Type Conversions"): "If the target type is an integer type and the operand type is real, the result is rounded to the nearest integer (away from zero if exactly halfway between two integers)." This rounding method is similar, but unfortunately not identical, to the IEEE Standard's "round to nearest or even" rounding mode. In fact, it would most likely not be possible to implement this function very efficiently on a machine conforming to the IEEE Standard: one would most likely have to add one half to or subtract one half from the floating-point number to be converted (depending on the sign of that number), and then truncate the sum or difference to an integer. Other functions available for this purpose, such as Unbiased_Rounding and Truncation described in section A.5.3 of [11], do coincide with the IEEE Standard's rounding modes.

4.6.2 Compilers for existing languages

Among the many compilers for existing languages, the IBM C/C++ compiler for OS/2 is very typical in its support for rounding modes. The compiler itself never generates code to change the rounding mode in effect. Furthermore, the compiler assumes the rounding mode in effect does not change across function calls, and that if the programmer changes the rounding mode, the original rounding mode is restored before calling any functions. A function in a library shipped with the compiler retrieves the current rounding mode and optionally changes it. Another function resets the floating-point unit (and presumably the rounding mode) to the compiler's default state. There is no documentation on what this default state is, or on which floating-point (sub)expressions, if any, are evaluated at compile time.

4.6.3 Extensions to existing languages

In the Standard Apple Numeric Environment (SANE), access to the rounding mode in effect is provided via two high-level language procedures: one to determine what rounding mode is currently in effect, and one to set it. There is no provision for saving or restoring only

the rounding mode, though of course the programmer can write a routine to do this. When a program begins execution, the rounding mode in effect is "round to nearest." When the rounding mode is set, the new rounding mode remains in effect until explicitly changed via another call to the procedure provided for this purpose[7]. Thus, the facilities SANE provides may be characterized in terms of the granularity of run time over which the user can control the rounding mode in effect.

In general, if the compiler does not perform any interprocedural analysis, the rounding mode in effect cannot be known at compile time, except from the beginning of the program until the first function or procedure is called. All numeric literals are evaluated at compile time using the round-to-nearest rounding mode, as are floating-point expressions denoting the values of named constants in the case of Pascal, or the initial values of static and external variables in the case of C. All other floating-point expressions are evaluated at run time using whatever rounding mode is in effect at that time.

In terms of support for rounding modes, the NCEG's extensions [75] are similar to those of SANE. The main difference from SANE is that the compiler is explicitly given permission to assume the rounding mode in effect is round to nearest at any point in a program, unless a point in the program is under the effect of a pragma called fenv_access. This pragma does not modify the rounding mode in effect. The scope of this pragma is determined statically, beginning at the point in which this pragma is turned on, and ending when this pragma is encountered in the source code again or at the end of the file, whichever is encountered first. In the programming model [75] assumes, functions do not modify the rounding mode in effect of their callers, unless a function's documentation says otherwise. This model is not enforceable by the compiler.

The other area in which the NCEG's extensions differ from SANE is that floating-point expressions denoting initial values of objects of some aggregate or union type are evaluated as if at compile time, rather than at run time.

4.6.4 New language designs

μln [30] takes a completely different approach to supporting rounding modes. The language does not provide any functions which the programmer can call to change the rounding mode in effect. Instead, what rounding mode is in effect at a given point in a program is related to the block structure of the program. Conceptually, whenever a new block is entered, such as when a function or compound statement begins execution, a new floating-point environment is established. The rounding mode in that new environment is inherited from that of the enclosing environment, if any, unless the programmer explicitly specifies a rounding mode for that block. In addition, the caller of a function can override the rounding mode in the outermost environment of that function, which, in the absence of any explicit indication, would be the default rounding mode ("round to nearest").

[7]It is also possible to write an assembly language procedure, which, when called, possibly by a program written in a high-level language, will change the rounding mode. It is unlikely that such a procedure would be hardware-independent, though, but the effect of calling such a procedure would be similar to that of calling the high-level language procedure already provided.

μln also provides special unary operators which specify the rounding mode to be used during the evaluation of their operands, which may be arbitrary subexpressions. This facility may be used for subexpressions embedded within constant expressions.

Whenever the language requires the value of a floating-point expression to be known at compile time, the rounding mode used during the evaluation of that expression is round to nearest. All other floating-point expressions are evaluated using whatever the current rounding mode happens to be. If the compiler is able to determine at compile time what this mode will be, it may be able to evaluate at least part of the expression at compile time.

Thus, given these two facilities, the programmer has control over what rounding mode will be used, either at the level of a subexpression or a block at a time. In addition, functions can specify what rounding mode should be in effect in the functions they call, but not that of any function further down its call chain, or that of its caller.

4.7 Why having more than one rounding mode available is useful

Before actually discussing what kind of support compilers and high-level languages ought to provide for rounding modes, it is important to note why one might want to have more than one rounding mode available. The four main uses of rounding modes are:

1. *Interval arithmetic*, that is, arithmetic on infinite sets of real numbers whose upper and lower bounds are floating-point numbers. Although the result of an arithmetic operation is seldom a single point, interval arithmetic is attractive because it provides a way to find an interval in which the result of a computation is guaranteed to lie.

2. *Common mathematical functions.* Different rounding modes may be useful for implementing certain common mathematical functions, such as $\lfloor x \rfloor$ (floor) and $\lceil x \rceil$ (ceiling).

3. *Disproving the stability of an algorithm.* Sometimes it is difficult or time consuming to prove that an algorithm is numerically stable, that is, that small perturbations in the input will not change the final result very much, or that the final result actually obtained is the correct result for a slightly different input. In such cases, it may be attractive to run through the algorithm using different rounding modes to see if the final result changes very much. If the final result changes markedly when different rounding modes are used, it is quite likely that the algorithm is not numerically stable, and that its usefulness is dubious.

4. *Emulating other floating-point arithmetic engines.* In some cases, one may want to obtain results similar to those obtained on some other computer, whose floating-point arithmetic engine may round results differently than, say, the IEEE Standard's round to nearest. Although the IEEE Standard might not define a rounding mode identical to the desired rounding method, it may still be possible to implement this rounding method by a series of operations involving the use of one or more rounding modes defined in the IEEE Standard.

4.8 What support should exist in high-level languages

In three of the four cases described in the preceding section, ideal support for rounding modes would include the ability to specify the rounding mode to be used for one specific floating-point operation when the operand or operands are scalars, and the ability to specify the rounding mode to be used during the execution of one or more statements when the operand or operands are vectors. In addition, some means of propagating the specified rounding mode to functions and procedures called would be required.

Of the four cases described in the preceding section, the third case, that of disproving the stability of an algorithm, requires a different kind of language facility. In this case, one should ideally be able to control the rounding mode in effect over the portion of the file or files that contains the code for the algorithm being tested. In most such cases, it would probably be acceptable to have a means to specify the rounding mode to be used on a file-by-file basis. In any case, it would be advantageous to be able to determine what rounding mode a program is using by examining the source code that was compiled.

Another important consideration is whether the compiler can determine what rounding mode is in effect at a given point in a program. Since the IEEE Standard requires the rounding mode to be settable dynamically, those who drafted the standard surely intended that this capability be available to high-level language programmers. However, if a language has this feature, the compiler will in general not know what rounding mode is in effect at an arbitrary point in a program. Consequently, the compiler will most likely only be able make few, if any, improvements in the code generated for floating-point expression evaluation.

One way to increase the likelihood that the compiler will know what rounding mode is in effect at an arbitrary point in a program is for the compiler to enforce, or at least assume, that, unless the programmer specifies otherwise, the rounding mode in effect when a function or procedure begins execution is the default rounding mode, and that upon returning from a function or procedure, the rounding mode in effect is the mode that was in effect before that function or procedure was called. Additionally, the compiler should know if there is any possibility that the rounding mode in effect at a given point in a program is not the default rounding mode. For example, if there is some way for the caller of a function or procedure to specify what rounding mode should be in effect during the execution of that function or procedure, the compiler should know if that function or procedure will ever be called with a rounding mode other than the default rounding mode.

If the compiler is to enforce such a convention, it is important to consider the overhead involved, which mainly occurs when calling or returning from functions and procedures. In a naïve compiler, the code generated at the beginning of every function or procedure would save the current rounding mode and set it to the default rounding mode, while the code generated when the function or procedure returns to its caller would restore the rounding mode previously saved.

A more efficient way of enforcing this convention is for the caller of a function or procedure to save its rounding mode whenever the compiler has any doubts as to whether the current rounding mode is the default rounding mode. (If the compiler happens to know what the

current rounding mode is, the compiler does not need to generate code to save the current rounding mode.) The caller can then set the rounding mode to the default rounding mode just before calling the function or procedure, and restore the previous rounding mode upon returning from the callee. Since in most programs, the rounding mode changes only rarely if at all, there would usually be little overhead in using this scheme.

Unfortunately, this scheme is not very effective in the context of mixed-language programming, since the compiler will no longer be able to guarantee what rounding mode will be in effect upon returning from a function or procedure written in different language, or when a function or procedure written in a different language calls a function or procedure the compiler is processing. In such cases, the compiler may have to resort to the naïve scheme described above.

Finally, if it is necessary to save the result of some operation in some memory location, it may be important to make sure that the value in memory is the same as the value in the register. This is particularly relevant when a processor's registers have greater precision and/or range than the format used to store values in memory. In such processors, when a value is transferred from a register to some memory location, the value in the register may have to be rounded in order to fit its destination. However, the rounding mode in effect may be different than when the operation which produced the result was performed. The value stored in memory could therefore be very different from the intended result.

Chapter 5

Supporting Precision Modes in High-Level Languages

5.1 IEEE Standard requirements in regards to precision modes

In practice, precision modes are only relevant to floating-point arithmetic engines which do not deliver results of floating-point operations (presumably such as addition and multiplication) to destinations whose format is the single format. Such arithmetic engines must implement two or more user-selectable precision modes, which determine the actual precision with which floating-point operations are carried out.

In systems in which the arithmetic engine does not deliver results of floating-point operations to destinations whose format is the single format, a precision mode must be provided to cause results to be rounded to single precision. Similarly, in systems in which the arithmetic engine does not deliver results of floating-point operations to destinations in the double format, but instead delivers them to destinations wider than double precision, a precision mode must be provided to cause results to be rounded to double precision. One of the reasons for requiring these arithmetic engines to provide these precision modes is to allow them to "mimic, in the absence of over/underflow, the precisions of systems with single and double destinations." [47]

It is not clear which operations the precision mode in effect is supposed to affect, since the IEEE Standard discusses precision modes in terms of the results that should be delivered, and only mentions the word operations in a footnote. But according to section 2 ("Definitions") of the IEEE Standard, a mode is a variable which affects "the execution of ... arithmetic operations," so the precision mode in effect definitely affects the results of additions, subtractions, multiplications, and divisions. However, the first sentence of the second paragraph of section 4 ("Rounding") contains the phrase "all arithmetic operations except comparison and remainder." So perhaps the term arithmetic operation refers to all the operations defined in the IEEE Standard, although it appears to make little sense, for example, for the precision mode in effect to affect conversions between different floating-point formats.

The IEEE Standard does not specify what the default precision mode should be.

As with rounding modes, apparently the precision mode in effect must be settable dynamically, because of how the IEEE Standard defines the word mode. However, unlike the case for rounding modes, whether precision modes must actually be settable dynamically is debatable, as will be seen in section 5.2.2 below.

5.2 Hardware facilities for accessing precision modes

5.2.1 Processors with dynamic precision modes

Only a few floating-point architectures, most notably the Intel x86 and the Motorola 68000 series, have dynamic precision modes. In both of these architectures, the same special register that is used to control the rounding mode in effect is the one to which the user has access in order to, among other things, find out what precision mode is currently in effect, and to change it if so desired.

The sequence of instructions to find out what the current precision mode is, to save (only) the precision mode in some memory location, to set the precision mode to one that may or may not be statically known, and to restore a previously-saved precision mode is the same as the corresponding sequence for rounding modes (see chapter 4), except that different constants would be involved in manipulating the relevant bits. The clock timings will not be repeated here. As with rounding modes, the overhead of accessing or modifying the precision mode becomes more significant as the (maximum) floating-point performance of a processor increases.

In the Intel x86 series, the precision mode affects only the four basic arithmetic operations and the square root operation. There is no documentation on what happens if one or more operands are not representable in the format corresponding to the precision mode in effect (as might happen if a double extended precision operand were loaded in from memory). However, at least on the 486DX processor, all the bits in the operands affect the result, and the precision mode in effect merely determines how many significant (binary) digits the result of the operation will have. If the precision mode is single or double precision, neither the underflow nor the overflow exception is signaled if the result of an operation is outside the range of representable numbers for the format corresponding to the precision mode in effect, as long as the result is within the range of representable numbers for extended precision.

On the Intel 486DX processor, the precision mode in effect does not affect the performance of the floating-point unit. On other processors (not necessarily those manufactured by Intel), setting the precision mode to single or double precision may decrease the performance of the floating-point unit.

5.2.2 Processors with static precision modes

The idea of having static precision modes may seem strange at first, but they are potentially useful in an architecture lacking single-precision floating-point registers. The PowerPC is an example of such an architecture. In this architecture, the format of all floating-point registers

is the double format. The operands of all floating-point arithmetic operations are taken from floating-point registers, and the results of these operations are delivered to these registers.

The precision mode is encoded in the opcode for the following floating-point instructions: add, subtract, multiply, divide, extract the square root, and multiply-add[1] and its variations (mutiply-subtract, etc.). On some processors, such as the PowerPC 601, the single-precision versions of these instructions execute more quickly than their double-precision counterparts.

If one or more operands of these single-precision instructions are not representable as single-precision floating-point numbers, the result is undefined. If the (infinitely precise) result of one of these single-precision instructions is outside the range of the single format, the appropriate exceptions, if any, are signaled.

Dynamic precision modes can be implemented in software. Basically, the compiler would have to generate code which tests a variable (which could be held in a register) to determine what precision mode is in effect, and then executes the correct code sequence based on the result of this test. The only floating-point operations affected would be those for which the compiler is not able to determine statically what precision mode should be used.

There are several ways to implement dynamic precision modes in software on the PowerPC. One way is to avoid using the multiply-add instruction and its variations, and only generate double-precision instructions. If the precision mode in effect happens to be single precision, the result of each double-precision instruction would need to be converted to single precision[2]. The disadvantage of this scheme is that double-precision instructions may execute more slowly than single-precision instructions, and in this scheme, single-precision instructions are never generated. If the precision mode in effect is double precision, the overhead would be 2 instructions, that is, as many as 2 clock cycles. If the precision mode in effect is single precision, the overhead would be 3 instructions. Assuming the "round to single precision" instruction has the same timing characteristics as other single-precision instructions, due to the superscalar nature of this architecture, the overhead in terms of clock cycles would basically be the number of clock cycles required for the double-precision instruction to make its result available to the "round to single precision" instruction. In the case of multiplication, this would amount to 6 clock cycles on the PowerPC 601.

Another way to implement dynamic precision modes in software, one that does not favor the double precision mode so heavily, is to find out what precision mode is in effect first, and then perform the operation in the required precision. The overhead would be 2 instructions, or as many as 2 clock cycles, for one of the precision modes, and 3 instructions, or as many as 3 clock cycles, for the other mode. This overhead is significant, considering that the PowerPC 601 is capable of issuing one multiply-add instruction every clock cycle. However, it is possible to reduce this overhead by finding out what precision mode is in effect no more than once per basic block.

Actually, it is not clear whether the IEEE Standard requires dynamic precision modes, or if

[1]The multiply-add instruction produces a double-length (infinitely precise) product from two of its operands, and adds it to a third operand. The sum is then rounded to either single or double precision, and delivered to its destination.

[2]Chapter 6 explains why this happens to work. This technique does not work for the multiply-add instruction and its variations.

the latter can be static, as on the PowerPC. The reason is that one can claim that one's floating-point arithmetic engine, which is implemented as a combination of hardware and software, does not normally produce results to double or extended precision. Rather, one can claim that results are always delivered to various memory locations in the appropriate format. In order to perform, say, a single precision operation, the floating-point engine loads the operands from memory, performs the operation in single precision, producing a result which is temporarily stored in a register in either the double or an extended format (depending on the floating-point architecture), and finally stores this result to memory in the single format. From the point of view of performance, this would not be an attractive way to implement the IEEE Standard, but it does point out that at least theoretically, even though it seems that the IEEE Standard requires that precision modes be settable dynamically, in practice this is not a requirement.

5.2.3 Processors without precision modes

Floating-point architectures which are capable of delivering results of floating-point arithmetic operations to destinations whose formats may be any of the implemented floating-point formats do not need to have precision modes available in order to conform to the IEEE Standard. Most floating-point architectures fit this description.

One way to simulate dynamic precision modes on a processor lacking precision modes altogether[3] is for the compiler to generate code to always deliver the results of floating-point arithmetic operations to destinations whose format is the widest format implemented. Whenever the precision mode in effect is narrower than the widest format implemented, the result of each floating-point operation would need to be rounded to the format corresponding to the precision mode in effect, and then converted back to the widest format implemented[4]. On many processors, the overhead would range from 2 instructions, or as many as 2 or more clock cycles, to significantly more if the precision mode in effect is not the precision corresponding to the widest format implemented.

Another way to simulate dynamic precision modes on a processor lacking precision modes altogether is for the compiler to generate code to check what precision mode is in effect first, then convert the operands to the format corresponding to the precision mode in effect if necessary, perform the floating-point arithmetic operation(s) in the precision corresponding to the precision mode in effect, and finally convert the result(s) to the widest format implemented if necessary. Again, the overhead would range from 2 instructions, or as many as 2 or more clock cycles, to significantly more if the precision mode in effect is not the precision corresponding to the widest format implemented. The advantage of this scheme, however, is that the precision mode would need to be checked no more than once per basic block. As a consequence, some

[3] It makes no sense to simulate static precision modes on such processors.

[4] This assumes the significand of the widest format implemented is more than twice the width of the significand of the next-to-widest format implemented. Otherwise, if the precision mode in effect happens to be some precision in which the significand of the corresponding format is at least half as wide as the significand of the widest format implemented, floating-point arithmetic operations will need to deliver results to a destination whose format corresponds to the precision mode in effect. The result would then need to be converted to the widest format implemented.

unnecessary conversions could be eliminated.

5.3 Different ways of making precision modes available in a high-level language

As in the case of facilities for setting rounding modes, facilities available to the high-level language programmer for setting precision modes can be classified according to the granularity over which the user can control the mode in effect, as measured by the amount of source code or the amount of run time. A description of the different possibilities will not be repeated here.

However, unlike the case of facilities for setting rounding modes, it is reasonable for facilities for setting precision modes to not be explicitly available to the high-level language programmer. Instead, the precision mode in effect could be related to the compiler's floating-point expression evaluation strategy (see chapter 9), so that whatever facilities are available to influence floating-point expression evaluation could implicitly affect the precision mode in effect.

5.4 Handling static evaluation and numeric literals

Again, a discussion of evaluation of static floating-point expressions in relation to precision modes would be very similar to that of such expressions in relation rounding modes. Such a discussion will not be repeated here, especially since it would be pertinent in the context of floating-point expression evaluation (see chapter 9).

5.5 Issues related to mixed-language programming

The same kinds of issues that arise in mixed-language programming (or when using different compilers for the same language) with respect to rounding modes are also relevant to precision modes. Depending on the language, it may be reasonable for the compiler to simply assume that the precision mode in effect will never change during the execution of the program it is compiling, and that it is therefore safe to ignore the fact that precision modes have been implemented in the target machine.

5.6 What support exists in high-level languages

5.6.1 Compilers for existing languages

The same comments on the IBM C/C++ compiler for OS/2 and rounding modes apply to precision modes as well.

5.6.2 Extensions to existing languages

In the Standard Apple Numeric Environment (SANE), access to the precision mode in effect is provided via two high-level language procedures: one to determine what precision mode is

currently in effect, and one to set it. There is no provision for saving or restoring only the precision mode, though of course the programmer can write a routine to do this. When a program begins execution, the precision mode in effect is "results not shortened." When the precision mode is set, the new precision mode remains in effect until explicitly changed via another call to the procedure provided for this purpose[5]. Thus, the facilities SANE provides may be characterized in terms of the granularity of run time over which the user can control the precision mode in effect.

In general, if the compiler does not perform any interprocedural analysis, the precision mode in effect cannot be known at compile time, except from the beginning of the program until the first function or procedure is called. All floating-point expressions (including numeric literals) which are evaluated at compile time use the "results not shortened" precision mode. Floating-point expressions which are evaluated at run time use whatever precision mode is in effect at that time.

In terms of support for precision modes, the extensions in [75] are similar to those of SANE. The main differences from SANE are that no functions are provided to determine or set what precision mode is currently in effect, since such functions would not be applicable to all processors. (However, for implementations for processors for which such functions would be applicable, [75] does suggest that they provide such functions.) Also, the compiler is explicitly given permission to assume the precision mode in effect is "results not shortened" at any point in a program, unless a point in the program is under the effect of a pragma called fenv_access. This pragma does not modify the precision mode in effect. The scope of this pragma is determined statically, beginning at the point in which this pragma is turned on, and ending when this pragma is encountered in the source code again or at the end of the file, whichever is encountered first. In the programming model [75] assumes, functions do not modify the precision mode in effect of their callers, unless a function's documentation says otherwise. This model is not enforceable by the compiler.

5.6.3 Experimental language designs

μln's [30] approach to supporting precision modes is similar to the way it supports rounding modes. The default precision mode, which, in the absence of any explicit indication, is the precision mode in effect in the outermost environment of a function, is "extended precision" (that is, results not shortened).

As is the case with rounding modes, μln also provides special unary operators which specify the precision mode to be used during the evaluation of their operands, which may be arbitrary subexpressions. This facility may be used for subexpressions embedded within constant expressions.

Whenever the language requires the value of a floating-point expression to be known at compile time, the precision mode used during the evaluation of that expression is "results

[5]It is also possible to write an assembly language procedure, which, when called, possibly by a program written in a high-level language, will change the precision mode. It is unlikely that such a procedure would be hardware-independent, though, but the effect of calling such a procedure would be similar to that of calling the high-level language procedure already provided.

not shortened." All other floating-point expressions are evaluated using whatever the current precision mode happens to be. If the compiler is able to determine at compile time what this mode will be, it may be able to evaluate at least part of the expression at compile time.

Thus, given these two facilities, the programmer has control over what precision mode will be used, either at the level of a subexpression or a block at a time. In addition, functions can specify what precision mode should be in effect in the functions they call, but not that of any function further down its call chain, or that of its caller.

5.7 Why it is useful to be able to change the precision mode

It is important to note why one might want to be able to change the precision mode in effect. There are several reasons why this would be convenient:

1. *To avoid double rounding.* Double rounding may be illustrated with the following program fragment:

   ```
   a := b * c
   ```

   Suppose the precision of variables `a`, `b`, and `c` is double precision, and that the multiplication is computed to extended precision. Presumably, rounding will then occur twice: once as part of the multiplication, and a second time as part of the assignment to variable a. Double rounding may produce different results than if rounding were to occur just once. In some situations, double rounding is not desirable. The next section mentions briefly why compiler support is required in order to avoid double rounding; chapter 6 contains a more general discussion of this topic.

2. *To facilitate porting software designed for some other computer or compiler.* Whether done consciously or not, some software is written in such a way so as to depend on the features of a particular floating-point arithmetic engine or compiler. The ability to modify the precision mode in effect may make it easier to port such software to a different computer or compiler.

3. *To emulate other floating-point arithmetic engines.* Even if a program does not present anomalous behavior when run on a particular computer, one may in some cases want to obtain results similar to those obtained on some other computer, whose floating-point arithmetic engine may not have precision modes implemented, or to those obtained on a similar computer, but with a different compiler. In fact, one may want to obtain identical results on any member of a class of computers, all of which emulate some real or imagined floating-point arithmetic engine. (This topic is discussed in more detail in chapter 9.)

4. *To estimate the accuracy of a computation.* Sometimes it is difficult or time consuming to formulate the error bounds of a computation. Modifying the precision mode in effect (which may be significantly easier that rewriting the program), recomputing the result, and comparing it with previous results may in some cases be an effective way to obtain some idea of the accuracy of the computation.

5. *To determine whether one should use wider or narrower precision.* One may want to find out if one should use wider precision. In order to determine this, one might try modifying the precision mode in effect, which may be easier than rewriting the program. If one finds that using a wider precision would be beneficial, one may undertake the task of rewriting the program. On the other hand, one may want to find out if one can get away with using a narrower precision in one's program, which could have an impact on its performance. The ability to change the precision mode in effect could be very convenient in this situation.

5.8 Why compiler support is required in order to avoid double rounding

On a floating-point arithmetic engine which must have more than one precision mode available in order to conform to the IEEE Standard, there are two ways to avoid double rounding: 1) one could set the precision mode to correspond to the desired precision while the operation or operations in question are being performed; or 2) one could store the result of the operation(s) in some memory location in the format corresponding to the desired precision (provided numbers are correctly rounded when made to fit in memory), and then load the result of an operation back into the arithmetic engine if this result is needed in subsequent calculations. Chapter 6 discusses both of these techniques in detail, including constraints on their successful use.

Regardless of which technique (or combination of techniques) is used to avoid double rounding, it is clear that some cooperation from the compiler would be desirable, except perhaps if only one operation in the entire program needs a different precision mode to be in effect than all the other operations. Otherwise, if one is using the second technique described above, one might not be sure if one has obtained the desired results, particularly if one has used an overly-aggressive "optimizing" compiler. For example, how can one be sure that the number that was stored in memory is actually the number that will be used in subsequent calculations? If one is using the first technique described above, writing the program could become tedious. The reason for this is that it may not be enough to set the precision mode to correspond to the desired precision once in the program (possibly at the beginning), since one may want to perform different operations using different precisions. For example, if one is doing interpolation, say, in order to compute the value of a trigonometric function, one might start with a value obtained from a lookup table, and then evaluate a polynomial, which would provide a correction factor. The polynomial might be evaluated at high precision to reduce round-off error as well as cancellation error. The value of the polynomial would then be added to the value from the lookup table using a lower precision, which might be the precision used in most of the program[6].

[6]In this example, it might be nice if one could avoid double rounding when adding the value from the lookup table and the value of the polynomial. This can be achieved when using some floating-point units, such as the one in the Intel 80486DX, which can add two extended precision numbers together and yield a lower precision result, with rounding occurring just once.

5.9 What support should exist in high-level languages

A language designer or implementer must face a number of decisions with respect to how to handle precision modes: whether to provide any facilities at all, since precision modes are not implemented in all processors; whether the user should be able to explicitly specify what precision mode should be in effect, or whether this should somehow be done implicitly, for example, by specifying that double rounding should be avoided while performing certain operation(s); the section of source code over which one request to use a particular precision mode should apply, that is, whether a request to change the precision mode in effect should apply to a single operation, a subexpression, a series of statements, etc.; whether a function or procedure should be able to modify the precision mode of its caller or of the functions or procedures it calls; and whether the precision mode in effect should be known statically.

Some provision should be made for modifying (either implicitly or explicitly) the precision mode in effect, if only to allow the user to avoid double rounding. On some processors, such as those containing floating-point arithmetic engines conforming to Intel's x86 architecture, double rounding is not convenient to avoid otherwise. The question of how facilities for modifying precision mode in effect should be handled when the target machine does not implement precision modes is considered at the end of this section.

It ought to be possible to modify the precision mode in effect in a variety of ways. For example, it ought to be possible to tell the compiler that double rounding is to be avoided when evaluating a given (sub)expression. Otherwise, it might be tedious to write code that avoids double rounding in all cases, especially since the technique used to avoid double rounding might depend on the rounding mode in effect. Furthermore, the code to avoid double rounding can make understanding the program significantly more difficult. It may also be desirable to specify which technique the compiler should use in avoiding double rounding. If, for example, double rounding needs to be avoided in all cases except when the result of some operation is a denormalized number, a simpler technique can be used (merely setting the precision mode to correspond to the precision of the operation is sufficient), and the program's performance may be improved.

There may be some other situations in which one may want to implicitly specify what precision mode ought to be in effect at a given point in a program. Among the potential uses for different precision modes, section 5.7 above mentions porting software designed for some other computer or compiler, and emulating other floating-point arithmetic engines. Although it may be unreasonable for a language design or compiler to allow for compatibility with all possible computers, compilers, or floating-point arithmetic engines, compatibility with any one of a few, carefully chosen computers, compilers, or floating-point arithmetic engines might be feasible. In particular, the ability to emulate some "standard," possibly hypothetical, floating-point arithmetic engine may be desirable.

Besides ways of implicitly specifying what precision mode should be in effect at a given point in a program, some way of directly specifying this would also be convenient. For example, in order to estimate the accuracy of a computation, one might want to set the precision mode to correspond to single precision before recomputing the result. It would be more convenient to

be able to specify this directly, rather than specifying something like "emulate computer xyz," which happens to have only single precision arithmetic.

In order to accommodate all the uses for precision modes contemplated in section 5.7 above, the user ought to be offered a choice as far as the section of source code over which one request to use a particular precision mode should apply. The options should include the ability to specify (implicitly or explicitly) what precision mode ought to be in effect throughout a portion of a file, while a series of one or more statements is being executed, and during the evaluation of a given subexpression. The latter option is needed because it is important, for example, to be able to specify what precision mode should be in effect while the expression of an if statement is being evaluated. The other two options reduce the tedium of specifying what precision mode should be in effect when the desire is to have the same precision mode be in effect over a larger section of code. Other options are, of course, possible, but not as essential as these three.

It can be argued that an ordinary function or procedure should not have the ability to modify the precision mode in effect of its callers or the functions or procedures called. At least none of the uses for precision modes contemplated in section 5.7 above require this capability, if one views functions and procedures as being "black boxes." That is, given the expression $x + f(y)$, where the type of the variable x and the type of the value f returns are both numeric types, in terms of the uses for precision modes contemplated in section 5.7 above, it is not important what precision mode was in effect when the value of $f(y)$ was computed, any more than it matters how the value of x came to be. (x could be a parameter, for example.) What matters is what precision mode is in effect when the addition in the given expression is performed. If functions or procedures were able to modify the precision mode in effect of its callers, the value of $f(y)$ in the expression above might be very different, and the program might not work at all. If the precision mode in effect when $f(y)$ is computed should be different, then this should be so indicated when the code for $f(y)$ is compiled.

The question of whether the compiler ought to be able to determine what precision mode is in effect at a given point in a program is similar to that of whether the compiler ought to be able to determine what rounding mode is in effect at a given point in a program. Discussion of the latter appears in chapter 4; this discussion will not be repeated here.

Finally, there is the question of what to do if the target machine does not have precision modes implemented. Two kinds of facilities have been mentioned in this section: implicit and explicit means of specifying what precision mode should be in effect. Implicit means of specifying this will probably be meaningful even in target machines lacking precision modes. For example, it should be possible, in most cases, to comply with requests to avoid double rounding or to emulate computer xyz when target machines lack precision modes.

It is not completely obvious what to do with explicit means of specifying what precision mode should be in effect, particularly if the precision mode that should be in effect at a given point in a program is not known statically. There are basically two approaches in such a case: ignore the request because it is not applicable to the target machine, hopefully after providing a warning that this will happen; or honor the request and simulate (dynamic) precision modes in software. The former is much more attractive because the performance penalty of the latter would almost surely be too severe.

However, if the precision mode that should be in effect at a given point in a program is known statically, then, in practice, one must consider three cases: when the precision mode specified is "round to single precision," "round to double precision," and "round to (double) extended precision." (Widely-available processors that implement precision modes make available a possibly improper subset of these three precision modes.) One way to handle all three of these cases is to simply ignore them, again hopefully after providing a warning that this will happen. This may be a reasonable approach, since when the target machine does not implement precision modes, it may not make very much sense to attempt to comply with requests to use a specific precision mode. On the other hand, it is possible to attempt to comply with such requests.

If the precision mode specified is "round to double precision," then the compiler could produce code which converts all operands to double precision and performs all operations in double precision. Provided the widest precision implemented in the target machine is double precision, this recommendation should not prove to be objectionable. (Indeed, the following paragraphs assume the widest precision implemented in the target machine is double precision.)

If the precision mode specified is "round to (double) extended precision," then the compiler could either treat such a request as being equivalent to one for "round to double precision" (hopefully after providing a warning that this will happen), or perform extended precision operations in software. The latter would, however, not be very appealing because of the impact on performance.

If the precision mode specified is "round to single precision," then the compiler could produce code which converts all operands to single precision and performs all operations in single precision. Alternatively, all operations could be performed with the usual precision, with the result of each operation rounded to single precision. Either way presents certain problems: If all operands are first converted to single precision, results could be quite different. On the other hand, if the second alternative is used, the results of many operations could suffer double rounding: once when an operation is performed, and again when the result of that operation is converted to single precision. The second alternative is probably preferable to the first, since results would in most cases be closer to their corresponding infinitely precise results.

In the discussion in the preceding paragraph, one must keep in mind why the IEEE Standard seems to require some floating-point engines to have precision modes in the first place: to provide a way to obtain results similar to those obtained on machines that lack extended (and possibly double) precision. While this would seem to argue in favor of not even attempting to comply with requests to use a specific precision mode, making such attempts could facilitate achieving the goals mentioned in the last two uses for precision modes described in section 5.7 above, that is, estimating the accuracy of a computation, and determining whether one should use wider or narrower precision.

The discussion above brings up the point that language designs and implementations thereof should make it clear whether compilers (should) attempt to comply with requests to use a specific precision mode when the target machine does not implement precision modes. Perhaps a good design ought to provide a way for the user to specify whether or not such requests should be ignored when the target machine does not implement precision modes.

Chapter 6

Double Rounding

Double rounding is the phenomenon that occurs when the result of an operation is rounded to fit some intermediate destination, and then again when delivered to its final destination. For example, consider the following program fragment:

```
a := b * c
```

Suppose the precision of variables `a`, `b`, and `c` is double precision, and that products are computed to extended precision. The product of `b` and `c` will typically be rounded twice: once when the product is computed to extended precision, and again when the (extended precision) product is assigned to variable `a`. The value stored in variable `a` may be different than if the product of `b` and `c` were to be computed to double precision in the first place, that is, if rounding were to occur just once.

Goldberg gives the example of computing 1.9×0.66 using decimal arithmetic [65]. The exact product is 1.254, but if this product were rounded first to three significant digits and then to two, the final result would be 1.2 instead of 1.3 (the latter being closer to the exact product), if one were to round to the nearest number as specified in the *IEEE Standard for Radix-Independent Floating-Point Arithmetic* [48][1].

Double rounding can be a common occurrence when using some floating-point arithmetic engines which lack single precision registers: results of operations are typically rounded to fit in a register, whose width may be double precision or wider, before being stored in some memory location possibly in a format narrower than that of the registers. Examples of such floating-point arithmetic engines include those in processors conforming to Intel's x86 architecture, or to IBM's POWER architecture[2]. (Processors conforming to the POWER architecture appear in some older IBM workstations.)

[1] The *IEEE Standard for Binary Floating-Point Arithmetic* [47], which is better known than the aforementioned one for radix-independent arithmetic, is a subset of the latter.

[2] Even if an architecture lacks single precision registers, results of operations will not necessarily suffer double rounding. For example, in Motorola's PowerPC and DEC's Alpha architectures, results of single precision instructions are rounded to single precision (rather than double precision), even though they are stored in double precision registers. (Actually, as will be shown in this chapter, in cases such as these, it would make no difference if double rounding were to occur in the process of computing the results of single precision instructions.)

In some situations, double rounding is highly undesirable. The next section describes one such case, while the section after that discusses cases in which double rounding is absolutely harmless. The last two sections discuss how double rounding can be avoided in general.

6.1 Why double rounding can be undesirable

Consider the problem of converting a rational number, in which the numerator and the denominator are both arbitrary integers, to the nearest floating-point number that can be represented in a given floating-point format. In general, it is not obvious how this problem can be solved efficiently, but there may be many instances of this problem in which the numerator and the denominator can both be represented exactly as floating-point numbers in the same format as the final result. Therefore, it is important to be able to do this conversion very quickly in such cases. However, if double rounding cannot be avoided unequivocally even when doing an operation as simple as a division, it will be very difficult to design a solution to this problem with satisfactory performance in these cases.

How much would performance suffer if one cannot guarantee that double rounding will be avoided? In order to answer this question, consider a processor such as Intel's Pentium, and suppose that the result of the conversion routine must be in the double precision format. In this type of processor, arithmetic operations are normally computed to (double) extended precision. This means there are three ways to produce a double precision quotient: 1) produce an extended precision quotient, and then round this quotient to double precision (this alternative would be unacceptable, since it involves double rounding); 2) set the (rounding) precision mode to correspond to double precision[3] (this alternative may also be unacceptable if there is no convenient or portable way to do this); or 3) augment the first alternative with code that corrects the doubly rounded result in order to guarantee that the result will ultimately be the same as if it had been rounded just once.

One way to correct the doubly rounded result involves modifying the rounding mode in effect. This technique is discussed in more detail in section 6.3, and may be unacceptable on the same grounds as the second alternative above, and because there is a better way to correct the result: in this particular situation, double rounding is a problem only if the rounding mode in effect is "round to nearest," and the bits in the significand of the extended precision quotient which cannot be preserved in the final result consist of a leading one followed by only zeros. Therefore, all one need do is take corrective action whenever this bit pattern is detected.

In order to fully assess the impact double rounding has on this problem, one must consider what sort of corrective action is required. First, one must identify the two consecutive double precision floating-point numbers a and b that surround the extended precision quotient, and determine which of these two is closer to the infinitely precise quotient. This can be done by choosing one of the two numbers, multiplying it by the divisor using exact arithmetic, and

[3]Setting the (rounding) precision mode to correspond to double precision avoids double rounding whenever the numerator is greater than 3 or the denominator is less than or equal to $2^{E_{\max}-1}$ (see section 3.1), since in these cases the result will always be zero, infinity, NaN, or a value that can be expressed as a normalized double precision floating-point number, that is, the result will not be a denormal.

comparing this product with the dividend. Assuming $|a| < |b|$, if the difference between the exact product and the dividend is less than half an ulp (unit in the last place) of a times the divisor, the number chosen is closer to the infinitely precise quotient. If this difference is greater than half an ulp of a times the divisor, the other number is closer to the infinitely precise quotient. Finally, if this difference is exactly equal to half an ulp of a times the divisor, both numbers are equally close to the infinitely precise quotient. If a is closer to the infinitely precise quotient, a one must be subtracted from the least significant bit of the significand of the extended precision quotient. On the other hand, if b is closer to the infinitely precise quotient, a one must be added to the least significant bit of the significand of the extended precision quotient. If both a and b are equally close to the infinitely precise quotient, no correction needs to be made to the extended precision quotient. Now, when the (possibly modified) extended precision quotient is rounded to double precision, the final result will be the same as if the quotient had been rounded just once[4].

As can be seen, without the assurance that double rounding cannot occur, a simple problem, whose solution should have required no more than a division, now requires significant intellectual effort to solve. Fortunately, in this instance, the impact on performance in relative terms is not as severe as might otherwise have been the case, since corrective action is rarely needed, and division tends to take significantly more time to perform than other arithmetic operations.

It is interesting to note that it would most likely take less time to detect whether corrective action is necessary than to modify the precision mode in effect to correspond to double precision before performing the division, and then restore the precision mode back to its previous setting. Thus, even though the second alternative mentioned above (that is, the one that involves changing the precision mode in effect) is a conceptually simple solution, it might not be the solution with the best performance.

Double rounding can have an unfavorable impact on other problems as well. For example, when converting a decimal floating-point number to a binary floating-point number, in many cases, both the significand of the decimal number and the power of ten (or its reciprocal) can be expressed exactly as binary floating-point numbers in the format in which the result should be. A single multiplication or division is all that is required to produce the correctly rounded result, provided the result does not suffer double rounding.

[4]This discussion assumes it is not known what rounding mode is currently in effect, perhaps because it is not convenient or portable to determine this. If it were possible to determine that in fact the current rounding mode is not "round to nearest," no corrective action would be needed—double rounding would be harmless. Otherwise, the corrective action to be taken can be simplified somewhat: As before, let a and b be the two consecutive double precision floating-point numbers closest to the infinitely precise quotient, with $|a| < |b|$. Of these two numbers, multiply the one whose significand has a zero as its least significant bit by the divisor using exact arithmetic, and compare this product with the dividend. If the difference between the exact product and the dividend is not greater than half an ulp of a times the divisor, the number whose significand has a zero as its least significant bit is the correctly rounded (double precision) quotient. Otherwise, the other adjacent number is the correctly rounded quotient.

6.2 When is double rounding innocuous?

If double rounding can be disastrous, one might ask oneself, "Is double rounding ever harmless?" This section answers this question, at least from the context of how one can emulate single precision (binary) floating-point arithmetic when only double precision floating-point arithmetic is available.

In fact, this is the same context as that in [65], which unfortunately contains the following claim:

> *If x and y have p-bit significands, and $x+y$ is computed exactly and then rounded to q places, a second rounding to p places will not change the answer if $p \leq (q-1)/2$. This is true not only for addition, but also for multiplication, division, and square root.*

(The same claim also appears in the draft of the second edition of [65].) In addition to showing that this statement is not quite true for square root, this section contains proofs that this statement is true for the other arithmetic operations.

Information similar to that in this section apparently appears in some lecture notes from a course Prof. W. Kahan gave in 1988 [35, 42] (and to which this author does not have access). However, this information is repeated here, since it is not easily accessible.

This section considers two different algorithms for rounding to the nearest number: biased and unbiased rounding. In unbiased rounding, that is, rounding to nearest or even as specified in [47], if there are two consecutive floating-point numbers equally near to the infinitely precise result, the floating-point number whose significand has a zero as its least significant bit is chosen as the correctly rounded result. Biased rounding is similar, except that in the situation above, the floating-point number with the larger magnitude is chosen as the correctly rounded result.

If the rounding mode used is not "round to nearest" (that is, if the rounding mode is "round toward zero," "round toward positive infinity," or "round toward negative infinity"—see chapter 4), it is easy to see that double rounding cannot cause the final result to be different from what would have been obtained with single rounding, as long as single precision floating-point numbers are a subset of double precision floating-point numbers.

Even when rounding to the nearest number, double rounding is of concern only if the significant digits of the infinitely precise result form one of either one or two specific bit patterns. These patterns depend on whether biased or unbiased rounding is used. In the former case, the bit pattern of concern is

$$1d_1d_2\ldots d_{p-1}011\ldots 11d_{q+1}d_{q+2}\ldots,$$

where p and q are the number of digits in the significands of single and double precision numbers, respectively, and each d_n, $n > 0$, is a one or a zero. The reason why this pattern is of concern is that such a number, rounded to q digits, would yield $1d_1d_2\ldots d_{p-1}1$. Subsequent rounding of this number to p digits would yield the next largest p-digit number. However, the single precision number closest to the infinitely precise result would simply be $1d_1d_2\ldots d_{p-1}$.

If unbiased rounding is used, the bit patterns of concern are

$$1d_1d_2\ldots d_{p-2}1011\ldots 11d_{q+1}d_{q+2}\ldots,$$

which is similar to the pattern of concern for biased rounding, and

$$1d_1d_2\ldots d_{p-2}0100\ldots 00d_qd_{q+1}\ldots,$$

where either d_q is one and d_{q+n}, $n > 0$, are all zeros, or d_q is zero and d_{q+n}, $n > 0$, are not all zeros. The reason why the latter pattern is of concern is that such a number, rounded to q digits, would yield $1d_1d_2\ldots d_{p-2}01$. Subsequent rounding of this number to p digits would yield $1d_1d_2\ldots d_{p-2}0$. However, the single precision number closest to the infinitely precise result would be the next largest p-digit number: $1d_1d_2\ldots d_{p-2}1$.

The rest of this section proves, for the various arithmetic operations (addition, subtraction, multiplication, division, and square root), how many digits the significands of double precision numbers must have in order for double rounding to always yield the same result as would be obtained if rounding to yield a single precision number were to occur just once. These proofs take into account both unbiased rounding as well as biased rounding. (In some cases, the proofs are slightly different, depending on which of these two rounding methods is used.)

6.2.1 Addition

Theorem 1 *Let x and y be positive binary floating-point numbers whose significands consist of at most p digits, where $p \geq 2$, and let z be the binary floating-point number that most closely approximates $x + y$, and whose significand consists of at most q digits, where $q > p$. The binary floating-point number that most closely approximates $x + y$ and whose significand consists of at most p digits is the one that most closely approximates z if and only if $q \geq 2p + 1$.*

Proof. Without loss of generality, assume $x \geq y$; otherwise, interchange x and y in the rest of this proof. We will restrict our attention to floating-point numbers x and y such that $1 \leq x + y < 2$, since all pairs of positive binary floating-point numbers can be scaled by powers of two to meet these constraints.

There are potentially two cases in which our hypothesis is not trivially true: if $x + y$ (in infinite precision) were to look something like

$$1.d_1d_2\ldots d_{p-2}0100\ldots 00d_qd_{q+1}\ldots,$$

where either d_q is one and d_{q+n}, $n > 0$, are all zeros, or d_q is zero and d_{q+n}, $n > 0$, are not all zeros; or if $x + y$ (in infinite precision) were to look something like

$$1.d_1d_2\ldots d_{p-1}011\ldots 11d_{q+1}d_{q+2}\ldots.$$

In both cases, $y < 2^{p-q}$, since d_{q+n}, $n \geq 0$ are not all zeros. If $q \geq 2p + 1$, then $y < 2^{-p-1}$. Now, in the sum $x + y$ there are either $p + 1$ or $p + 2$ significant digits to the left of the $p + 2$nd digit to the right of the radix point, the first and last of which are nonzero. Therefore, x cannot be a number with at most p significant digits if $q \geq 2p + 1$ and $x + y$ (in infinite precision) looks something like $1.d_1d_2\ldots d_{p-2}0100\ldots 00d_qd_{q+1}\ldots$ or like $1.d_1d_2\ldots d_{p-1}011\ldots 11d_{q+1}d_{q+2}\ldots$.

If $p < q < 2p + 1$, the theorem is false: Consider the case where $x = 1 + 2^{1-p}$ and $y = 2^{-p}(1 - 2^{-p})$. $x + y$ looks something like

$$1.00\ldots001011\ldots11,$$

where there are $p - 2$ consecutive zeros immediately to the right of the radix point followed by a one, a zero, and p consecutive ones. If $p < q < 2p + 1$, then $z = 1 + 3 \cdot 2^{-p}$, and the p-digit number that most closely approximates z is $1 + 2^{2-p}$, whereas the p-digit number that most closely approximates $x + y$ is x. ∎

6.2.2 Subtraction

Theorem 2 *Let x and y be positive binary floating-point numbers whose significands consist of at most p digits, where $p \geq 2$, such that $x > y$, and let z be the binary floating-point number that most closely approximates $x - y$, and whose significand consists of at most q digits, where $q > p$. The binary floating-point number that most closely approximates $x - y$ and whose significand consists of at most p digits is the one that most closely approximates z if and only if $q \geq 2p + 1$ (using unbiased rounding) or $q \geq 2p$ (using biased rounding).*

Proof. We will restrict our attention to floating-point numbers x and y such that $1 \leq x - y < 2$, since all pairs of positive binary floating-point numbers x and y such that $x > y$ can be scaled by powers of two to meet these constraints.

There are potentially two cases in which our hypothesis is not trivially true: if $x - y$ (in infinite precision) were to look something like

$$1.d_1d_2\ldots d_{p-2}0100\ldots00d_qd_{q+1}\ldots,$$

where either d_q is one and d_{q+n}, $n > 0$, are all zeros, or d_q is zero and d_{q+n}, $n > 0$, are not all zeros; or if $x - y$ (in infinite precision) were to look something like

$$1.d_1d_2\ldots d_{p-1}011\ldots11d_{q+1}d_{q+2}\ldots.$$

If $x - y$ were to look as in the first case, then $x - y = 2^{2-p}a + 2^{-p} + c$, or

$$x = 2^{2-p}a + 2^{-p} + c + y,$$

where a is an integer such that $2^{p-2} \leq a < 2^{p-1}$, and $0 < c \leq 2^{-q}$. Now, $y < 2^{p-q}$, since d_{q+n}, $n \geq 0$ are not all zeros. If $q \geq 2p + 1$, then $y < 2^{-p-1}$ and $0 < c \leq 2^{-2p-1}$, so $c + y \leq 2^{-p-1}$. The sum $2^{2-p}a + 2^{-p}$ consists of one digit to the left of the radix point and p digits to the right of the radix point, for a total of $p + 1$ significant digits. Adding $c + y$ to this quantity can only increase the number of significant digits. Therefore, x cannot be a number with at most p significant digits if $q \geq 2p + 1$ and $x - y$ (in infinite precision) looks something like $1.d_1d_2\ldots d_{p-2}0100\ldots00d_qd_{q+1}\ldots$.

If $x - y$ were to look as in the second case, then $x - y = 2^{1-p}a + 2^{-p} - c$, or

$$x = 2^{1-p}a + 2^{-p} - c + y,$$

where a is an integer such that $2^{p-1} \leq a < 2^p$, and $0 < c \leq 2^{-q}$. Now, $y < 2^{p-q}$, since d_{q+n}, $n \geq 0$ are not all zeros. If $q \geq 2p$, then $y < 2^{-p}$ and $0 < c \leq 2^{-2p}$, so $-2^{-2p} < y - c < 2^{-p}$. The sum $2^{1-p}a + 2^{-p}$ consists of one digit to the left of the radix point and p digits to the right of the radix point, for a total of $p+1$ significant digits. Adding $y-c$ to this quantity cannot decrease the number of significant digits. Therefore, x cannot be a number with at most p significant digits if $q \geq 2p$ and $x-y$ (in infinite precision) looks something like $1.d_1 d_2 \ldots d_{p-1} 011 \ldots 11 d_{q+1} d_{q+2} \ldots$.

If $p < q < 2p + 1$, the theorem is false if using unbiased rounding: Consider the case where $x = 1 + 2^{1-p}$ and $y = 2^{-p}(1 - 2^{-p})$. $x - y$ looks something like

$$1.00 \ldots 00100 \ldots 001,$$

where there are $p - 1$ consecutive zeros immediately to the right of the radix point followed by a one, $p - 1$ consecutive zeros, and a one. If $p < q < 2p + 1$, then $z = 1 + 2^{-p}$, and the p-digit number that most closely approximates z is 1, whereas the p-digit number that most closely approximates $x - y$ is x.

If $p < q < 2p$, the theorem is also false, regardless of whether biased or unbiased rounding is used: Consider the case where $x = 1 + 2^{2-p}$ and $y = 2^{-p}(1 + 2^{1-p})$. $x - y$ looks something like

$$1.00 \ldots 001011 \ldots 11,$$

where there are $p - 2$ consecutive zeros immediately to the right of the radix point followed by a one, a zero, and $p - 1$ consecutive ones. If $p < q < 2p$, then $z = 1 + 3 \cdot 2^{-p}$[5], and the p-digit number that most closely approximates z is x, whereas the p-digit number that most closely approximates $x - y$ is $1 + 2^{1-p}$. ∎

6.2.3 Multiplication

Theorem 3 *Let x and y be positive binary floating-point numbers whose significands consist of at most p digits, where $p \geq 4$, and let z be the binary floating-point number that most closely approximates xy, and whose significand consists of at most q digits, where $q > p$. If $q \geq 2p$, the binary floating-point number that most closely approximates xy and whose significand consists of at most p digits is the one that most closely approximates z. (This is not always the case if $p < q < 2p$[6].)*

Proof. We will restrict our attention to floating-point numbers y where $1 \leq y < 2$, and floating-point numbers x such that $1 \leq xy < 2$, since all pairs of positive binary floating-point numbers can be scaled by powers of two to meet these constraints.

There are potentially two cases in which our hypothesis is not trivially true: if xy (in infinite precision) were to look something like

$$1.d_1 d_2 \ldots d_{p-2} 0100 \ldots 00 d_q d_{q+1} \ldots,$$

[5] If $p = 2$, this is true only if using biased rounding.

[6] The reason for restricting p to values no smaller than 4 is that q can be less than $2p$ otherwise.

where either d_q is one and d_{q+n}, $n > 0$, are all zeros, or d_q is zero and d_{q+n}, $n > 0$, are not all zeros; or if xy (in infinite precision) were to look something like

$$1.d_1 d_2 \ldots d_{p-1} 011 \ldots 11 d_{q+1} d_{q+2} \ldots .$$

Now, $x = 2^e a$ for some integer e and some integer a such that $2^{p-1} \leq a < 2^p$. Similarly, $y = 2^{1-p} b$ for some integer b such that $2^{p-1} \leq b < 2^p$. Thus, $xy = 2^{e-p+1} ab$, with $2^{2p-2} \leq ab < 2^{2p}$. Therefore, since xy consists of at most $2p$ significant digits, xy cannot look like $1.d_1 d_2 \ldots d_{p-2} 0100 \ldots 00 d_q d_{q+1} \ldots$ or like $1.d_1 d_2 \ldots d_{p-1} 011 \ldots 11 d_{q+1} d_{q+2} \ldots$.

If $p < q < 2p$, the theorem can be false if using unbiased rounding: Consider the case where $p = 4$ and $x = y = 13$. $xy = 13^2 = 169 = 10101001_2$. If $p < q < 2p$, then $z = 168$, and the p-digit number that most closely approximates z is 160, whereas the p-digit number that most closely approximates xy is 176[7]. ∎

6.2.4 Division

Theorem 4 *Let x and y be positive binary floating-point numbers whose significands consist of at most p digits, where $p \geq 2$, and let z be the binary floating-point number that most closely approximates x/y, and whose significand consists of at most q digits, where $q > p$. If $q \geq 2p$, the binary floating-point number that most closely approximates x/y and whose significand consists of at most p digits is the one that most closely approximates z. This is not necessarily the case when using unbiased rounding and $p < q < 2p$.*

Proof. We will restrict our attention to floating-point numbers y where $2^{p-1} \leq y < 2^p$, and floating-point numbers x such that $1 \leq x/y < 2$, since all pairs of positive binary floating-point numbers can be scaled by powers of two to meet these constraints.

There are potentially two cases in which our hypothesis is not trivially true: if x/y (in infinite precision) were to look something like

$$1.d_1 d_2 \ldots d_{p-2} 0100 \ldots 00 d_q d_{q+1} \ldots ,$$

where either d_q is one and d_{q+n}, $n > 0$, are all zeros, or d_q is zero and d_{q+n}, $n > 0$, are not all zeros; or if x/y (in infinite precision) were to look something like

$$1.d_1 d_2 \ldots d_{p-1} 011 \ldots 11 d_{q+1} d_{q+2} \ldots .$$

If x/y were to look as in the first case, then $x/y = 2^{2-p} a + 2^{-p} + c$, or

$$x = 2^{2-p} ay + 2^{-p} y + cy,$$

where a is an integer such that $2^{p-2} \leq a < 2^{p-1}$, and $0 < c \leq 2^{-q}$. Now, $2^{2-p} ay + 2^{-p} y$ consists of at least p digits to the left of the radix point, and no more than p digits to the right of the radix point. If $q \geq 2p$, then $0 < cy < 2^{-p}$, and adding cy to $2^{2-p} ay + 2^{-p} y$ can only

[7] Another example is $p = 5$ and $x = y = 23$, and yet another is $p = 6$, $x = 45$, and $y = 59$. The latter example shows that this theorem can be false if $p \geq 6$ and $p < q < 2p$, even when using biased rounding.

increase the number of significant digits in the latter quantity. Therefore, x cannot be a number with at most p significant digits if $q \geq 2p$ and x/y (in infinite precision) looks something like $1.d_1d_2\ldots d_{p-2}0100\ldots00d_qd_{q+1}\ldots$.

If x/y were to look as in the second case, then $x/y = 2^{1-p}a + 2^{-p} - c$, or

$$x = 2^{1-p}ay + 2^{-p}y - cy,$$

where a is an integer such that $2^{p-1} \leq a < 2^p$, and $0 < c \leq 2^{-q}$. Now, $2^{1-p}ay + 2^{-p}y$ consists of at least p digits to the left of the radix point, and no more than p digits to the right of the radix point. If $q \geq 2p$, then $0 < cy < 2^{-p}$, and subtracting cy from $2^{1-p}ay + 2^{-p}y$ can only increase the number of significant digits in the latter quantity. Therefore, x cannot be a number with at most p significant digits if $q \geq 2p$ and x/y (in infinite precision) looks something like $1.d_1d_2\ldots d_{p-1}011\ldots11d_{q+1}d_{q+2}\ldots$.

If $p < q < 2p$, the theorem is false if using unbiased rounding: Consider the case where $x = 1$ and $y = 1 - 2^{-p}$. Computing the first few terms of the Taylor series expansion yields

$$x/y = 1/(1 - 2^{-p}) \approx 1 + 2^{-p} + 2^{-2p} + 2^{-3p},$$

which looks something like

$$1.00\ldots00100\ldots00100\ldots,$$

where there are $p-1$ consecutive zeros immediately to the right of the radix point followed by a one, $p-1$ consecutive zeros, and another one. If $p < q < 2p$, then $z = 1+2^{-p}$, and the p-digit number that most closely approximates z is 1, whereas the p-digit number that most closely approximates x/y is $1 + 2^{1-p}$[8]. ∎

6.2.5 Square root

Theorem 5 *Let x be a positive binary floating-point number whose significand consists of at most p digits, where $p \geq 2$, and let y be the binary floating-point number that most closely approximates $\sqrt{x}$, and whose significand consists of at most q digits, where $q > p$. The binary floating-point number that most closely approximates $\sqrt{x}$ and whose significand consists of at most p digits is the one that most closely approximates y if and only if $q \geq 2p + 2$.*

Proof. We will restrict our attention to floating-point numbers x where $1 \leq x < 4$, since all positive binary floating-point numbers can be written as $2^e x$, where e is an even integer and $1 \leq x < 4$.

There are potentially two cases in which our hypothesis is not trivially true: if $\sqrt{x}$ (in infinite precision) were to look something like

$$1.d_1d_2\ldots d_{p-2}0100\ldots00d_qd_{q+1}\ldots,$$

[8]Another example is $p = 4$, $x = 13$, and $y = 11$: $x/y = 1.0010111010001\ldots$, so this theorem is most likely false if $p \geq 4$ and $p < q < 2p$, even when using biased rounding.

where either d_q is one and d_{q+n}, $n > 0$, are all zeros, or d_q is zero and d_{q+n}, $n > 0$, are not all zeros; or if $\sqrt{x}$ (in infinite precision) were to look something like

$$1.d_1 d_2 \ldots d_{p-1} 011 \ldots 11 d_{q+1} d_{q+2} \ldots .$$

If $\sqrt{x}$ were to look as in the first case, then

$$2^{2-p}a + 2^{-p} < \sqrt{x} \leq 2^{2-p}a + 2^{-p} + 2^{-q},$$

where a is an integer such that $2^{p-2} \leq a < 2^{p-1}$. Let $z = 2^{2-p}a$. This means that

$$\begin{aligned} x &> z^2 + 2^{1-p}z + 2^{-2p} \\ x &\leq z^2 + 2^{1-p}z + 2^{-2p} + 2^{1-q}z + 2^{1-p-q} + 2^{-2q}. \end{aligned}$$

Now, the lower bound for x is a number with exactly $2p$ digits to the right of the radix point and one or two digits to the left of the radix point. Thus, x cannot be a number with at most p significant digits unless the upper bound for x were no smaller than the smallest p-digit number larger than the lower bound.

However, if $q \geq 2p$ (a stricter constraint than was mentioned in the theorem), then the upper bound cannot be greater than or equal to this p-digit number. The reason is that $z^2 + 2^{1-p}z$ has at most $2p-3$ digits to the right of the radix point, while $2^{-2p} + 2^{1-q}z + 2^{1-p-q} + 2^{-2q} < 2^{3-2p}$. In other words, the lower bound looks something like $d_{-1}d_0.d_1 d_2 \ldots d_{2p-3}001$, and no matter how big $2^{1-q}z + 2^{1-p-q} + 2^{-2q}$ is, adding the latter quantity to the lower bound cannot affect the lower bound's $(2p-3)$rd digit. Therefore, if $q \geq 2p$, $\sqrt{x}$ (in infinite precision) cannot look something like $1.d_1 d_2 \ldots d_{p-2} 0100 \ldots 00 d_q d_{q+1} \ldots$.

If $\sqrt{x}$ were to look as in the second case, then

$$2^{1-p}a + 2^{-p} - 2^{-q} \leq \sqrt{x} < 2^{1-p}a + 2^{-p},$$

where a is an integer such that $2^{p-1} \leq a < 2^p$. Let $z = 2^{1-p}a$. This means that

$$\begin{aligned} x &\leq z^2 + 2^{1-p}z + 2^{-2p} \\ x &> z^2 + 2^{1-p}z + 2^{-2p} - 2^{1-q}z - 2^{1-p-q} + 2^{-2q}. \end{aligned}$$

Now, the upper bound for x is a number with exactly $2p$ digits to the right of the radix point and one or two digits to the left of the radix point. Thus, x cannot be a number with at most p significant digits unless the lower bound for x were no larger than the largest p-digit number smaller than the upper bound.

However, if $q \geq 2p + 2$, then the lower bound cannot be less than or equal to this p-digit number. The reason is that $2^{1-q}z + 2^{1-p-q} - 2^{-2q} < 2^{-2p}$. Therefore, subtracting this quantity from the upper bound cannot result in a number consisting of fewer than $2p$ digits to the right of the radix point. Therefore, if $q \geq 2p + 2$, $\sqrt{x}$ (in infinite precision) cannot look something like $1.d_1 d_2 \ldots d_{p-1} 011 \ldots 11 d_{q+1} d_{q+2} \ldots$.

Operation	Minimum number of digits required
Addition	$2p+1$
Subtraction	$2p+1$ (unbiased rounding)
	$2p$ (biased rounding)
Multiplication	$2p$
Division	$2p$
Square root	$2p+2$

Table 6.1: Number of digits required to emulate single precision arithmetic using double precision arithmetic

If $p < q < 2p+2$, the theorem is false: Consider $x = 1 - 2^{-p}$. Computing the first few terms of the Taylor series expansion yields

$$\sqrt{1-2^{-p}} \approx 1 - 2^{-p-1} - 2^{-2p-3} - 2^{-3p-5},$$

which looks something like

$$0.11\ldots11011\ldots11011\ldots,$$

where there are p consecutive ones immediately to the right of the radix point followed by a zero, $p+1$ consecutive ones, and another zero. If $p < q < 2p+2$, then $y = 1 - 2^{-p-1}$, and the p-digit number that most closely approximates y is 1, whereas the p-digit number that most closely approximates $\sqrt{x}$ is x. ∎

6.2.6 Additional comments

In order to emulate single precision floating-point arithmetic faithfully using double precision arithmetic, if results are rounded to the nearest representable floating-point number, double precision floating-point numbers must consist of more than twice as many significant digits as single precision floating-point numbers. More specifically, Table 6.1 lists the minimum number of significant digits required for each of the five arithmetic operations. (In this table, p is the number of digits in the significands of single precision numbers, *Addition* refers to the addition of two numbers with the same sign, and *Subtraction* refers to the addition of two numbers with opposite signs.)

It may not be apparent from Table 6.1 that with just two or three exceptions, even if the significands of double precision numbers were to consist of only $2p$ significant digits, one could avoid changing the final result of an arithmetic operation when double rounding occurs: If one were to make just a few slight modifications to Priest's proof of Theorem 5 [67] (which postdates this author's proof by a few days), one could show that if it were not for numbers of the form $2^e(1-2^{-p})$, where e is an even integer, only $2p$ digits would be needed for square root. Also, for subtraction (using unbiased rounding) and addition, if double precision numbers consisted of only $2p$ significant digits, double rounding could change the final result only if the operand

whose magnitude is smaller were equal to $2^{e-p}(1-2^{-p})$, where e is the exponent of the operand whose magnitude is larger.

(Incidentally, judging from the proof of Theorem 4, one gets the impression that it may possibly be easier to prove theorems about floating-point arithmetic engines conforming to [47] than about those that do not fully conform to this standard.)

Alas, the author was not able to show that for any $p \geq 6$, q must be greater than or equal to $2p$ in order for Theorem 3 to be true, regardless of whether biased or unbiased rounding is used. Also, the author was not able to show that for any $p \geq 4$, q must be greater than or equal to $2p$ in order for Theorem 4 to be true when biased rounding is used.

6.3 On avoiding double rounding

Double rounding can be easily avoided on a floating-point arithmetic engine which does not need to have precision modes available in order to conform to the IEEE Standard. But on those that must have more than one precision mode available, there are two ways to avoid the effects of double rounding: 1) one could set the precision mode to correspond to the desired precision while the operation or operations in question are being performed; or 2) one could store the result of the operation(s) in some memory location in the format corresponding to the desired precision (provided numbers are correctly rounded when made to fit in memory), and then load the result of an operation back into the arithmetic engine if this result is needed in subsequent calculations.

Note that both of these techniques can be used in combination. For example, if the precision of all the variables in a program is not wider than double precision, the precision mode could be set to "round to double precision" for the duration of the entire program. Any single precision operations for which one wants single precision results could be handled using the second technique described above.

The first technique described above is not always effective in eliminating double rounding. This is because the IEEE Standard does not fully specify what happens when the result of an operation lies outside the range of representable numbers for the format corresponding to the precision mode in effect, and the precision mode in effect corresponds to a precision that is narrower than the widest precision implemented. For example, on the Intel 80486DX, the widest precision implemented is (double) extended precision. If one were to set the precision mode to correspond to double precision, and the result of an operation is finite and lies outside the range of representable numbers for double precision, the significand of the result would still be rounded just as if the result were representable as a double precision number, but the exponent would not be in the range for double precision exponents. This means that if it were necessary to store the result in memory for some reason, the number stored in memory could differ from the original result: the number stored in memory could either be $\pm\infty$ instead of some large finite number, or a denormalized number or zero instead of a small normalized number. In the second case, even though the precision mode is set to correspond to double precision, the "final" result of an operation may still suffer double rounding: once when the operation is performed, and a second time when the result of this operation is stored in memory. Thus, a

small change in the program or a change in the level of optimization could cause the results to be different, depending on whether the result of an operation has to be stored in memory or not.

If the rounding mode in effect is "round to nearest," the second technique described above is also not always effective in eliminating the harm from double rounding. Section 6.4 mentions what the constraints for the successful use of this option are. As briefly outlined in [41], one way to do away with these constraints and that is effective regardless of the rounding mode in effect is to set the rounding mode to "round toward zero" before performing an operation. If the inexact exception is signaled while performing the operation, the (infinitely precise) result can be rounded in the same way it would be rounded if there were no trap handler for the inexact exception (that is, rounded toward zero), and if this rounded result is exactly half way between two consecutive floating-point numbers representable in the destination format, then a one could be added to or subtracted from the position occupied by the least significant bit of the significand of the rounded result, depending on the sign of the rounded result[9]. Now, when this possibly modified result is stored to memory using the original rounding mode (which might not have been "round toward zero"), the stored result will be the same as if rounding had occurred once.

The only constraint for this technique to work is that, assuming the radix for the representation of floating-point numbers is two, the representation of the significand of wide precision floating-point numbers must be at least one bit wider than that of the destination format, and the range of exponents for the former must be at least as wide as that of the latter. In other words, the midpoint of every interval whose endpoints are distinct adjacent floating-point numbers exactly representable in the destination format must be exactly representable as a wide precision floating-point number. In addition, the largest finite floating-point number exactly representable in the destination format plus one half ulp of that must be exactly representable as a wide precision floating-point number, and similarly for the smallest finite floating-point number exactly representable in the destination format.

Note that this technique may not be portable, since it potentially involves changing the rounding mode twice.

[9]Special consideration is needed if the original rounding mode is "round toward positive infinity" or "round toward negative infinity." If the original rounding mode is "round toward positive infinity" and the rounded result is positive, then a one must be added to the position occupied by the least significant bit of the significand if the rounded result is exactly representable in the destination format. Similarly, if the original rounding mode is "round toward negative infinity" and the rounded result is negative, then a one must be subtracted from the position occupied by the least significant bit of the significand if the rounded result is exactly representable in the destination format. One might think that if the original rounding mode is "round toward positive infinity" or "round toward negative infinity" and the rounded result is zero, it may not be possible to know what the final result should be: zero, the largest negative number, or the smallest positive number. However, if the result is zero, the inexact exception will not be raised: if the infinitely precise result is nonzero, the result can be best approximated by a nonzero wide precision floating-point number, rather than by zero.

6.4 Practical ways of avoiding double rounding

From the discussion in the previous section, it would seem that there are two foolproof ways of avoiding double rounding. One potentially involves changing the rounding mode twice, and is therefore likely to drastically slow down floating-point arithmetic performance (see chapter 9), and the other way (converting wide precision results to a narrower format, that is, innocuous double rounding) can only be used if certain constraints are met, and is therefore not always applicable to a given situation. Are there any other ways of avoiding double rounding without limiting constraints and without severely degrading performance?

Before examining other techniques for avoiding double rounding, however, one ought to consider exactly how much the two techniques mentioned above affect floating-point performance. The steps involved in performing a floating-point operation using the first technique would typically be as follows:

- save the current rounding mode and the inexact flag;
- clear the inexact flag and disable the inexact exception handler if applicable;
- set the rounding mode to "round toward zero;"
- perform the operation itself;
- find out if the significand of the result ends in the right number of zeros, and if so, find out what rounding mode was saved in the first step above (in most cases, nothing will need to be done to the result);
- restore the rounding mode that was saved in the first step above;
- convert the result to the destination format;
- if the inexact flag is not set and the final result is the same as the original result, restore the inexact flag to its original state and reenable the inexact exception handler if applicable;
- otherwise, set the inexact flag, reenable the inexact exception handler if applicable, and signal the inexact exception.

As can be seen, there is a very significant amount of overhead involved for every floating-point operation. Certainly, pipelining of floating-point operations would not be possible.

It may be possible to omit some of the steps outlined above. For example, the current rounding mode only needs to be saved once per basic block, and perhaps not at all if it is known statically. Also, in most cases, there will be no exception handler for the inexact exception, and in fact, it may not even be important to faithfully keep track of whether the inexact exception ever occurred while performing some computation. Even so, on an Intel Pentium, floating-point arithmetic performance would slow down by a factor of at least 24.

In contrast, simply generating wide precision results and converting each result to its destination format slows down floating-point arithmetic performance by a factor of between 4 and

7 on an Intel Pentium, depending on whether other (independent) instructions can execute concurrently. Performance could be improved if it were possible to perform this conversion without actually having to access memory. Unfortunately, the Pentium's instruction set does not allow for this.

The simplest and potentially fastest way to avoid double rounding is to set the precision mode to correspond to the desired precision. This is especially attractive if the precision mode in effect hardly ever needs to be modified, but it is not completely foolproof, since this technique does not work if the result of an operation is in the range of denormalized numbers. This technique can be modified to make it foolproof: Before performing any floating-point operation, the operand or operands can be stored in some convenient memory location (or in floating-point registers), along with a code that indicates what operation is about to be performed[10]. After the operation is performed, the result can be examined, and if it is within the range of denormalized numbers, the operation can be performed again using one of the techniques described above[11]. This technique would slow down floating-point arithmetic performance by a factor of about 10 on an Intel Pentium, assuming the precision mode in effect does not need to be changed.

Another way to avoid double rounding is to set the precision mode to correspond to the desired precision, copy the operand or operands along with a code that indicates what operation is about to be performed, and perform the operation. The result can then be stored in memory in the destination format, and compared with the original result. If the value stored in memory is not identical to the original result, the operation can be repeated using one of the techniques described above. This technique would slow down floating-point arithmetic performance by a factor of between 14 and 17 on an Intel Pentium, depending on whether other (independent) instructions can execute concurrently while the two values are being compared. This technique is slower on the Pentium than the previous technique because of the awkwardness in comparing two floating-point values on the Pentium.

Still another way to avoid double rounding is to again copy the operand or operands along with a code that indicates what operation is about to be performed, then generate a wide precision result (using the rounding mode that should be in effect for the given section of code of which the operation is a part), and find out if this wide precision result is exactly half way between two adjacent floating-point numbers representable in the destination format. If so, the operation can be repeated using one of the techniques described above. Otherwise, the result can be safely converted to the destination format. This technique would slow down floating-point arithmetic performance by a factor of about 18 on an Intel Pentium.

The Java Grande Forum reports an ingenious way of avoiding double rounding on processors conforming to the Intel x86 architecture, which also involves setting the precision mode to correspond to the desired precision [51]. The observation is made that when the precision

[10] Floating-point units for the Intel x86 series processors have registers containing information on the last floating-point instruction executed, including information on the operand(s), but these registers cannot be used in this case because they do not actually contain the operand(s) themselves as they existed before the operation was performed.

[11] It may seem that one could establish an exception handler for the underflow exception to avoid the overhead of having to check the magnitude of the result, but this would not work, since narrow precision denormalized numbers are perfectly normal numbers in wide precision.

mode is set as described, double rounding is of concern only when performing multiplication or division, in which case one of the operands can be scaled in such a way that if the unscaled result would be a denormal if represented in the desired data format, the scaled result would also be a denormal in extended precision. The scaling is then undone. This technique would slow down floating-point arithmetic performance by a factor of about 4 on an Intel Pentium.

To summarize, then, if the precision of a series of operations is the same, and the result of each operation is known not to be within the range of denormalized numbers (or if double rounding is acceptable when results lie within this range), then the best way to avoid double rounding is to simply set the precision mode to correspond to the desired precision. Otherwise, for single precision operations, the best way to avoid double rounding is to use the second technique described in this section (that is, generate a wide [double or wider] precision result, and convert it to single precision), since double rounding can be proven to be harmless in this case. Otherwise, if a number of double precision operations need to be performed, the last technique (reported by the Java Grande Forum) is best. Finally, if extended precision operations need to be performed along with only a few double precision operations in between, the penultimate technique described above is best for double precision operations. Of course, all of this presupposes that the rounding mode in effect is either "round to nearest" or unknown, since otherwise double rounding could be easily avoided using the same techniques as for single precision operations.

Interestingly, on an Intel Pentium, double rounding can be avoided in almost all cases sometimes with practically no performance degradation, but if double rounding must be avoided in absolutely all cases, floating-point arithmetic performance slows down by roughly an order of magnitude if less than (double) extended precision is desired. Therefore, double rounding should be avoided only if absolutely necessary.

Chapter 7

Supporting the Standard Operations in High-Level Languages

7.1 IEEE Standard requirements in regards to operations

The IEEE Standard requires implementations to provide the following operations:

- basic arithmetic operations (addition, subtraction, multiplication, and division), with the result being in a format at least as wide as the wider operand[1] (this comment also applies to the remainder operation below);
- remainder, defined as follows: Let n be the integer closest to x/y. (If two consecutive integers are equally close, let n be the even one.) The remainder $r = x - yn$. (The result of this operation is not affected by the rounding or precision mode in effect.);
- square root;
- conversions among the supported floating-point formats (if more than one format is supported, that is);
- conversions between floating-point and integer formats;
- rounding a floating-point number to an integer value, with the result being a floating-point number;
- conversion between decimal numbers represented according to an implementation defined specification and binary floating-point numbers; and

[1] Quite likely, one reason for the requirement on the format of the result is to ensure that certain information (as specified in sections 7.3 and 7.4 of [47]) can be provided to a trap handler when overflow or underflow is signaled [40].

- comparison between any two floating-point numbers, whose formats may differ.

Implementations are allowed to consider copying a floating-point value without changing its format as an operation, which would affect whether exceptions are signaled when floating-point values are moved from one location to another.

The last two operations in the list above merit further comments. The strict accuracy requirements applicable to the basic arithmetic operations also apply to conversion between binary and decimal floating-point numbers, provided the value being converted falls within a certain specified range. Otherwise, another set of more relaxed accuracy requirements apply. Presumably, this is because this is the best that could be done using reasonable algorithms commonly known at that time—see, for example, the algorithms Coonen developed [19]. (By now, it has been shown that the same strict accuracy requirements that apply to other operations can be reasonably applied to conversion between binary and decimal floating-point numbers as well [58, 71, 33].)

The result of a comparison may be delivered in one of two ways: as a true-false response to a predicate naming the relation of interest, or conceptually as a string of four bits identifying which relation is true. These four bits correspond to the relations greater than, less than, equal, and unordered, with the last one being true whenever one or both operands is NaN (Not a Number). If an implementation provides predicates for comparison, it must at least provide the six traditional ones: $>$, $<$, $\geq$, $\leq$, $=$, $\neq$. In addition, the IEEE Standard recommends that a predicate for the unordered relation be provided, and the appendix of the IEEE Standard recommends that a predicate for $<>$ (less than or greater than) be provided (see below). (As many as 26 predicates are possible, since one of the possible relations between two values is unordered, and since two versions of each type of comparison are possible: one signaling the invalid operation exception whenever at least one of the operands is a NaN, and the other one not. There are fewer than 32 because predicates that are always true or always false are not interesting, and the equal and not equal predicates never signal the invalid operation exception as long as neither of the values being compared is a signaling NaN.)

It is important to note that these functions are all specified in a generic manner. At least one "instantiation" of each of these functions is required, but the IEEE Standard does not require that any one particular "instantiation" be provided. For example, the IEEE Standard does not specifically require implementations to provide a function which adds two floating-point numbers in the single format. It does, however, require implementations to provide a function which adds two floating-point numbers, at least one of which is in the single format. The sum may be delivered to a destination in any supported format, the only restriction being that the sum be correctly rounded to the precision of one of the supported formats that is at least as wide as that of the operands and not wider than the destination (lest the result suffer double rounding).

In addition to these operations, the appendix of the IEEE Standard (which is technically not part of the standard itself) recommends that implementations provide functions to do the following:

- copy the sign from one number to another, possibly without checking what the value of the latter number is (note that this function can be used to find out what the sign of a

value is);

- change the sign of a number, whose value may be NaN, without performing subtraction;
- scale a number by an integral power of two, where the power of two is not approximated, but exact;
- extract the (unbiased) exponent of a number, with the result being a floating-point number whose value is an integer;
- given two floating-point numbers, find a floating-point number closer to one of them and adjacent to the other one;
- determine if a number is finite;
- determine if a value is NaN;
- determine if a number is less than or greater than another number (note that this is not equivalent to "not equal" if one or both operands are NaNs, and that the IEEE Standard does not actually require the comparison operation to be capable of determining if a number is either less than or greater than another number);
- given two values, determine if one or both are NaNs (although in many implementations this may be accomplished by simply comparing the two values, the IEEE Standard does not actually require the comparison operation to be capable of determining if one or both operands are NaNs);
- given a value, determine what kind of value it is (zero, finite, NaN, etc.), and what its sign is;
- convert a string to a floating-point value as though at run time, taking into consideration the rounding and precision modes currently in effect, as well as the side effects (that is, exceptions) that would occur; and
- rounding a floating-point number to an integer value, with the result being a floating-point number, and without signaling the inexact exception[2].

(The last two functions appear only in the appendix of the more generalized version of the IEEE Standard [48].) Of course, exceptional situations may arise when performing some of these functions; the appendix of the Standard specifies what should be done in these situations (with [48] being slightly more thorough than [47]). Source code for some of these functions is publicly available [18].

[2]Unlike [47], [48] normally requires signaling the inexact exception when rounding a floating-point number to an integer value if the value of the floating-point number is not already an integer.

7.2 How different architectures implement these operations

The IEEE Standard may be implemented totally in hardware, totally in software, or (more typically) as a combination of hardware and software. This section surveys several IEEE-compatible floating-point architectures, and discusses which of the operations mentioned in the previous section are implemented in hardware, and which operations must be implemented in software.

7.2.1 CISC architectures

Intel x86 architecture

Members of Intel's x87 series of chips (such as the 8087 and 80287), some of which were designed concurrently with the development of the IEEE Standard, are implementations of the first floating-point architecture intended to be IEEE-compatible [69]. As one might expect from a CISC architecture, almost all of the operations described in the IEEE Standard are implemented in hardware. The following three paragraphs discuss the operations not fully implemented in hardware:

- A partial remainder instruction is provided to facilitate implementation of the remainder operation. If the difference between the (binary) exponents of the dividend and divisor is less than 64, then executing this instruction once is sufficient. Otherwise, this instruction must be executed repeatedly until the magnitude of the dividend becomes no larger than half that of the divisor.

- Conversions between any supported floating-point or integer format and the double extended format are implemented, but other conversions, such as conversions between the single and double formats, require a series of two instructions: a load into an extended precision register, and a store to memory.

- The software required to convert between binary and decimal floating-point numbers can be simplified by using instructions which convert between decimal integers whose digits are represented as hexadecimal digits, and integers represented as binary floating-point numbers. In addition, there are instructions which compute $f(x) = 2^x - 1$, where $|x| < 1$, and $f(x, y) = y \log_2 x$, as well as an instruction to load the constant $\log_2 10$ into a register.

In addition, five of the recommended functions mentioned in the appendix of the IEEE Standard are implemented in hardware: change the sign of a number, scale a number by a power of two, determine if a number is greater than or less than another number, determine if a given value is NaN, and determine if one or both of two given values are NaNs. (The latter three can be accomplished by using comparison.) Some of the other recommended functions can be easily implemented in software using special instructions:

- extracting the exponent of a number requires two instructions: one to separate the exponent from the significand, and another to get rid of the significand;

- finding out if a number is finite requires a series of instructions: one to determine what kind of number it is, and several more to determine if the number was found to be normal, denormalized, or zero (on some processors, such as the Intel Pentium, this function can be implemented more quickly in software without using any special instructions);

- there is an instruction which, given a value, determines what kind of value it is (zero, finite, NaN, etc.), and what its sign is, but the bits in the status register which contains the result of this instruction need to be isolated and stored in some other register or in memory.

The remaining two functions must be implemented in software without the benefit of any special instructions: copy the sign from one number to another, and find a floating-point number adjacent to another one.

Except for store instructions, all instructions which normally deliver a floating-point result do so to a floating-point register. Furthermore, since at least one of the operands in a binary arithmetic operation must be held in a register, the formats of the operands of such an operation are often dissimilar. In general, it is not possible to obtain a correctly rounded double precision sum, difference, product, or quotient of two double precision numbers, or square root of a double precision number, without software assistance[3].

Motorola 68000 architecture

Motorola is another early implementer of the IEEE Standard. Some of the floating-point units in the M68000 family implement the IEEE Standard in hardware to an even greater extent than members of Intel's x87 series of chips. In addition to implementing the same operations and functions in hardware as the latter, the MC68881 and MC68882 chips also fully implement the following operations and functions in hardware:

- the remainder operation;

- decimal floating-point numbers in which each digit of the mantissa and exponent is represented by a hexadecimal digit can be converted to an extended precision (binary) floating-point number and vice-versa with a single instruction; in fact, up to one operand of an arithmetic operation may be a decimal floating-point number;

- extracting the exponent of a number.

As in the Intel x86 architecture, except for store instructions, all instructions which normally deliver a floating-point result do so to a floating-point register. In contrast to the Intel x86 architecture, however, it is possible to obtain correctly rounded single and double precision

[3]As explained in chapter 6, even setting the precision mode to correspond to double precision will not always yield a correctly rounded double precision result.

results by setting the precision mode appropriately[4]. Nevertheless, single and double precision results are converted to (double) extended precision.

Some members of the M68000 family, such as the MC68040 and MC68060, do not implement the IEEE Standard in hardware as fully as some other members, such as the MC68881 and MC68882. In particular, the former two do not implement in hardware the remainder operation, conversions between binary and decimal floating-point numbers, scaling by an integral power of two, or exponent extraction. In addition, the MC68040 does not implement rounding a floating-point number to an integer value in hardware. Interestingly, the floating-point units in many implementations of RISC architectures also do not implement these operations in hardware.

7.2.2 RISC architectures

The floating-point units of implementations of RISC architectures tend to implement a relatively small subset of the IEEE Standard in hardware: the basic four arithmetic operations and perhaps square root; conversions between different floating-point formats and between floating-point and integer formats; and comparison. In addition, most RISC architectures provide instructions to change the sign of floating-point values, and to determine if a given value is NaN (which is usually accomplished by comparing the given value with itself). The following paragraphs describe how some of the popular RISC architectures differ from what has been described in this paragraph.

Motorola PowerPC architecture

The instruction set of the PowerPC architecture includes an instruction to extract the square root of a number. However, this instruction is not implemented in some implementations of this architecture, such as the PowerPC 601, 603, and 604 chips.

Although all floating-point registers store numbers in the double format, separate instructions are provided for single and double precision arithmetic. All such instructions take their operand(s) from one or more registers, and deliver their results to a register. In the case of single precision instructions, operands are assumed to be values which are exactly representable in the single format; otherwise, the result is undefined. Although results of single precision instructions are exactly representable in the single format, they are converted to the double format before being stored in a register.

Digital Alpha architecture

The instruction set of the Alpha architecture includes an instruction which copies the sign from one number to another (a similar instruction can be used to change the sign of a number), but does not include an instruction to extract the square root of a number. Although there is an

[4] As mentioned in chapter 5, accessing the precision mode frequently is likely to degrade floating-point performance significantly. Also, note that although the Motorola 68000 architecture includes instructions that produce single or double precision products or quotients regardless of the precision mode in effect, these instructions may produce results outside the range of single or double precision numbers. Therefore, these instructions are not always useful in avoiding double rounding.

instruction to determine if a value is NaN, or alternatively if one or both of two given values are NaNs, if indeed one or both values are NaNs, this instruction always traps. Therefore, unless performance is not critical, this instruction is only useful in practice if the given value or values are not likely to be NaNs.

As in the PowerPC architecture, separate instructions are provided for single and double precision arithmetic, even though all registers store numbers in the double format.

SPARC architecture

Although the instruction set of the SPARC architecture includes an instruction to extract the square root of a number, it does not include instructions to convert floating-point numbers to integers using the rounding mode currently in effect, as specified in the IEEE Standard. However, it does include instructions to convert floating-point numbers to integers using the rounding mode "round toward zero."

Separate instructions are provided for single, double, and quadruple precision arithmetic. The floating-point registers are capable of storing up to 32 different single precision values. Pairs of (single precision) floating-point registers may be used to store up to 16 double precision values, while quadruplets of (single precision) floating-point registers may be used to store up to 8 quadruple precision values.

7.2.3 How different architectures implement comparison

As mentioned previously, the IEEE Standard allows comparison to be implemented in one of two ways: as an operation that sets one of four bits depending on the relation between the two operands, or as a set of predicates. The former is the most common way to implement comparison. Indeed, of the architectures mentioned above, only the Alpha architecture implements comparison as a set of (four) predicates: equal, less than, less than or equal, and unordered. Other predicates (greater than, greater than or equal) are available by reversing the order of the operands, so this set of predicates barely fulfills the IEEE Standard's minimum requirement in this area. If any operand of any of these predicates (including unordered) is NaN, a trap occurs (as would similarly occur if a page fault were encountered). These predicates deliver a zero or nonzero number to a floating-point register, depending on whether the relation specified is false or true, respectively. Control can then be conditionally transferred to another part of the program based on whether the result of the predicate is zero or nonzero[5].

Another architecture that implements comparison as a set of predicates is Hewlett-Packard's PA-RISC. However, in this architecture, a full set of 32 predicates is provided, 16 of which signal the invalid operation exception only when one or both operands are signaling NaNs. (The remaining 16 predicates signal the invalid operation exception whenever one or both operands is any kind of NaN [see chapter 8].) These predicates set a bit in the status register to indicate

[5]Control can be conditionally transferred to another part of the program based on the relation of the value in a register and zero, but the instructions provided for this purpose can behave unexpectedly when the value in a register is not finite.

whether the result is true or false. This bit can then be tested using another instruction, with execution of the instruction following that depending on the value of this bit.

Other architectures typically have two comparison instructions: one which signals the invalid operation exception whenever one or both operands are NaN, and another which only does so if one or both operands are signaling NaNs. Both instructions set the proper bit or bits in a status register, depending on the relation between the operands. Control can then be conditionally transferred to another part of the program based on which bit or bits are set in this register. In some architectures, the contents of this register must first be transferred to another register, such as a fixed-point or a different status register, before conditionally transferring control to another part of the program.

For example, for the Intel x86 architecture, the typical sequence is to compare the two values, transfer the contents of the status register to the accumulator, and optionally transfer the contents of the accumulator to the flags register[6]. Control can then be conditionally transferred to another part of the program using a sequence of one or more instructions, depending on the desired predicate[7].

On the other hand, the SPARC architecture implements comparison in a manner similar to the Precision Architecture, except that two different comparison instructions are provided, and the desired predicate determines which of 16 different variations of the floating-point branch instruction is used.

7.3 Making the standard operations available in high-level languages

An obvious way of making all these operations available in a high-level language is through function calls. This is the easiest way, since most, if not all, programming languages already provide some way of calling functions. Therefore, it would not be necessary to extend a language in order to make these operations available in this way.

A disadvantage of solely using function calls is the negative impact on the readability of programs, assuming the language normally uses some other syntax, such as infix notation, for expressing numerical computation. A more significant disadvantage is that in some cases, the overhead of calling a function can greatly overwhelm the actual computation of the function. This can severely degrade floating-point performance. Of course, it is possible to treat these functions as special inline functions to reduce the impact on performance. The operations that would benefit the most from such inlining are addition, subtraction, multiplication, conversions between the supported floating-point formats, and comparison. These operations tend to be

[6] Although Intel's literature does not appear to mention this, it is no slower, and likely faster in many cases, to test the value in the accumulator directly without first transferring the contents of the accumulator to the flags register.

[7] For some predicates, such as greater than, one (conditional jump) instruction can suffice if the number of bytes between the jump instruction and its target is small enough. For other predicates, such as equal, at least two instructions are needed: one to determine if the relation between the two operands is ordered or unordered, and another to determine if the relation is equal. The reason has to do with what combination of bits in the flags register are checked when the various conditional jump instructions are executed.

well supported in hardware (that is, they require one or at most a few instructions, each of which executes quickly), and they occur quite frequently in programs in which floating-point computation is dominant.

Another way in which these operations can be made available is through special syntax, such as infix notation. However, in some languages, this would require extending the definition of the language, which may be difficult or impractical, and suitable syntax for some of these operations may not be obvious. For example, it is not easy to devise suitable syntax for an operation which, when given two floating-point numbers, finds a floating-point number closer to one of them and adjacent to the other one. Even if such syntax were invented, depending on the actual syntax used, readability of programs can be reduced in proportion to the increase in size of the lexical vocabulary.

Yet another way to make these operations available is through attributes associated with data types. Conceptually, this approach is similar to the object oriented programming paradigm: Each floating-point data type corresponds to an instance of the class of floating-point numbers. In order to perform an operation, the desired instance and operation (that is, attribute) must be specified, along with the data to be used in the operation. One disadvantage to this approach is that it does not easily allow for operations involving more than one floating-point data format. Also, not all programming languages provide a way of denoting attributes in general.

To illustrate the trade-offs involved, consider comparison. There can be up to 26 comparison predicates, since the traditional trichotomy does not hold, and some predicates signal the invalid operation exception when an operand is any kind of NaN. It would be easy to provide access to all 26 predicates using function calls, but this may degrade performance, and writing something like `lt(a, b)` is not as natural as writing `a < b`.

On the other hand, it is not easy to find suitable syntax to denote, say, an operation that returns true if either operand is NaN, or if the first operand is less than or equal to the second, and that does not signal the invalid operation exception unless either operand is a signaling NaN. One might try something like `a ?<= b`, but at first glance it is not apparent what this expression means. In particular, it is not intuitively obvious whether the invalid operation exception will be signaled if either operand is a quiet NaN.

The comparison predicates could also be made available using attributes. For example, double'lt(a,b) might denote the less than predicate for operands whose format is the double format. Disregarding whatever virtues (or lack thereof) this syntax may have, how would one denote comparison between an operand whose format is the single format, and another whose format is the double format? Of course, one may decide that comparison should only be allowed between operands whose format is the same.

Finally, more than one way could be used to provide access to the different predicates. For example, some predicates might be available via infix notation, while access to others might require making function calls. This approach may be attractive for languages in which special syntax is already provided for some of the predicates, and whose definitions cannot be conveniently modified.

However, this approach is not entirely satisfactory. The next section mentions one reason

why this is so. A second reason is that the lack of consistency can be troubling. Consider, for example, a hypothetical language in which the predicate "less than, greater than, or unordered" (that is, not equal) is available via infix notation, whereas access to the predicate "less than or greater than" requires a function call, resulting in very dissimilar precedences for the two predicates. Why should there be a difference in how these predicates are accessed, and why should their precedences be different? In such a language, one may be tempted to make all comparison predicates available through function calls for the sake of consistency.

7.4 Handling static evaluation

In many programming languages, certain expressions are allowed to be evaluated at compile time rather than at run time. These expressions are often allowed to involve certain operations, such as the basic four arithmetic operations. Some syntactic features, such as arbitrary function calls, might not be allowed in such expressions[8]. If this is the case, the syntax used to denote the operations described in the IEEE Standard could impact which operations such expressions can involve.

The set of operations that a language definition allows in an expression evaluated at compile time often seems arbitrary. Going back to the example at the end of the previous section, the predicate "less than, greater than, or unordered" (that is, not equal) might be allowed in such expressions because special syntax can be used to denote this predicate, whereas the predicate "less than or greater than" might not be allowed because it requires a function call. Ideally, all the operations described in the IEEE Standard and at least some of the ones in the appendix, such as changing the sign of a number or scaling a number by a power of two, should be allowed in these expressions, even though not all implementations of the IEEE Standard provide these operations in hardware.

7.5 What support exists in high-level languages

7.5.1 Existing language designs

Many languages provide a way to denote that basic arithmetic is to be performed on floating-point numbers; many language definitions even go so far as to specify what the data type of the result of a given expression is. Unfortunately, there is usually no correlation between the type and the accuracy of the result, nor is it usually specified what the mapping is between the symbols of which the language makes use and the operations specified in the IEEE Standard, or even those available in the target arithmetic engine.

Without this knowledge, it is not possible to determine with complete certainty which specific operation will be performed at a given point in a program. For example, one might expect that the expression `a+b`, where both `a` and `b` represent arbitrary double precision floating-point numbers, ought to yield the sum of these two numbers, correctly rounded to double

[8] A likely reason for this is that in some languages, operations that are only available through function calls are not universally implemented in hardware.

precision. But if the language definition does not actually require this, a (hypothetical) fully conforming language processor may actually instruct the target machine to produce the sum accurate to extended precision (making it cumbersome at best to devise a way to obtain this sum correctly rounded to double precision), or worse, to only single precision. Alternatively, a fully conforming language processor may instruct the target machine to produce the sum accurate to double precision, but the sum might not be correctly rounded (for example, the sum might suffer double rounding).

For example, the definition of the C language [16] seems to suggest that arithmetic involving double precision quantities must be performed using double precision arithmetic, unless the language processor "can ascertain that the result would be the same as if it were executed using double-precision arithmetic." However, this document provides little clue as to the distinction between, say, single and double precision arithmetic, other than to require that the set of single precision values be a subset of the set of double precision values. One may get the impression that the floating-point model on which the various constants (such as DBL_MANT_DIG, the number of digits in the significands of double precision values, and DBL_MAX_EXP, roughly the maximum exponent a double precision value may have) defined in the standard header file float.h are based would have bearing on what constitutes single and double precision arithmetic, but this model is not required to coincide with the model on which the implementation is based.

Of course, even if a language definition does not mandate a particular mapping between the symbols of the language and the operations specified in the IEEE Standard, an implementation may still provide access to the latter, perhaps through a combination of special syntax (such as infix notation) provided by the language, and function calls. Therefore, it is relevant to consider how many operations can be made available in a hypothetical implementation without resorting to function calls, as well as which operations can be made available using calls to functions defined in standard libraries (if any).

Again, taking the definition of the C language as an example, the following operations cannot be made available in any way other than through function calls: remainder, square root, and rounding a floating-point number to an integral value. There are standard library functions which compute remainders and square roots, but the definitions of these functions are slightly different. As a result, in the case of the former, different results may be returned, and in the latter, additional action beyond that specified in the IEEE Standard is required when the operand is negative.

Of the functions mentioned in the appendix of the IEEE Standard, the unary minus operator can be taken to correspond to changing the sign of a number, and there is a standard library function which scales a number by an integral power of two. There is another standard library function which can be used to extract the exponent of a number, but the result would in most cases be one more than what the function described in the appendix would return, and the result is delivered as an integer, rather than as a floating-point number. In order to determine if a number is finite, one could take the absolute value of the number using a standard library function, and compare that to the macro HUGE_VAL, assuming this macro is actually an expression whose value is infinity. It may appear that the semantics of a few of the other functions in this appendix could be expressed using comparison, but the possibility of side

effects in the computation of the value(s) of interest would make this strategy cumbersome.

Ada [11] is different from most languages in the sense that if an implementation claims to support the Numerics Annex of the language standard, the (relative) error in results from predefined operators on floating-point numbers must be within certain bounds. Even this requirement, though, does not go far enough to force implementations to provide access to the operations described in the IEEE Standard.

Despite this, an implementation can provide access to most of the operations described in the IEEE Standard using the facilities described in the language standard. Remainders and rounding to an integral value can be computed via floating-point attributes, and square roots may be extracted by calling a standard library function. However, the language definition makes no provision for directly converting floating-point values to integer formats in a way that is compatible with the IEEE Standard, though this can be accomplished by first rounding the floating-point value to an integral value, and then converting the latter to an integer format.

Many of the functions described in the appendix of the IEEE Standard are available via attributes: copying the sign from one number to another, scaling a number by a power of two, finding a number adjacent to another one, and extracting the exponent of a number. (As in C, the result from the latter would in most cases be one more than what the function described in the appendix would return.) The unary minus operator can be used to change the sign of a number.

None of the languages discussed in this section provide more than the traditional six relational operators. Any other kind of relational operation would require a call to a non-standard function.

7.5.2 Compilers for existing languages

Sun's C compiler provides access to all the operations described in the IEEE Standard and to most of the functions described in the appendix [74]. Access to some operations, such as remainder, square root, and rounding to an integral value, requires calling functions in their C library. The only functions that are not provided are "less than or greater than," and determining if one or both of two given values are NaNs. Access to only the traditional six relational operations are provided.

7.5.3 Extensions to existing languages

The Standard Apple Numerics Environment (SANE) provides access to all the operations described in the IEEE Standard, as well as to most of the functions described in the appendix [12]. Except for conversions from the double extended format, operations produce results in the double extended format if they deliver a floating-point result, regardless of the format of their operands[9]. Computing remainders, extraction of square roots, floating-point to integer conversion, and rounding to integral value can only be accomplished by calling functions, such as ones provided in the standard library. Explicit conversions may be indicated by calling other

[9]If an operand is not already in the double extended format, it is converted to this format before the operation is performed.

standard library functions. In addition to the traditional six relational operators, a function in the standard library can be called to find out what the relationship between two values is, if any. However, there is no way to avoid signaling the invalid operation exception if an operand is NaN, unless the equality operators (that is, equal or not equal) are used.

No functions are provided to determine if a value is finite or NaN, if a number is less than or greater than another number, or if one or both of two given values are NaNs.

In terms of support for the operations and functions described in the IEEE Standard, the extensions in [75] are similar to those of SANE, except that no standard library function is provided to convert a value from a wider to a narrower floating-point format (explicit casts can be used instead), or to find out what the relationship between two values is. Operations which produce a floating-point value do not necessarily deliver results in the double extended format.

The sign of a value can be changed by using the unary minus operator. Standard library functions provide access to all the other functions described in the appendix of the IEEE Standard.

7.5.4 New language designs

μln [30] provides operators for all the operations described in the IEEE Standard, including fourteen relational operators. Although μln does not explicitly provide functions as described in the appendix of the IEEE Standard, some of μln's operators mimic their functionality.

Both the Java [38] and Limbo [59] languages are different from the languages mentioned above in that conforming language translators must generate code which, when executed, exhibits behavior identical to that of a specified virtual machine executing code generated by a translator for that virtual machine. (The most straightforward way of achieving this, of course, is to use a reference translator, whose target is the specified virtual machine, and to execute the resultant generated code on a simulator of that virtual machine.) These languages are therefore not directly portable to a wide variety of targets in the traditional sense, and they can thus make many assumptions about the target machine that other more widely portable languages cannot make.

Despite the fact that the Java Virtual Machine (JVM) [61] is compatible with the IEEE Standard, the language definition does not mention a way to directly convert a value from a floating-point format to an integer format in the manner described by the IEEE Standard. Computing remainders, extraction of square roots, and rounding to integral value can only be accomplished by invoking methods (which are analogous to functions), such as ones provided in the standard library. Access to only the traditional six relational operations are provided. Of the functions described in the appendix of the IEEE Standard, the only ones to which access is directly provided are changing the sign of a number and determining if a value is NaN. One can determine if a value is finite by negating the (Boolean) value returned by a method which determines if a value is infinite, provided, of course, the value in question is not a NaN.

In order to map more directly to the actual hardware being used, Limbo does not adhere to the IEEE Standard as closely as does Java. (For example, Dis, Limbo's corresponding virtual machine, is not guaranteed to provide gradual underflow.) However, it does seem to provide

access to all the operations described in the IEEE Standard[10] and to the first seven functions described in the appendix of the IEEE Standard. As in Java, access to only the traditional six relational operations are provided.

7.6 What support should exist in high-level languages

There are a number of issues to consider when discussing how a high-level language definition or processor ought to support the operations and functions described in the IEEE Standard and its appendix. Perhaps the first issue to come to mind is which operations and functions should be supported. Certainly all the operations described in the IEEE Standard should be supported. Of the functions described in the appendix, changing the sign of a number and determining if a value is NaN are particularly important. In addition, providing some way to determine the sign of a number, although not mentioned in the IEEE Standard or its appendix, could prove useful. (A function that copies the sign from one number to another, as recommended in the appendix of the IEEE Standard, would be even more useful, since it could be used for a wider variety of purposes[11].) However, there is little excuse for not providing support for all the operations and functions described in the IEEE Standard and its appendix, given that most, if not all, languages provide some way to call functions.

A more interesting question is whether these operations and functions should be supported via special syntax, function calls, or attributes. To answer this question, one must consider whether the mechanism used affects the performance of programs. If so, one should rank these operations and functions according to how frequently they are encountered in programs. If the amount of time it takes to perform a particular operation or function rarely impacts the overall performance of programs, then the mechanism used is not so critical. On the other hand, if an operation or function occurs frequently in programs, then the mechanism that provides the best performance should be used. The operations and functions that tend to occur frequently in programs in which numerical computations predominate include all the operations described in the IEEE Standard except for remainder, rounding to an integral value, and conversions between binary and decimal numbers, as well as changing the sign of a number and determining if a value is NaN, both of which are described in the appendix.

Another consideration is how well a particular mechanism fits in with the way other features of the language are designed. For example, if a language does not already have a mechanism similar to attributes, then it would probably not make sense for the only use of such a mechanism to be making these operations and functions available.

Of course, an existing language may already provide a way to denote operations such as the basic arithmetic operations. Whenever possible, whatever syntax the language already provides should be used to denote the operations and functions described in the IEEE Standard.

[10] However, the only floating-point format supported is the double format. Therefore, it is not possible, for example, to multiply two numbers represented in the single format, as the IEEE Standard requires.

[11] Having a function that merely determines the sign of a number would be quite useful, even if a more general function as recommended in the IEEE Standard is provided. Apparently, the IEEE Standard did not recommend the former in order to reduce the number of recommended functions [43, 20].

Sometimes, however, this is not possible. For example, a language may have a special operator to denote the remainder operation. When applied to floating-point operands, the semantics of this operator may differ from that described in the IEEE Standard. (The IEEE Standard's remainder operation sometimes produces a negative result.) In this case, it would probably not be a good idea to change the semantics of this operator from what the language definition has already established. Instead, this operation should be made available some other way. The operations that can typically be made available without modifying the language definition include all the operations described in the IEEE Standard except for remainder and rounding to an integral value, as well as changing the sign of a number. (Ideal support for comparison may require modifying the language definition—see below.)

Regardless of what mechanism is used to provide access to these operations and functions, a language definition or processor should not add overhead every time an operation is performed. For example, in C [16], whenever the standard square root function is called with a negative argument, a certain value has to be stored in a certain variable to indicate this. This means that whenever a C language processor cannot be certain that an argument to the square root function cannot be negative, it must either cause the argument (or result or status flags—see chapter 8) to be tested[12]. This kind of overhead can be significant, particularly if a single precision result is desired (since on many machines, it can be obtained relatively quickly compared to the speed with which double precision results can be obtained), especially since it may cause the language processor to emit a function call, even if the target machine has an instruction to perform the operation.

Another pitfall to avoid is to design a language or implementation in such a way that operations that are not always available in hardware end up occurring frequently in programs. For example, the main consideration in designing the Java and Limbo languages was not high performance floating-point computation. This is reflected in the fact that both these languages provide an operation that multiplies two double precision numbers, yielding a correctly rounded double precision product[13]. Unfortunately, in general, this, and other similar operations, require a series of instructions on architectures such as the Intel x86 (see chapter 6), thus precluding any hope for stellar floating-point performance on such architectures. Since this particular operation would likely occur quite frequently in programs that perform intensive floating-point computations, Java and Limbo may not be good languages to use to write such programs.

Notwithstanding the above, there are times when such operations become indispensable, in some cases because otherwise, slower algorithms must be employed. Therefore, a language design or implementation ought to make such operations available somehow, with the understanding that use of such operations may curtail the floating-point performance of a program

[12]In some cases, it may be possible to arrange to have the floating-point engine generate a signal whenever an argument to the square root function is negative. Whenever this signal is generated, the computer can store the required value in the required variable. The advantage of this scheme is that very little overhead is incurred when an argument to the square root function is nonnegative. However, this can be a very tricky scheme to implement, since the signal has to be generated at the right point in time, and the handling of this signal must not interfere with any signal handlers the user may have specified. Lack of precise interrupts in the target machine may further complicate matters.

[13]In Java 2 [4], double precision products are not required to be correctly rounded, unless strict mode is used.

if not used only when absolutely necessary. Incidentally, a weakness in Ada's floating-point model is that one is not guaranteed, for example, that the product of two double precision numbers is actually a double precision number. (The product might be correctly rounded to double extended precision instead.) Having an attribute that maps its operand to the nearest representable double precision number does not help, since the result may still not be correctly rounded to double precision, that is, the result may suffer double rounding. In short, it is not possible to write an Ada program that is guaranteed to be capable of multiplying two arbitrary double precision numbers, producing a correctly rounded double precision product, without essentially resorting to a floating-point engine implemented in software, even if the program will only be run on machines whose (hardware) floating-point engines conform to the IEEE Standard.

Most languages provide a set of six relational operators: $=$, $\neq$, $<$, $\leq$, $>$, $\geq$. Unfortunately, this set of six does not take into account the lack of an ordering relationship between NaNs and other numbers. As was mentioned earlier, a total of 26 distinct relational operators are possible. This set of 26 can be reduced to a more compact set of 14; the remaining 12 can be made accessible using Boolean negation. In most cases, it should be possible to provide access to all 14 relational combinations via functions calls; finding a set of symbols to represent each one of these different combinations in an intuitive manner can be challenging. Of the 8 relational combinations not represented by the six traditional relational operators, the two most important ones are unordered and less than or greater than. These two relational combinations should be accessible in the same way the traditional six are. The remaining six combinations can be made accessible via a call to a (possibly built in or intrinsic) function whose arguments are the values to be compared and the set of relations of interest, that is, greater than, less than, equal to, or unordered. Another possible argument to such a function might be whether the invalid operation exception should be signaled if one or both of the values being compared are any kind of NaN.

Finally, the operations and functions described in the IEEE Standard and its appendix can be made available redundantly in more than one way. For example, some of the relational combinations (mentioned in the paragraph above) might only be available via function calls, whereas as other combinations could be denoted by either special syntax involving symbols such as $<$, $>$, and $=$, as well as via function calls. (Of course, there are tradeoffs to this scheme—see section 7.3.) A slightly different example is decimal to binary conversion. In many languages, the appearance of a real numeric literal in a source program implicitly denotes that decimal to binary conversion is to take place. However, such a conversion might also be indicated by a function call or some other special syntax (such as type casting), or both, which could be useful in cases in which one wants the value converted to a specific floating-point format.

Chapter 8

Supporting Exceptional Situations in High-Level Languages

8.1 IEEE Standard requirements in regards to special computational situations

In addition to ordinary (normalized) floating-point numbers, the following special values must be representable in all supported formats:

- ± 0, which may be thought of as (signed) numbers whose magnitude is infinitely small, even though both are equal to zero (that is, they both compare equal to each other);
- $\pm \infty$, whose magnitude is larger than that of any finite number;
- quiet and signaling NaNs, which do not represent any number (NaN stands for Not a Number); the former, unlike the latter, generally propagate through computations without causing any exceptions to be signaled; and
- denormalized numbers, which are numbers with reduced precision that allow for gradual underflow; unlike normalized numbers, their most significant digit is zero, and their magnitude is very small[1].

Except for how quiet and signaling NaNs are to be distinguished[2], the actual encoding for these special values is specified for the basic formats (that is, the single and double formats—see Chapter 3). For the extended formats, the encoding of these special values is implementation dependent.

[1] Without denormalized numbers, important axioms, such as $x \neq y \Rightarrow x - y \neq 0$ would not hold [23, 69].

[2] According to the appendix of the IEEE Standard, "it is a mistake to use the sign bit [in a floating-point format] to distinguish signaling NaNs from quiet NaNs." This is because otherwise, changing the sign of a value could change a quiet NaN (which may have crept into a computation due to the occurrence of a prior exceptional situation) to a signaling NaN or vice versa, causing unwarranted exceptions to be signaled, and preventing warranted exceptions from being signaled.

Assuming trap handlers are absent, these special values usually arise as a result of exceptional situations. For example, when underflow occurs, the result is usually (but not always) a denormalized number or ± 0, whereas $\pm\infty$ is usually produced when overflow or division by zero occurs. When an invalid operation is attempted, or when an operand is a signaling or quiet NaN, the result is usually a quiet NaN. (A signaling NaN is never produced in the course of normal computation. Its appearance in a computation is always due to an explicit user request, such as from a language processor.)

The IEEE Standard lists special rules that apply when these special values appear as operands. For example, (-0)-(+0) yields -0. On the other hand, (+0)-(+0) yields +0, except when the rounding mode is "round toward negative infinity," in which case the result is (-0). 0/0 produces a (quiet) NaN, as does 0*infinity. The sum or product of infinity and a finite number is an appropriately signed infinity.

An exceptional situation is one in which there is no universal agreement as to what action should be taken [52, 54]. Five different kinds of exceptional situations are defined:

- invalid operation, such as extracting the square root of a negative number, multiplying zero and infinity, or doing basic arithmetic involving NaNs;
- division by zero, with the dividend being a nonzero finite number;
- overflow, which occurs whenever the magnitude of the (mathematically) infinitely precise result is so large that if there were no restrictions on the destination format's exponent range, the rounded result would not fit within the (actual) constraints of the destination format;
- underflow, which occurs whenever the (mathematically) infinitely precise result is nonzero, and its magnitude is strictly between the largest normalized negative number and the smallest normalized positive number of the destination format[3]; and
- inexact, which occurs whenever the (mathematically) infinitely precise result differs from the delivered result[4].

Whether trap handlers can be associated with these exceptional situations is implementation dependent. If no trap handlers are associated with these exceptional situations (which is the default), and the destination has a floating-point format, the result to be delivered is specified in the IEEE Standard. In most cases, the delivered result is the most noncontroversial one for a given situation. For example, when the inexact exception is signaled, the default result is

[3]If no trap handler is associated with the underflow exception, underflow is not signaled unless abnormal loss of accuracy also occurs, that is, if the actual rounded result differs from either the infinitely precise result or the rounded result that would be produced if there were no constraints on the exponent range, depending on the implementation. Another implementation dependent detail is whether the magnitude of the result is tested before or after rounding. None of this affects the actual delivered result, but it does cause the exact situations in which the underflow exception is signaled to be implementation dependent.

[4]If the overflow exception is signaled and no corresponding trap handler is associated with it, the inexact exception is also signaled.

the correctly rounded result. When an invalid operation is attempted, the default result is a NaN, provided the destination has a floating-point format. The default response for division by zero, a correctly signed infinity, is a more controversial choice, since some would argue it is not always possible to determine what the correct sign should be, and that the result should be an unsigned infinity, that is, the projective model of infinity should be used (as was possible to do using implementations of an earlier draft of the Standard, such as the 8087 [50]). Still, a correctly signed infinity is a better choice than any other value, including NaN, which is intended to be used in cases in which any other value would be misleading [54].

If an implementation allows trap handlers to be associated with these exceptional situations, certain information must sometimes be made available to the trap handler, depending on which exception was signaled: in the case of the invalid operation and division by zero exceptions, no information is required to be delivered; in the case of the inexact exception, the default (that is, correctly rounded) result is to be delivered; in the case of overflow and underflow, the result to be delivered is scaled by a specified power of two in order to enhance the odds of being able to sensibly continue the computation[5]. The IEEE Standard recommends that other information, such as what operation was being performed at the time the exception was signaled, be made available to the trap handler.

A "sticky" status flag accessible to the user is associated with each of these exceptional situations, and records whether the corresponding exceptional situation ever occurred during the course of a computation. A conforming floating-point arithmetic engine never clears these flags unless the user or a language processor requests it. This allows for a (very) poor man's scheme for trap handling: all the status flags can be cleared before performing an operation. After the operation has completed, the status flag(s) can be examined to determine if an exceptional situation of interest occurred so that appropriate action can be taken. Needless to say, this scheme is most effective when its use is required only sparingly, because of the overhead in accessing the status flags twice for every operation[6], and because this scheme effectively inhibits any possibility of pipelining in most floating-point units.

8.2 Different architectures' support for exceptional situations

8.2.1 CISC architectures

Intel x86 architecture

The Intel x86 architecture has encodings for all the special values in all its floating-point formats (single, double, double extended). Some encodings in the double extended format are illegal[7];

[5]Though not a common feature among programming languages, the IEEE Standard does recommend that it be possible to continue a computation at the point in which an exception was signaled. Examples showing the usefulness of this capability appear later in this chapter.

[6]As will be seen below, in some implementations of the IEEE Standard, the status flags only need to be accessed once for every operation due to additional facilities being provided beyond those required by the Standard.

[7]These illegal encodings have to do with the fact that in the double extended format, the bit to the left of the radix point in the significand is explicit and is actually stored. If this bit is zero when it should be one, it is an illegal encoding. On the other hand, if this bit is one when it should be zero (that is, the value

when these encodings are encountered as operands, the invalid operation exception is signaled. (Thus, when the invalid operation exception is signaled, one cannot be sure without further investigation whether this occurred due to an exceptional situation described by the IEEE Standard, or due to some architecture-specific situation[8].)

Starting with the 80287XL [49], the behavior of Intel implementations of the x86 architecture fully conforms with the IEEE Standard in the area of special computational situations[9]. The way status flags are accessed is similar to the way the control register is accessed (see Chapter 4): the contents of the status register is first transferred to some memory location or to some other register, the relevant bits are manipulated, and, if desired, a new value is placed back into the status register. Storing a value into the status register may cause an exception to be signaled immediately, depending on which exceptions are unmasked. Significant overhead is incurred in storing an arbitrary value into the status register, since the only two instructions that can be used for this purpose also modify the control word and several other registers. However, there is a special instruction to clear all the status flags at once; another instruction initializes the floating-point unit, clearing all the status flags (and floating-point registers) in the process. It is possible to transfer the contents of the status register to the accumulator with one instruction, so finding out which exceptions have occurred during the course of a computation is a relatively fast operation. Still, adding the overhead of a function call to these operations, as would typically be the case when programming in a high level language, discourages making their use commonplace.

Trap handlers may be associated with the exceptional situations defined in the IEEE Standard with some software assistance and cooperation from the operating system. Each kind of exceptional situation can be masked or unmasked individually. If an unmasked exceptional situation occurs, the floating-point unit signals the system in order to interrupt the processor and cause it to execute a preestablished section of code (that is, an interrupt handler)[10], which, depending on the operating system, might only be specifiable by calling special functions. Enough information is kept to enable this section of code to determine what kind of exceptional situation was encountered, and which instruction was being executed at the time the exceptional situation occurred[11]. In addition, the operands are available if the invalid operation or division

is a pseudo-denormal, which, like true denormals, has the smallest exponent possible), the invalid operation exception is not signaled when encountered as an operand. Instead, it is treated as if it were a denormal (that is, the nonstandard denormal operand exception—a vestige from an early draft of the IEEE Standard, as are some of the illegal encodings—is signaled), but its value is the same as if its exponent were the second smallest possible. (In other words, it is implicitly normalized.) Perhaps a reason these illegal encodings have not been somehow "legalized" is that otherwise there could have been too much confusion over how early floating-point units (which were designed before the IEEE Standard was finalized) differed in their behavior compared to later floating-point units when executing the same program with identical inputs.

[8]Other situations not covered by the IEEE Standard which cause the invalid operation exception to be signaled happen when a value is pushed on a full floating-point register file (which is organized as a stack), or when a value is popped from an empty floating-point register file.

[9]Earlier implementations conform to an early draft of the IEEE Standard.

[10]Only one interrupt handler can be specified at a time to handle exceptional situations involving the floating-point unit. This interrupt handler can, of course, call other functions or procedures, depending on what kind of exceptional situation occurred.

[11]This assumes exceptions are not being signaled due to explicit manipulation of the status and/or control

by zero exception was signaled, or if the overflow or underflow exception was signaled and the destination was some memory location (rather than a register), provided in all cases that the corresponding exception is unmasked, and that subsequent non-floating-point instructions have not overwritten any operands. The kind of operation that was being performed and the destination's format can be deduced from the instruction being executed at the time the exceptional situation occurred.

Without additional software, Intel's x86 architecture does not conform to the IEEE Standard in at least one aspect: When converting from a floating-point format (which implies that the destination is some memory location), if the magnitude of the result is so large or so small as to be unrepresentable in the destination's format, even after scaling the result as described in the IEEE Standard, if the corresponding trap handler is enabled, no result whatsoever is delivered to the trap handler, in violation of the IEEE Standard. As mentioned above, the original operand is preserved, though, so it is at least possible for the trap handler to compute a suitable result for the operation. In any other situation, the result the IEEE Standard requires, if any, is delivered to the trap handler.

It is possible to write a trap handler that behaves as if the result of an operation is a given arbitrary value whenever a specified exception is signaled. One has to take care not to overwrite the operand(s) until one is certain it or they are not needed, and that the result is not used before one is certain it is available, since the floating-point unit may complete its instructions after subsequent (non-floating-point) instructions. In addition, one has to determine where the result was to be stored and what instruction should be executed next[12].

Motorola 68000 architecture

The Motorola 68000 architecture is similar in many ways to the Intel x86 architecture in the area of special computational situations. The following paragraphs mention some of the differences between these two architectures.

There are no illegal encodings in any of the floating-point formats, that is, interpretations are established for all possible encodings. However, some implementations of the 68000 architecture, such as the 68040 and 68060, cannot handle in hardware finite unnormalized values (that is, denormals or unnormals, the latter occurring only in the extended format) as operands.

In addition to the (sticky) status flags, the status register contains information on which exception(s), if any, occurred while executing the last floating-point instruction[13]. This improves

word registers.

[12]This instruction might not be the one following the floating-point instruction that caused the exception to be signaled because subsequent (non-floating-point) instructions may have already been executed, possibly due to the fact that the floating-point unit sometimes waits until the next floating-point instruction before signaling an exception.

[13]Events that cause the inexact and invalid operation exceptions are further subdivided, allowing exception handlers to be invoked only when specific situations occur, rather than always being invoked when the inexact or invalid operation exceptions are signaled. However, this leads to certain complications, since more than one bit needs to be set in the floating-point control register in order to always have the floating-point unit invoke a given exception handler whenever one of these (IEEE-defined) exceptions are signaled. Also, in the case of the invalid operation exception, several entries in the table of exception handler pointers need to be updated

the performance of the "poor man's" scheme for trap handling, since the status flags do not need to be cleared before every operation. Although no instruction is provided to specifically clear all the status flags at once[14], the overhead of storing an arbitrary value into the status register is not as significant as on the Intel x86 architecture, since there is an instruction in the 68000 architecture to do specifically this. Doing so never causes an exception to be signaled.

In some implementations of the M68000 family, such as the 68040 and 68060, an exception handler is always invoked whenever certain kinds of exceptional situations occur, such as overflow and underflow. This is because the hardware is incapable of producing a reasonable result in these cases. The exception handler can analyze the situation, and either generate the correct result if the user did not specify a (nondefault) exception handler (that is, the exception is masked), or invoke the exception handler the user specified.

8.2.2 RISC architectures

Hardware facilities in implementations of RISC architectures typically support only the single and double formats. Interpretations for all encodings in these formats are as specified in the IEEE Standard. In some architectures or implementations thereof, a trap is generated whenever certain special values are encountered as operands or supposed to be delivered as a result, which can be detrimental to the performance of some programs. Status flags are usually implemented in hardware. Given some cooperation from the operating system, it is usually possible to associate exception handlers with the various kinds of exceptional situations.

The following sections outline and comment on how some of the RISC architectures differ from one another in these areas.

Motorola PowerPC architecture

In some implementations of the PowerPC architecture, such as the PowerPC 604, if a certain bit is set in the Floating-Point Status and Control Register (FPSCR), denormalized numbers are treated as zeros when encountered as operands; if the IEEE Standard specifies that a denormalized number is to be delivered as a result, a zero is delivered instead. If this bit is not set, the behavior with respect to denormalized numbers is as specified by the IEEE Standard.

Although five bits in the Floating-Point Status and Control Register (FPSCR) correspond to the five status flags the IEEE Standard requires, the one corresponding to the invalid operation exception cannot be set or cleared explicitly. Instead, this bit is always set automatically anytime any of 9 (sticky) status "subflags" is set, and is clear otherwise[15]. A minimum of 5

whenever one wishes to notify the system which exception handler should be invoked whenever this exception is signaled. Finally, if a given exception handler is invoked for more than one kind of exception, more than one bit needs to be tested in order to determine whether the inexact or invalid operation exception occurred.

[14] Resetting the floating-point unit clears all the status flags at once, but doing so overwrites the floating-point registers as well. Furthermore, this can only be done in supervisor mode, which precludes doing so in normal programs without resorting to special system calls.

[15] In some implementations of the PowerPC architecture, such as the PowerPC 601, not all 9 "subflags" are implemented. However, because of the location of the "subflags" that are implemented within the FPSCR, the amount of time it takes to clear all 9 flags is not reduced because of the unimplemented "subflags."

instructions (for example, 5 instances of the mcrfs[16] instruction) are required to clear all the status flags. This is a relatively expensive operation, especially considering that the pipeline in the floating-point unit drains before an instruction accessing the FPSCR is executed.

Exceptions can either be ignored or signaled precisely or imprecisely, depending on the value of two bits in the machine state register (MSR). In the precise mode, the processor is able to identify the instruction that caused the exception to be signaled. In the imprecise mode, of which there are in fact two variations, this is not necessarily the case. When in the imprecise recoverable mode, it is possible to backtrack and determine which instruction caused the exception to be signaled, whereas in the imprecise nonrecoverable mode, this may not be the case. Furthermore, in the imprecise nonrecoverable mode, the operands of the instruction that caused the exception to be signaled may have been overwritten, and any results that instruction may have produced, even if erroneous, may have been used (or overwritten) in subsequent instructions. Therefore, the imprecise nonrecoverable mode is not compatible with the IEEE Standard. When an exception is signaled in the imprecise recoverable mode, it is not clear whether it is possible to determine which exception occurred, since more than one instruction may have signaled an exception. Note, however, that the IEEE Standard does not actually require this capability, though it does recommend that this be possible. Both the precise mode and the imprecise recoverable mode are compatible with the IEEE Standard.

Different implementations of the PowerPC architecture may not implement all four of these modes, but they will all least be capable of ignoring exceptions (as the IEEE Standard requires) and (one supposes) of signaling exceptions precisely. Performance may be degraded if exceptions are to be signaled precisely.

SPARC architecture

Although the SPARC architecture defines a quadruple precision floating-point format, many implementations of this architecture do not implement this format in hardware. There are encodings for all the special values in this format.

In some implementations of the SPARC architecture, if a certain bit is set in the Floating-Point State Register (FSR), arithmetic performed by the floating-point unit may not conform to the IEEE Standard. For example, denormalized numbers may be treated as zeros when encountered as operands; if the IEEE Standard specifies that a denormalized number is to be delivered as a result, a zero may be delivered instead. If this bit is not set, the behavior (for example, with respect to denormalized numbers) is as specified by the IEEE Standard.

In addition to the five status flags required by the IEEE Standard, there are five additional flags that indicate which exception(s), if any, occurred while executing the last floating-point instruction. As mentioned previously, this improves the performance of the "poor man's" scheme for trap handling, since the status flags do not need to be cleared before every operation. Modifying the status flags is a relatively expensive operation, since the only way to do so is to store the value of the FSR in some memory location, then load this value into an integer

[16] Mcrfs stands for "move to condition register from FPSCR." This instruction clears the bits in the FPSCR that were copied.

register, modify it, store it back to memory, and finally load the modified value into the FSR. Accessing the FSR causes the pipeline in the floating-point unit to drain.

In most implementations, floating-point exceptions are not signaled until the floating-point unit attempts to execute a subsequent instruction. However, the floating-point unit is required to maintain information about which instruction caused the exception to be signaled, provided the program establishes a trap handler for floating-point exceptions before any floating-point exceptions occur[17]. In this case, the floating-point unit also maintains information on any other floating-point instructions it has not yet finished executing, with the expectation that the trap handler will reexecute these instructions.

If a program establishes a trap handler for floating-point exceptions before any floating-point exceptions occur, it is possible to find out, should a floating-point exception occur, information such as which exception was signaled, what kind of operation was being performed, and what the operand(s) were. However, no results are ever delivered to the exception handler. Therefore, one cannot claim that the SPARC architecture conforms to the IEEE Standard unless one considers the established trap handler for floating-point exceptions to be part of the implementation of the Standard, and, in the case of the overflow, underflow, and inexact exceptions, that trap handler simulates the instruction that caused the exception to be signaled.

HP's PA-RISC architecture

HP's PA-RISC architecture is similar to the SPARC architecture with respect to special computational situations. One difference between these two architectures is that the former does not have flags that explicitly indicate which exception(s), if any, occurred while executing the last floating-point instruction. The remaining paragraphs in this section describe other differences between these two architectures.

In HP's PA-RISC architecture, floating-point exceptions are not signaled until the floating-point unit attempts to execute a subsequent instruction. However, the floating-point unit is required to maintain a list of instructions that it had already started, but could not finish, executing at the time it signaled the floating-point exception. This list of instructions includes the instruction that caused the exception to be signaled, any other instructions that would cause an exception to be signaled, and any instructions that cannot be executed to completion until some other instruction in the list finishes executing, possibly due to data dependencies.

Without additional software, HP's PA-RISC architecture does not conform to the IEEE Standard in at least one aspect: When converting from a floating-point format, if the magnitude of the result is so large or so small as to be unrepresentable in the destination's format, even after scaling the result as described in the IEEE Standard, if the corresponding trap handler is enabled, no result whatsoever is delivered to the trap handler, in violation of the IEEE Standard. The original operand is preserved, though, so it is at least possible for the trap

[17] If a floating-point exception is signaled before a program establishes a handler for floating-point exceptions, an implementation of the SPARC architecture is not required to maintain reliable information that would enable an exception handler to simulate the instruction that caused the exception to be signaled. If the program then subsequently establishes a handler for floating-point exceptions, the processor must not change its model for dealing with floating-point exceptions while that program is being executed.

handler to compute a suitable result for the operation. However, unlike Intel's x86 architecture, an exception different from any kind described in the IEEE Standard is signaled in these kinds of situations[18]. In any other situation, the result the IEEE Standard requires, if any, is delivered to the trap handler.

DEC's Alpha architecture

Of all the architectures discussed here, DEC's Alpha architecture provides the least support for special computational situations. Although it supports all the special values, whenever a special value other than negative zero is encountered, the invalid operation exception is signaled, and software must simulate the instruction if appropriate[19]. Similarly, if the result of an operation is supposed to be a non-finite value, the appropriate floating-point exception is signaled, and software is expected to determine what the appropriate result should be, if any. While it is not possible to mask the invalid operation, division by zero, and overflow exceptions, masking the underflow exception causes a positive zero to be generated whenever underflow occurs[20], even if the rounded result is supposed to be a denormalized number, negative zero, or a number whose magnitude is that of the smallest positive normalized number. When underflow occurs, the only way to obtain the result mandated by the IEEE Standard is to unmask the underflow exception, and have appropriate software generate the desired result. There is no provision in hardware to allow the decision as to whether the underflow and inexact exceptions should be masked or unmasked to be made at run time. It appears that the status flags in the Floating-Point Control Register (FPCR) are updated as the IEEE Standard requires. As might be expected, only unpredictable results are delivered to trap handlers.

8.2.3 How different architectures distinguish between signaling and quiet NaNs

In all architectures, except for HP's PA-RISC and DEC's Alpha, and all formats, signaling and quiet NaNs are distinguished by whether the most significant bit of the fractional part of the significand is a zero or one, respectively. In HP's PA-RISC architecture, signaling and quiet NaNs are distinguished by whether the most significant bit of the fractional part of the significand is a one or zero, respectively—the opposite of the other architectures. DEC's Alpha architecture does not make a distinction between signaling and quiet NaNs. Instead, Alpha processors invoke a trap handler whenever a NaN is encountered as an operand, or whenever

[18] Documentation from SunSoft [73], in apparent contradiction to that from Hewlett-Packard [5], claims that if a floating-point to integer conversion overflows the destination's integer format, the overflow exception is signaled. (Apparently, this is the result of system software [68].) While the IEEE Standard neither explicitly allows this behavior nor explicitly prohibits it, this is probably not the way the originators of the IEEE Standard intended the occurrence of this phenomenon to be signaled.

[19] Because exceptions are not signaled precisely, it can be a challenge to determine which instruction caused an exception to be signaled, especially if the sequence of instructions executed was chosen so as to maximize the performance of the program. In some cases, unless one chooses a sequence of instructions that is suboptimal in terms of performance, it may be impossible to determine which instruction caused an exception to be signaled.

[20] To be more precise, a positive zero is presumably produced when tininess is detected.

the IEEE Standard requires the invalid operation exception to be signaled. (There is no way to prevent the processor from doing so.) The system or user trap handler is expected to distinguish between signaling and quiet NaNs, and to generate a suitable result or take appropriate action in computational situations involving NaNs.

8.2.4 How different architectures detect underflow

In the PowerPC and SPARC architectures, tininess is detected before rounding, that is, by examining the magnitude of the infinitely precise result. In Intel's x86 architecture, HP's PA-RISC, and DEC's Alpha architectures, tininess is detected after rounding, that is, by examining the magnitude of the infinitely precisely result rounded to the precision of the destination in such a way as if there were no constraints on the range of the exponent. In all these architectures, loss of accuracy is detected as an inexact result, that is, in the same manner that they determine whether the inexact exception should be signaled. (The alternative way to detect loss of accuracy is to round the infinitely precise result as if there were no constraints on how small the exponent can be, and then compare this rounded result to what would have to be delivered to the destination, taking into account the actual constraints on the exponent. A denormalization loss occurs if the two rounded results are different.)

Incidentally, although Intel's literature does not divulge exactly how underflow is detected in its x86 architecture, it is possible to find this out by using the following technique: Let b and c be floating-point numbers whose precision is p bits, and let $\oplus$ denote a basic arithmetic operation on real numbers, such that

$$(1 - 2^{1-p})2^{e_{\min}} < a = b \oplus c \leq (1 - 2^{-p})2^{e_{\min}}$$

where a is a real number and $2^{e_{\min}}$ is the smallest positive normalized number whose precision is p bits. If the rounding mode in effect is "round toward positive infinity" (see Chapter 4) and all traps are disabled, the rounded result of this operation will be $2^{e_{\min}}$. Now, if tininess is detected after rounding, the underflow flag will not be set as a result of this operation, since the rounded result is a normalized number. However, if tininess is detected before rounding, the underflow flag will be set as a result of this operation, regardless of how loss of accuracy is detected, since the rounded result differs from both the infinitely precise result a, as well as what would be the result if there were no constraints on how small the exponent can be.

(It is also possible to find out how tininess is detected using the default rounding mode, "round to nearest or even," provided either that loss of accuracy is detected as an inexact result, or that it is possible to associate a trap handler with the underflow exception.)

To find out how loss of accuracy is detected, one can use the following technique: Let b and c be floating-point numbers whose precision is p bits, and let $\oplus$ denote a basic arithmetic operation on real numbers, such that

$$(1 - 2^{-p-1})2^{e_{\min}-p+1} \leq a = b \oplus c \leq (1 - 2^{-p} - 2^{-p-1})2^{e_{\min}}$$

where a is a real number and $2^{e_{\min}}$ is the smallest positive normalized number whose precision is p bits. (This assures that if the rounding mode in effect is "round to nearest or even," the

rounded result will be a denormalized number.) Furthermore, suppose a' is rounded (to nearest or even) to p bits (without regard to what the constraints are on the exponent), and that there are enough trailing zeros in the significand of a' to allow it to be exactly representable as a denormalized number. (The pair $b = 5 \times 2^{e_{\min}-3}$, which is a denormalized number, and $c = 1 + 2^{2-p}$ would fit these constraints if $\oplus$ denotes multiplication.) If the rounding mode in effect is "round to nearest or even" and all traps are disabled, the underflow flag will not be set as a result of this operation if loss of accuracy is detected as a denormalization loss, since the rounded result is exactly representable as a denormalized number. However, if loss of accuracy is detected as an inexact result, the underflow flag will be set as a result of this operation, regardless of how tininess is detected, since the rounded result a' differs from the infinitely precise result a, and the magnitudes of both a and a' are smaller than that of the smallest positive normalized number.

8.3 Handling special computational situations in high-level languages

There are a number of issues related to special computational situations that impact the design or implementation of a high-level language. This section discusses some of them, such as how to represent special values, how to support arithmetic involving special values and the signaling of exceptions, how to make status flags available, and when (or whether) they should be saved, restored, or cleared. In addition, this section will briefly mention some issues related to giving the user the ability to specify what the behavior should be when exceptional situations arise.

8.3.1 Ways of representing special values

There are cases in which one needs to explicitly represent a special value in a program. There are several ways a language design (or an implementation thereof) can make provision for this feature. There could be one or more special functions that return special values, though it may not be possible to use this or these functions in expressions that are to be evaluated at compile time. Another way to make this feature available is to have special named constants (for example, keywords or special strings or identifiers) that represent the special values, but which do not involve any computation. This is an attractive method because named constants are usually allowed in expressions that are to be evaluated at compile time, though, as will be seen in the next section, one may want to specify that one or more exceptions are to be signaled at the point in which a special value is used.

A third way in which special values may be represented is as an expression which, when evaluated, will yield the desired special value, and which can be evaluated at compile time. For example, infinity could be denoted by 1.0/0.0. The disadvantages are that expressions such as these are generally not as readable as using named constants, for example (using a macro preprocessor can help in this regard), and it is not clear whether any exceptions should or will be signaled when such an expression is evaluated[21].

[21] If such an expression is evaluated at compile time, exceptions might not be signaled, but they could if

Yet another way in which special values may be represented is as an attribute. (See the appendix for an example using attributes.) Conceptually, this has some similarity to calling a special function or to using a named constant, but no computation is involved, and a specific data type is associated with the specified value. Attributes are not convenient if one does not want a data type associated with a value, or if one wants one or more exceptions to be signaled at the point in which a special value is used.

In input/output routines, special values can take the form of special alphanumeric strings.

8.3.2 Handling static evaluation and numeric literals

Chapter 4 mentions some of the issues relevant to static evaluation and numeric literals in the context of special computational situations. As regards to special computational situations, one challenge is how to preserve the full semantics of the operations described in the IEEE Standard, including aspects related to the signaling of exceptions. For example, suppose a program contains floating-point expressions (or numeric literals) that, according to the language definition, must be evaluated at compile time. Furthermore, suppose that one or more exceptions (such as the inexact exception) are signaled during the evaluation of those floating-point expressions. When that program has executed a section of code that includes such an expression, should the status flags corresponding to those exceptions be set? Should exception handlers, if any, corresponding to those exceptions be invoked? Very few language definitions specify whether this should be the case, and, in fact, it is not trivial to do so, especially in a cross compiler, since such a compiler would, for example, have to faithfully replicate the manner in which tininess and loss of accuracy is detected in the target floating-point arithmetic engine. A further complicating factor is that the rounding and precision modes in effect can affect whether exceptions are signaled during the evaluation of a floating-point expression. Since setting status flags tends to be an expensive operation in many processors, and since in so many cases at least the inexact exception would be signaled when evaluating a floating-point expression, it may be more efficient to simply evaluate all floating-point expressions at run time, which would set the appropriate status flags as a byproduct, rather than attempt to evaluate some of them at compile time, and set the appropriate status flags explicitly. The same kinds of issues are relevant in the context of code simplifications, such as constant folding.

A different problem in which similar issues are important is that of designing a library of transcendental functions, such as sine and cosine, that is as compatible as possible with the IEEE Standard. The implementation of these functions should ideally produce final results that are as close as possible to the (infinitely precise) mathematical values. Furthermore, in all cases, these functions should ideally signal only the appropriate exceptions according to whether overflow, underflow, etc. would occur if the mathematical value were rounded (according to the rounding mode in effect) to the final result actually delivered to the caller, regardless of what exceptions were signaled while performing the calculations required to arrive at the final result. This problem is similar to that of static evaluation of floating-point expressions and numeric

evaluated at run time. One may not know ahead of time if a given expression will be evaluated at compile time or at run time.

literals in that there is usually a disparity between the description of what the "ideal" behavior should be (that is, that the appropriate exceptions should be signaled), and the behavior of actual implementations (that is, they typically do not signal any exceptions, or else they signal the wrong ones). In addition, devising an implementation that exhibits the "ideal" behavior is nontrivial, especially if good performance is desired.

As for numeric literals, this problem is exacerbated by the lack of a good way to represent special values, such as infinity and NaNs, in many programming languages. Programmers often resort to expressions such as 1.0/0.0 for infinity and 0.0/0.0 for NaNs. Even a value such as negative zero may be difficult to represent without resorting to a nontrivial expression, which may prevent such a value from being evaluated at compile time[22]. If expressions such as these were truly meant to represent special values, then no exceptions should be signaled when they are evaluated, but how is a language processor supposed to know this? In fact, expressions such as these are often used to explicitly cause an exception to be signaled, perhaps in addition to producing a special value. For example, an implementation of the exponential function may compare its argument with some suitably large number, and if its argument is larger than that number, it may want to return infinity to its caller and signal overflow at the same time. Writing 1e99999 or HUGE*HUGE, where HUGE is close to the overflow threshold, may be a convenient way to do this.

In some programming languages, special values may be virtually impossible to represent in static (floating-point) expressions, especially if implementations of a given language are required to stop processing when reaching a floating-point expression involving, say, overflow or division by zero.

8.3.3 Supporting arithmetic involving special computational situations

Some languages have a different notion of what should happen when special computational situations arise. One well known example is APL, whose definition proclaims that 0/0 is 1 (see, for example, section 7.2.4 "Divide" of [25]). Other language definitions (such as Limbo [59]) or implementations thereof demand that execution of a program stop if certain exceptional situations occur[23]. For example, before the advent of the IEEE Standard, it was quite common for execution of a program to stop whenever overflow or division by zero occurred. When a language is inherently incompatible with the IEEE Standard in this way, perhaps the best an implementation can do is to offer the programmer a choice as to whether the behavior specified in the language definition or the IEEE Standard ought to be followed. If a language definition does not forbid (nor require) the behavior specified in the IEEE Standard, then an implementation ought to provide this behavior.

[22] One way to represent negative zero is as f(x), where f denotes a function that multiplies its argument with (positive) zero, and x is a negative number. Simply writing -1.0 * 0.0 may not be satisfactory in all languages if one wants to be sure one gets a negative zero, regardless of which language processor is used.

[23] The IEEE Standard does not actually forbid this behavior, but neither does it require conforming implementations to be capable of stopping execution of a program whenever an exceptional situation occurs. In fact, the default behavior in a conforming implementation is to continue execution of the program after notifying the user (by setting the corresponding status flag) that an exceptional event occurred.

Some languages or implementations thereof include a standard library of functions. Some of the functionality in these libraries may overlap with that described in the IEEE Standard. For example, there may be a function that extracts square roots, or one that converts between decimal and binary floating-point numbers. In cases in which the IEEE Standard requires additional functionality not required by a language definition (such as support for NaNs), it may be possible (and convenient) to simply add this functionality in implementations of the language, as may be the case with conversion between decimal and binary floating-point numbers. Unfortunately, language definitions sometimes require additional functionality not required by the IEEE Standard (an example may be the square root function), or even worse, their requirements may conflict with those of the IEEE Standard. In such cases, it may be useful to have additional functions in the library that conform to the IEEE Standard. Such functions may even allow for more efficient implementations, particularly if additional functionality not required by the IEEE Standard is eliminated.

A number of code improvements that many language processors perform can lead to unexpected results when the target is an arithmetic engine compatible with the IEEE Standard [26]. For example, code such as:

```
if (x<>x) then f(x) end
```

where the type of `x` is some floating-point type, may seem nonsensical at first glance, but it would not be wise for the language processor to simply eliminate all traces of this code, unless it can ascertain that `x` cannot be a NaN.

In fact, eliminating any code at all can lead to unexpected results, since the programmer might be depending on the fact that the code in question will have certain side effects under certain conditions. For example, the subtraction in the expression `y:=x-0.0` cannot be eliminated without some thought, because the programmer might be using such an expression to ensure that the value of `y` will not be a signaling NaN (but instead a quiet NaN if `x` is a signaling NaN), or that `y` will be a negative zero in the case that `x` is a positive zero. Also, the user may be depending on the invalid operation exception being signaled whenever `x` is a NaN.

Another temptation to which many language processors succumb is to use registers containing values that have more (but hopefully not less) precision than what was specified in the program. This can cause different side effects than if arithmetic were performed with the precision specified. Using the example in the previous paragraph, if the format of `y` were the single format, and the value of `x` resided in a double precision register, simply using the value in the double precision register in subsequent arithmetic operations involving `y` could produce different results and cause different exceptions to be signaled than if the value of `x` converted to the single format were used, particularly if the value of `x` were too large to be representable in the single format. Of course, using more precision than is specified can produce more accurate results, but the programmer may not want extra accuracy at that point in the program.

8.3.4 Ways of allowing access to status flags

One way of allowing access to status flags is through special syntax, such as keywords and/or special identifiers. For example, suppose the special identifier `overflow` represented the status

flag corresponding to the overflow exception. The statement `overflow := false;` might mean that the status flag corresponding to the overflow exception is to be cleared. A potential problem with this approach is that it may be difficult to devise a way to manipulate (for example, test) an arbitrary combination of flags at once.

Another way of allowing access to status flags is through special functions. This has the advantage of typically not requiring the addition of a new feature to the language definition. However, unless a language processor has some knowledge of what these special functions do, it is unlikely that the use of these functions will be attractive in sections of code in which performance is critical. In addition, these functions may not get called as intended. For example, suppose one has code of the following form:

```
clear_status_flags ();
a := b + c;
status_flags := get_status_flags ();
```

where the function calls in the first and third lines of this fragment are intended to produce obvious effects, and a, b, and c denote floating-point quantities. If neither b nor c are accessible to called functions, such as the ones that appear in this fragment, a language processor may decide to arrange the addition to be performed either before or after calling either function, not realizing that functions can introduce subtle side effects in the execution environment[24] [39].

From an abstract point of view, it is ultimately desirable to have a way to represent a "volatile" set (that is, one in which members can be added as a byproduct of certain operations, possibly including function calls), along with various operations on that set, including union, intersection, and set difference. (In general, the members of this set would be those status flags corresponding to the exceptions, if any, that have been signaled since the last time the status flags were cleared, or, if the status flags have never been cleared explicitly, the beginning of the program.) However, choosing an overly abstract way of representing the status flags and operations on them runs the risk of causing programs that use these facilities to be too inefficient, since it may be difficult to devise an implementation that is sufficiently clever in processing such uses.

Similarly, it is not desirable if one ultimately has to resort to manipulating the status flags one at a time, since the performance of many programs will most likely suffer, given that accessing the status flags is a relatively slow operation in most floating-point arithmetic engines. Regardless of the method a language designer or implementer chooses to use to allow access to the status flags, the frequency with which the status flags in floating-point arithmetic engines must be accessed ought to be minimized. Section 8.6 gives examples of how this can be accomplished.

8.3.5 Ways of managing the status flags

Different languages and implementations thereof can offer differing amounts of automation as far as when and how the status flags are set and cleared explicitly, that is, not as a byproduct of

[24] Indeed, a language processor could decide to omit the addition if the given processor were to determine that the resulting sum is not needed later on in the program.

an operation. On the one extreme, status flags could be manipulated only when the programmer requests this explicitly. More generally, facilities for managing the status flags can be classified according to the granularity of the interval allowed between successive manipulations of the status flags, as measured by the amount of source code or run time. Chapter 4 discusses this in more detail in the context of facilities for setting rounding modes. In particular, such facilities can be characterized by whether the status of the status flags of the caller of a function affects which status flags are set and which are cleared when the called function begins execution, and similarly for the status of the status flags upon returning from a called function. In fact, such characterizations can be generalized to the point of being based on whether the status of the status flags before entering or leaving smaller blocks of code have any effect on the status of the status flags in the block of code about to be entered or resumed.

Given the point to which one can generalize this concept, one may wonder whether it is worthwhile implementing more than the simplest scheme possible for managing the status flags. Perhaps the most important reason for doing so is to allow one to conveniently view a group of (arithmetic) operations as if they were one large atomic operation. For example, computation in a trigonometric function can be decomposed into a series of basic arithmetic operations. When calling such a function, one is not normally interested in knowing whether an exception was signaled while computing an intermediate result. Rather, one would more likely be interested in knowing whether an exception would be signaled if one were to round the (infinitely precise) mathematical value to fit the destination format. In fact, in some cases, the actual computation of some function may not cause any exceptions to be signaled, and yet the signaling of one or more exceptions really ought to accompany the final result returned to the caller. Consider, for example, division of a sufficiently large (but finite) dividend by a sufficiently small interval that straddles zero. Regardless of what the model of infinity is (projective or affine), the final result can be arrived at by simply noticing how large the dividend is, how small the magnitude of the divisor is, and the fact that the divisor straddles zero, that is, the final result can be arrived at with only a few comparisons, none of which would signal any exceptions. And yet, in such a case, it would be appropriate to signal both the division by zero and overflow exceptions.

Another possible use for facilities related to managing the status flags is as a crude debugging technique. One may want to find out, for example, if a certain exception is ever signaled in a certain section of code, or, more generally, one may want to find out at what point (or points) an exception is signaled. On a system in which it is not convenient to stop execution of a program when a given exception is signaled (a capability which the IEEE Standard does not require), one could clear the status flags (possibly after saving them) before entering a section of code of interest, and examine them upon leaving that section.

In language implementations in which it is possible to associate (and disassociate) exception handlers with floating-point exceptions, yet another use for these facilities is to control which blocks of code can cause an exception handler to be invoked. Consider a hypothetical language implementation in which exception handlers are invoked whenever the appropriate status flag(s) are set, and in which called functions do not inherit their callers' exception handlers. Let us further assume that a called function can affect which status flags are set upon returning to its caller. One may want to prevent an exception handler from being invoked upon returning to

the caller, because the exceptions, if any, that are signaled while executing the called function may not be relevant. This means that the appropriate status flag(s) need to be cleared before returning to the caller. (One way to accomplish this is to restore the caller's status flags.)

8.3.6 Facilities for exception handling

It is outside the scope of this work to exhaustively discuss facilities for exception handling. Hauser discusses this subject at greater length [39]. However, a few comments are in order.

In some languages, when an exception is raised and there is a corresponding exception handler, that handler is invoked, and control never returns to the point at which the exception was raised. There certainly are cases in which this kind of behavior is desirable. For example, in Gaussian elimination, if division by zero ever occurs, there may be no sense in completing the computation, since this is an indication that, for practical purposes, the matrix is singular. An exception handler that causes the procedure to return to its caller could be appropriate in this case.

However, there are other cases in which this kind of behavior is not desirable. For example, suppose one is computing the Euclidean norm of a vector whose components have large magnitudes. If overflow occurs while squaring the components or adding the squares of the components, it would be desirable to be able to adjust the sum obtained so far by a suitable scaling factor, recompute the operation that overflowed using the new scaling factor, and continue the computation at the point of interruption, again using the new scaling factor. The final result could then be adjusted by this scaling factor. Even if overflow occurs during the intermediate calculations, the final (adjusted) result could still be within the range of representable finite numbers. Note that in this example, the exception handler should ideally have access to local variables, so an exception handling facility in which exception handlers are merely nonnested functions would not be ideal.

An even simpler kind of exception handling capability can be useful in some cases. Consider computing $(\sin x)/x$. As x approaches zero, the (mathematical) value of this expression approaches one. However, if this expression were computed using an arithmetic engine compatible with the IEEE Standard, and no traps are taken (for example, they are disabled), the result would be a NaN if x is equal to zero. In such cases, it would be convenient if one could specify that if a given exception is signaled when a particular operation is performed, a nonstandard result different from that specified by the IEEE Standard (one in this example) should be produced; the exception handler need not do anything else.

Another capability that is important is to be able to enable an exception handler for a given type of exception in some parts of a program, and be able to disable all exception handlers for that type of exception in other parts of the program. For example, suppose one normally wants execution of a particular program to stop whenever the invalid operation exception is signaled. There may be sections of code in the program, however, in which one knows that if the invalid operation exception were to be signaled, producing a NaN as a result would be perfectly acceptable, and it would therefore not be worthwhile to stop execution of the program. In fact, if one had to ensure that the invalid operation exception did not occur in these sections of code, performance could suffer significantly. Thus, the ability to disable all exception handlers for

the invalid operation exception in these sections of code would be convenient.

Kahan proposes an exception handling scheme that provides only a limited set of choices as to what actions can be taken when an exception is signaled [54]. These include aborting execution of the program, invoking the debugger, replacing the result with a given value, incrementing a counter (so that in the case of overflow or underflow one can find out how many times a result has been scaled), and producing the default value specified by the IEEE Standard.

Hull et al proposes another exception handling facility appropriate for systems in which exceptions are raised imprecisely, that is, possibly at a later point than that at which the exception actually occurred [44]. This scheme has the advantage of being relatively easy to implement on a system that fully conforms to the IEEE Standard, since it could basically involve clearing the status flags at the beginning of a block of code, and checking them at the end of the block of code to determine if an exception handler should be invoked. The disadvantage of this scheme is that some efficiency can be lost due to either unnecessarily proceeding with a computation past the point in which an exception occurs, or having to recompute portions of a computation prior to the point in which an exception occurred because of the fact that one cannot be sure exactly how much of the computation was completed before the exception occurred.

As with ways of managing the status flags (see the previous section), facilities for exception handling can be classified according to the granularity of the interval over which an exception handler can be enabled (and disabled), as measured by the amount of source code or run time.

8.4 Issues related to mixed-language programming

Many potential minefields in the area of special computational situations await those who dare to engage in mixed-language programming, or even use more than one language implementation to process different parts of the same program. Only a few of these potential pitfalls are mentioned below.

Some language processors may not be prepared to deal with special values, such as infinities or NaNs. Therefore, if a procedure is processed with an IEEE Standard-aware language implementation, and that procedure calls another that is processed by a language implementation that is not prepared to deal with special values, that other procedure may produce wrong results if any special values are included among the parameters. For example, that other procedure may produce a certain value if a parameter is less than some value. The language processor may reverse the condition, and cause the procedure to produce the given value if the parameter is not greater than or equal to the specified value. If the parameter happens to be a NaN, the wrong value will likely be produced. Even worse, execution of the program may end unnecessarily when the other procedure encounters a special value.

8.5 What support exists in high-level languages

8.5.1 Existing language designs

As with the operations described in the IEEE Standard (see Chapter 7), one can likewise consider what support exists for special computational situations in the definition of a high-level language, as compared to what support can be provided by an implementation that fully conforms to that definition.

Except for prohibiting negative signed zeros in formatted output records, Fortran 90 [32] makes no reference to any special values. According to this standard, "any numeric operation whose result is not mathematically defined is prohibited in the execution of an executable program." This basically implies that execution of a program translated by a translator conforming to the Fortran standard must not continue past a point in which the invalid operation or division by zero exception would be signaled. Finding a way to portably represent a NaN would thus be problematic.

The ISO C Standard [16] also provides little support for special computational situations. This standard allows, but does not require, positive infinity to be represented (as an expression of type double) by the macro HUGE_VAL. There is no provision for representing any other special values. An implementation's behavior in the presence of an expression that is mathematically undefined is up to the implementer. Thus, there is no way to portably represent any of the special values (since HUGE_VAL is not guaranteed to represent infinity, and since this standard does not dictate any correspondence between its types and the formats defined in the IEEE Standard), or to access such features as the status flags. Despite this, there is nothing in this standard that would prevent an implementer from fully supporting the IEEE Standard in the area of special computational situations.

Besides exception handling being part of the language, C++ does not provide any additional support for special computational situations [15]. Note, however, that, compared to the C language, special values, such as infinity and NaNs, cannot be as conveniently represented as constant expressions, since implementations are required to issue at least one diagnostic message for programs containing numeric literals such as 1e9999 (which one may be tempted to use to represent infinity), or constant expressions that are mathematically undefined, such as 0.0/0.0 (which one may be tempted to use to represent a NaN).

Even Ada does not provide significantly better support for special computational situations. In fact, unlike C++, once an exception handler is invoked, control cannot be transferred back to the statement in which the exception was raised. However, also unlike C++, there are attributes that can be used to find out at run time if denormals and signed zeros (with the semantics described in the IEEE Standard) are supported. There is no provision in Ada's model of floating-point arithmetic for nonfinite values, such as infinity and NaNs, though any of these values may be produced under appropriate circumstances as the result of an expression if the value of the attribute Machine_Overflows is false. The invalid operation, division by zero, and overflow exceptions all roughly correspond to Ada's predefined Constraint_Error exception, but the latter does not have to be raised when any of the former types of exceptions occurs if the value of the attribute Machine_Overflows is false.

8.5.2 Compilers for existing languages

Sun's C and Fortran compilers provide library functions that return specified special values, which can then be assigned to variables at run time [73]. Another library function provides access to the processor's floating-point status register, so that the status flags can be manipulated. Yet another library function allows one to enable and disable exception handlers[25]. The performance penalty of using these two library functions is relatively high, since some of their arguments are character strings whose meaning must be decoded, rather than numerical values that can be used with little or no manipulation. Also, it is not possible to find out and also modify the status of the status flags with one (library) function call, so if one wishes to perform both of these actions with the facilities provided, one has to call the same function twice, even though this function must first find out what the status of the processor's status flags is in order to modify them. So, while it is possible to access the IEEE Standard's features related to special computational situations, the facilities provided in this area are fairly primitive, at a low level, and not terribly efficient.

8.5.3 Extensions to existing languages

The Standard Apple Numerics Environment (SANE) provides a function which returns a quiet NaN [12]. The C implementation of SANE provides another function which returns infinity; in the Pascal implementation of SANE, INF is a predefined constant whose value is infinite. The I/O routines support infinities and quiet NaNs. Other functions or procedures get or set individual status flags, or enable or disable individual traps. In some implementations, it may be possible to associate an exception type with an arbitrary exception handler. Procedures are provided to save or restore the entire floating-point environment, including all five status flags, perhaps subsequently clearing all the status flags or setting the status flags that were set before the procedure was called.

In terms of support for the special computational situations described in the IEEE Standard, the extensions in [75] are similar to those of SANE, except that macros are used to represent nonfinite special values (these macros may expand to function calls, however, so they cannot be portably used in constant expressions), more than one status flag can be manipulated at a time, and no facilities are provided to deal with trap handlers. Interestingly, there is a function which explicitly signals specified exceptions, and another one that is intended to merely restore the state of the status flags to one that was previously saved. Thus, so as to avoid invoking any trap handlers before returning to its caller, on many processors, the latter function may have the side effect of disabling any traps the user may have explicitly enabled, perhaps by calling a special function to do so.

[25] Exception handlers are simply arbitrary functions that accept a certain number and type of arguments. One of those arguments can be examined to determine which instruction caused the exception to be signaled. Returning from an exception handler resumes the computation at the point of interruption, perhaps with an unexpected result used in place of that which should have been produced by the operation that caused the exception handler to be invoked. Of course, it is possible to write an exception handler that determines what that operation was, and ensures a reasonable result is used when the computation resumes.

8.5.4 New language designs

The Java language [38] does not provide any support for signaling NaNs, but it does provide ways of representing all other nonfinite special values as constant expressions. There are no facilities for accessing the status flags or for dealing with trap handlers related to floating-point arithmetic. In situations in which the IEEE Standard requires an exception to be signaled, the result produced is that specified by the IEEE Standard when no traps are enabled. A floating-point numeric literal that overflows, or whose magnitude is too small to be represented as a denormalized number, causes an error at translation time. I/O facilities support all special values except for NaNs on input and signaling NaNs on output.

At least a couple of efforts are under way to improve Java's support for special computational situations, among other things. RealJava [21] adds facilities for accessing the status flags. At present, it is difficult to obtain information on Borneo [22], the other effort to improve Java's support for numerical programming.

Limbo [59] provides constants for positive infinity and a quiet NaN. Support for denormalized numbers may be absent in some implementations. Arithmetic involving special values otherwise conforms to the IEEE Standard. Functions are provided to access the status flags; other functions can be used to enable or disable traps, or to find out if traps corresponding to specified exception types are enabled or disabled. By default, any kind of exception other than the inexact exception causes execution of a program to stop.

μln [30] provides constants for positive infinity and a signaling NaN. Special keywords provide access to individual status flags. The status flags are automatically saved and then cleared when a function is called, and status flags corresponding to exception handlers that are enabled in a compound statement are saved and cleared when execution of that compound statement begins. If any status flag was saved when execution of a compound statement began, the status flags that were saved are automatically updated upon exiting that compound statement. That is, if a status flag is set when execution of the compound statement ends, or if a status flag was set when the status flags were saved, it remains set after restoring the status flags that were saved. Similarly, the status flags of the caller are automatically updated when a function returns to its caller.

In μln, by default, execution of a program terminates when the invalid operation, divide by zero, or overflow exception is signaled, a warning message is printed when the underflow exception is signaled, and the inexact exception is ignored. A trap can be explicitly disabled by using the keyword ignore in an exception handler. The scope of a given exception handler is the innermost compound statement of which it is a part. Special functions are available within an exception handler to get detailed information on what caused the exception, such as what kind of exception was signaled, what the operands were, and what kind of operation was attempted. In addition, one can request in an exception handler that a given value, which need not be constant, replace the result of an arithmetic operation, or that execution of the entire compound statement be restarted (presumably after taking corrective action). Alternatively, an exception handler can arrange to have control return to the function's caller, and to resignal the exception in the context of the caller. Of course, a language with these kinds of facilities would be quite challenging to implement for certain target processors.

8.6 What support should exist in high-level languages

There is no excuse for not providing in a high-level language a way to represent all the special values mentioned in the IEEE Standard, since this can be accomplished, for example, by providing one or more functions, each of which returns a given special value. Ideally, though, it should be possible to represent all the special values in a way that allows them to be used in any context in which a floating-point constant expression is allowed, and that does not involve signaling any exceptions or issuing diagnostic messages. In the case of an existing language, this may require an extension to the language. In addition, input and output routines ought to be provided to allow one to convert any of these special values between their internal and external representations. Again, this may require extending the language in some cases.

Facilities for accessing the status flags ought to also be ubiquitous, since, again, this can be accomplished by providing special functions for this purpose. Now, accessing a floating-point arithmetic engine's status flags causes pipeline stalls in many cases, so some care should be taken to design facilities that allow one to minimize the number of times the engine's status flags are actually accessed. For example, in many cases, one may want to save the status flags and subsequently clear them. On many processors, in order to clear the status flags, one has to obtain the value of the register containing the status flags, clear certain bits in that value, and then store the modified value back into this register. Thus, having a function that provides the values of (selected) status flags and then clears them is preferable to always having to call two distinct functions: one to obtain the values of the status flags, and one to clear them. As another example, one may want to set certain status flags and clear all others. Being able to do this by calling just one function can decrease the number of times the engine's status flags are actually accessed. Yet another way to decrease the number of times the engine's status flags are accessed is to provide a way to access more than one status flag at a time, so the engine's status flags do not have to be accessed once for each status flag.

As mentioned in section 8.3.5, another important capability involving the status flags is the ability to make an arbitrary group of statements appear as if it were an atomic operation, at least from the point of view of which status flags are set, and what exceptions are signaled. Some automation beyond simply providing functions to obtain the status of the status flags and set and clear them would be helpful, since otherwise a typical sequence of operations would be something like this:

1. obtain and save the status of the status flags;
2. clear the status flags;
3. disable all traps temporarily;
4. perform the given sequence of statements;
5. restore the status flags that were previously saved;
6. enable the traps that were disabled temporarily, if any;

7. set the flags corresponding to the exceptions, if any, that would be signaled if the sequence of statements were truly an atomic operation, the expectation being that if a trap is enabled for at least one of these exceptions, the appropriate exception handler, if any, will be invoked immediately.

If the sequence of statements were the entire body of the function, it may not be convenient to always exit the function from the same point in the code. If this is the case, code for steps 5–7 above would need to be duplicated more than once.

The question of what sort of automation in the management of the status flags would be appropriate (and practical) is difficult to answer, particularly if one wants to make an entire function appear as an atomic operation. One of the factors that contributes to this difficulty is that in essence, all the operations described in the IEEE Standard return a composite value: a numerical result and a set of exceptions that were signaled either while converting the infinitely precise result to a representable value, or because the result is (potentially) ill defined.

If one can be assured that no traps are ever enabled, then this problem becomes a bit easier to deal with, because then the point in which one signals the exceptions that are appropriate to the sequence of statements that one wants to consider as an atomic operation is not as critical[26]. Also, if it is not important to be able to specify a given value as being the result of a sequence of statements, and also virtually simultaneously signal one or more exceptions (as happens with the basic arithmetic operations), the problem can be simplified.

Even if this problem is simplified as mentioned, automation in the management of the status flags is likely to require an extension in the case of an existing language. For example, one could envision extending an existing language with a pragma having semantics such as: Save the status of the status flags, and optionally clear the status flags and disable all traps temporarily; perform the sequence of statements with which this pragma is associated; when control leaves this sequence of statements for any reason, update the saved status flags with the flags that are now set, restore the updated status flags, and enable any traps that were disabled temporarily. Of course, all this could also be accomplished less conveniently by using a wrapper function to implement the functionality of the (hypothetical) pragma, provided the sequence of statements in question can be encapsulated in a separate function.

Sometimes, in addition to making a sequence of statements appear as an atomic operation, one wishes to find out what exceptions were signaled by that "atomic operation," possibly without actually invoking any exception handlers. (In fact, this may be desired in cases in which "that 'atomic operation'" is a basic arithmetic operation.) In other words, taking the view of operations returning composite values, as mentioned above, one sometimes wants to obtain the full composite value, and not just the numerical part of the result.

Perhaps a more generalized paradigm that incorporates these ideas is to view arbitrary sequences of statements or operations as instances of functions of the form:

operation (operands, exception_type ⇒ action_or_value ...)

[26]If the status flags are actually implemented as, say, pointers to the point in which the associated exception type was signaled, with a null pointer representing a flag that is clear, then it could potentially be more critical exactly when an exception is signaled. However, in actual practice, if an exception handler is not invoked, knowing only approximately where an exception was signaled is usually good enough.

where operation denotes a given sequence of statements or operations, operands denotes the operand or operands used in the sequence of statements or operations, exception_type denotes one (or more) of the exception types described in the IEEE Standard, and action_or_value indicates whether an exception handler associated with the given exception type should be invoked or not, or, in the case of the given exception being signaled, if some other value (ideally computed only if the given exception type is signaled) should replace the result that would normally be computed by the sequence of statements or operations. action_or_value could alternatively be any of the choices available in Kahan's proposed exception handling scheme (see section 8.3.6. More than one exception_type, action_or_value pair could be associated with a given operation. The return type of such a function would be a composite value consisting of a numerical result and the set of exceptions signaled by the function (regardless of whether the signaling of these exceptions is ignored or not). Note that the ability to return a composite value does not abolish a need for the "sticky" status flags described in the IEEE Standard; these flags could continue to be updated, for example, as described in the above discussion of the hypothetical pragma.

One of the possible applications of this paradigm is to provide a way to control exactly at what points exceptions are to be raised in a program written in a language like C++ or Ada that has exception handling facilities. In these languages, predefined operators are restricted in terms of what exceptions can be raised while performing the operations they denote, and the corresponding language standards do not mention (at least some of) the exception types described in the IEEE Standard as being among those that can be raised. Many times one is only interested in what exceptions might occur in but a few places in a program. By writing these operators using functional notation, one can visually mark the points in which one wants exceptions to be raised if warranted, and specify exactly which exception types are of interest for each individual operation.

Of course, this paradigm is also applicable to sequences of statements or operations that one wishes to view as an atomic operation, whether or not they can be encapsulated in a function, though it may be less convenient to apply this paradigm to a sequence of statements or operators that is not encapsulated in a function, since the technique of using a wrapper function to specifically provide support for handling exceptional situations would no longer be applicable. Note also that this paradigm provides slightly more control than that proposed by Hull et al [44] (see section 8.3.6) because it at least provides information on what types of exception occurred, if any, but at the same time, this paradigm retains its suitability to systems which lack precise traps or the ability to enable and disable traps.

As mentioned in section 8.3.3, many techniques that language processors use to improve code are problematic from the point of view of support for special computational situations. For example, given the expression f(x*0.0), it would be tempting to evaluate the parameter at translation time, rather than at run time. But if x is not finite, or if x is negative zero, the value of the parameter would not be (positive) zero. Even if x were a numeric literal such as 1e999 (which represents a value that is too large to be represented in the double format, but is not too large to be represented in the double extended format), it would still not be clear, particularly in absence of clear guidance from the language definition, how to best translate this

product. Should overflow be signaled (assuming the parameter is intended to be representable in the double format), and then should the invalid operation exception be signaled, or should both of these exceptions be signaled simultaneously (because the parameter was evaluated at translation time), or should one or none of these exceptions be signaled, or should the parameter be evaluated partially or entirely at run time so that exceptions are signaled more properly? If the parameter is at least partially evaluated at run time on a system in which floating-point parameters are passed in double extended registers, should the multiplicand first be converted to a value representable in the double format (that is, positive infinity), or should parameters not be converted at all, or should only the final value of the parameter be converted? (A different example would show that the evaluation technique used could make a difference.)

Other code improvement techniques are also problematic. For example, code motion, including moving loop invariants outside a loop, can cause exceptions to be signaled spuriously or at the wrong time. Even a simple fragment such as the following illustrates the hazards of code motion (and common subexpression elimination):

```
y := x * 0.0; f(); y := x * 0.0;
```

Even assuming that floating-point variables x and y cannot be accessed by the procedure f(), it is questionable whether it is safe to postpone the first multiplication until after procedure f() is called, or even eliminate the first multiplication altogether, since proper behavior of procedure f() may depend on the status of the status flags. (For example, procedure f() may perform some corrective action depending on what exceptions, if any, have occurred.) It is similarly questionable whether the second multiplication can be safely eliminated or performed before calling procedure f(), since procedure f() may clear the status flags, and proper behavior of code after the above fragment may depend on the status of the status flags.

Not only does the format used in evaluating an expression affect which exceptions are signaled, but the instruction used is significant as well. For example, some processors, such as implementations of the PowerPC architecture, have instructions which multiply two values and add the resulting product to a third value, with rounding occurring only once after the sum has been produced. When these instructions are used, not only can results differ, but which exceptions are signaled, if any, can differ from what would be the case if products and sums were computed by distinct instructions.

All the above tends to support the conclusion that nearly all floating-point expressions are volatile, in the sense that they cannot be removed as superfluous code, eliminated as common subexpressions, moved around, or even evaluated at translation time, due to possible side effects that are not obvious. (Probably the only law of which one can take advantage without concern for possible side effects is the commutative law for addition and multiplication[27].) It is therefore critical to be able to indicate (implicitly or explicitly) which sections of code depend on exceptions being signaled accurately, and in which sections of code there is no need to be concerned with which evaluation formats or instructions are used to compute floating-point

[27] Multiplication by one can also be eliminated without concern for possible side effects, provided one can show that neither of the operands can be a signaling NaN. This is not hard to show in most cases, since signaling NaNs cannot easily propagate through a computation.

expressions, or whether any exceptions at all will be signaled while computing these expressions, or whether any operands are special values, or even what the rounding or precision modes in effect are (see chapters 4 and 5).

In general, the more information the language processor has about what exceptions can and cannot be signaled while executing a given section of code, and whether these are at all relevant, the greater the chance that it might be possible to detect a way to safely improve the efficiency of the code. In particular, if one were to use the paradigm mentioned in this section, the language processor may be able to determine that whether certain exceptions are signaled or not is not important, perhaps because the given sequence of statements or operations mentions what exceptions should be "exported" when execution of the given sequence is finished, and no exceptions that are signaled while executing the given sequence matter. For example, an implementation of the cosine function may not care what exceptions are signaled while the return value is being computed. It may simply produce a result and possibly signal the inexact or invalid operation exception. In such a case, it may be safe to apply some code improvements that alter what exceptions are signaled, and when they are signaled.

Chapter 9

Floating-Point Expression Evaluation Schemes

A study of the impact the IEEE Standard has on programming languages would not be complete without a discussion of floating-point expression evaluation schemes, that is, how floating-point operations and operands map to instruction sets of target processors. There are several ways to characterize such schemes [77, 26, 55], but basically, floating-point expression evaluation schemes can be categorized as to whether they are predictable, that is, whether a reasonably competent programmer can determine how the floating-point operations and operands in a given piece of source code (with sufficient context) are mapped to specific operations on (possibly arbitrary) target processors.

Both predictable and unpredictable schemes can be further categorized. For example, unpredictable schemes can be characterized as to whether they tend to make efficient use of a processor's capabilities. Predictable schemes, on the other hand, can be characterized as to whether they are portable (that is, applicable to a wide range of processors), or processor specific. A special class of portable schemes are those whose aim is to provide for bit-for-bit identical results on a variety of target processors. (Section 9.3 discusses this in greater detail.)

Before giving specific examples of some predictable expression evaluation schemes, one distinction that merits mentioning is the relationship (or absence thereof) between the data types of arbitrary floating-point expressions and how these expressions can actually be evaluated. Many language definitions associate floating-point expressions with data types for the purpose of semantic type checking. The particular data type with which a floating-point expression is associated typically depends on what the operator or function call is, the data type(s) (and number) of its operand(s) or argument(s), and possibly the context in which the operator or function call appears (such as its expected type). For example, in the case of a binary operator, the data type of its result is often the wider of its operands.

Many programmers (and perhaps language designers as well) assume that an expression's associated data type somehow influences the precision with which the operation (or function call) is computed. However, most language definitions stop short of specifying whether a programmer can actually rely on an expression's associated data type to determine to what precision that

expression will be computed. As a result, conforming implementations can (and sometimes do) evaluate floating-point expressions in unexpected (and sometimes seemingly arbitrary) ways. Indeed, a fully-conforming (but brain-damaged) Fortran [31] compiler could assert that $2 + 2$ is 5! Less worrisome, but still potentially troublesome on occasion, is the case of an expression being evaluated more precisely than expected [66].

9.1 Predictable expression evaluation schemes

There are potentially many different predictable expression evaluation schemes. However, some of the most popular are strict evaluation, widest available, and widest needed [26]. These can be more easily understood by considering an expression such as `d + (s * s)`, where the type of `d` corresponds to the double format, and the type of `s` corresponds to the single format.

In strict evaluation, the result of a binary operation is computed to the precision of that of the wider of its operands, and, if necessary, the narrower of its operands is converted to the format corresponding to this precision. The result of a unary operation is computed to the precision of that of its operand; no conversions are necessary. In the expression mentioned above, the multiplication would be performed in single precision, and the product would subsequently be converted to double precision. Finally, the addition would be performed in double precision.

In the "widest available" scheme, all operands in an expression are converted to the widest floating-point format available on the target machine, and all operations are computed to the precision corresponding to this format. In the expression given above, for example, both s and d would be converted to the double extended format if the target machine were one conforming to the Intel x86 architecture, and both the multiplication as well as the addition would be performed in double extended precision.

In the "widest needed" scheme, all operands in an expression are converted to the format corresponding to the widest floating-point data type used in the expression, and all operations are computed to the precision corresponding to this format. In the expression given above, for example, s would be converted to the double format, and both the multiplication as well as the addition would be performed in double precision.

Variations on these schemes are possible, of course [75]. For example, subexpressions not involving operands wider than that corresponding to the double format could be evaluated in double precision, while operations having at least one operand in double extended precision could be evaluated in this wider precision.

9.2 Advantages and disadvantages of various evaluation schemes

Some of these schemes favor certain classes of target machines. For example, in terms of performance, strict evaluation favors machines in which operations involving only the single format are faster than operations involving wider floating-point formats, or in which conversions to wider floating-point formats decrease performance, such as SPARC-based machines.

On machines conforming to the Intel x86 architecture, strict evaluation and the widest needed scheme lead to relatively poor performance, especially if the types of variables within expressions or in different expressions within a basic block correspond to different floating-point formats, with the double extended format being used in some (but not necessarily all) of the expressions[1]. The widest available scheme suits these machines much better, since these machines tend to perform better when the precision to which operations must be computed does not change very often.

The widest needed scheme does not particularly tend to favor any particular class of target machines more than other schemes. Rather, its attractiveness stems from other considerations, such as its applicability to a more generalized floating-point system consisting of a wide variety of floating-point formats (such as multiple precision floating-point systems), and its balance between protecting naive programmers from specifying a precision that is too narrow for a given application, and to what degree it tends to produce unexpected results due to its inducing the use of wider precision than is readily apparent from the source code. Unfortunately, use of the widest needed scheme tends to reduce performance on more than one class of machines, for example SPARC-based machines and machines that conform to the Intel x86 architecture. (The paragraphs above explains why this is the case.) Performance is acceptable on machines such as those conforming to the Alpha or PowerPC architecture, whose floating-point registers support the widest floating-point format available, and which have instructions to perform arithmetic in less precision than that of its floating-point registers.

As has been hinted at above, strict evaluation proffers the user very little protection against recondite mistakes related to the precision of operands or choice of formulas on which computations are based, while the widest available scheme provides the most protection against such mistakes. (The standard quadratic formula is a classic example of a case in which use of extra precision enables one to obtain better results: strict evaluation can cause all digits of accuracy to be lost in the expression $b^2 - 4ac$ when the minuend and subtrahend are nearly equal, whereas the widest available scheme might at least allow one to determine the sign of this expression, providing the precision of the variables a, b, and c is less than the widest precision available.)

On the other hand, the semantics of strict evaluation tends to be more intuitive to programmers who are not experts at numerical programming, while the widest available scheme can cause surprising results to be produced in certain situations in which the extra precision it affords is unexpected, especially for naive programmers. In addition, certain algorithms, such as some that simulate higher precision, require that a very specific precision be used in computations—extra precision will cause these algorithms to produce nonsensical results. (Sun's fdlibm, a freely available math library, is an example of such code [72].) For this reason, if the widest available scheme is used, some means ought to be provided to allow an alternative scheme to be used where needed.

As can be seen, none of these schemes fits everyone's needs, and none of them enhances

[1] As noted in chapter 6, this is because on such machines, basic arithmetic operations cannot be computed in any precision other than double extended, unless one resorts to modifying a special register, which is time consuming and tends to disrupt the floating-point pipeline. One could alternatively save the result of every floating-point operation to memory using the appropriate format, and then reload results that are subsequently needed in further computations, but this too is time consuming and tends to disrupt the floating-point pipeline.

performance on all machines. So if a language definition were able to accommodate at most one of these schemes, the language designer would face a difficult choice indeed! Furthermore, if speed is the overriding concern, with quality of the arithmetic being a very secondary concern (as would be the case in some graphics applications), none of these schemes is likely to be very appealing, except perhaps for a specialized class of machines: witness Sun's dilemma in coming up with a way to allow better floating-point performance in implementations of Java for machines conforming to the Intel x86 architecture [60].

9.3 Bit-for-bit identical results and the IEEE Standard

Being able to obtain bit-for-bit identical results, regardless of the machine used to perform computations, is attractive to many. One of the reasons is that it is costly to validate the results obtained on every machine on which a program might be run. (In some safety critical applications, different results from those obtained on some "standard" machine are equated with wrong results, even if those results are actually more accurate!) Another reason is the need for reproducibility when different parts of a calculation are performed by different machines in a network of heterogeneous machines possibly widely separated geographically. (In fact, in such an environment, the same part of a calculation could be performed by different machines at different points in time.) A third reason is that sometimes full bit-for-bit reproducibility is simply very highly desired so that users can count on a particular behavior regardless of the machine used to run a given application.

In general, the need for bit-for-bit identical results dictates that the expression evaluation scheme be the same, regardless of what machine is being used. Any of the evaluation schemes discussed above can be used as a basis for obtaining bit-for-bit identical results, provided the floating-point formats involved are effectively identical, that is, the floating-point arithmetic engines of interest can simulate one that can operate on and produce results in the floating-point formats the expression evaluation scheme requires. Obviously, this is not a sufficient condition for obtaining bit-for-bit identical results. Using the Java language [38] as the background for discussion, the remainder of this section looks at what other conditions are required.

9.3.1 Optional and implementation defined features of the IEEE Standard

The creators of the Java language often invoke the slogan "write once, run anywhere": implementations are required to produce bit-for-bit identical results in all but a few cases[2]. In order to achieve this goal for floating-point arithmetic operations, Java requires implementations to conform to the IEEE Standard. Whether this is enough is certainly not clear. In fact, because floating-point arithmetic is notorious for varying in subtle ways from one floating-point unit to another (even among those made by the same manufacturer), some are skeptical as to whether achieving bit-for-bit identical results across a variety of floating-point systems is practical, even if one were to restrict one's attention to arithmetic engines conforming to the IEEE Standard,

[2]Section 10.2.2 mentions a case in which bit-for-bit identical results are not guaranteed.

especially since making it possible to obtain bit-for-bit identical results does not appear to be (wisely, in some people's opinions) among the IEEE Standard's goals.

Before dismissing this goal altogether as a hopelessly impractical one, it would be wise to consider which features of the IEEE Standard are either optional or explicitly implementation defined, since these are the ones that could lead to differing results among IEEE-Standard-conforming arithmetic engines. The following features come to mind:

1. Data formats. Although the IEEE Standard defines the characteristics of the basic formats (that is, the single and double formats), thus providing an unambiguous interpretation for every possible encoding, it only establishes a few constraints on the extended formats. Furthermore, it does not even require that implementations support any format other than the single format. Certainly, the availability and use of different (optional) data formats can lead to possibly dramatically different results.

2. NaNs. The IEEE Standard does not specify the encodings of values in the extended formats, much less how NaNs are to be encoded in these formats. In fact, the IEEE Standard does not even specify how signaling and quiet NaNs are to be distinguished in the basic formats, or what bit pattern is to be produced when, say, an invalid operation is attempted with operands that are not NaNs, and all traps are disabled. Consequently, a particular encoding may represent a signaling NaN in one implementation, and a quiet NaN in another implementation, and different implementations can produce different bit patterns when an invalid operation is attempted. While this does not ultimately change the actual result of a computation, particularly when all traps are disabled, different implementations can fail to produce bit-for-bit identical results when NaNs are involved.

3. Whether rounding precision modes are implemented. The IEEE Standard allows implementations to either deliver results to destinations of varying widths, depending on the widths of the operands, or (except in the case of conversions) to always deliver results to a fixed format, provided the width of this format is at least as wide as the double format. In the latter case, it must be possible to cause results to be rounded to (supported) precision(s) shorter than that of the fixed format. The exponent range, on the other hand, is not required to be narrower than that of the fixed format—see chapter 5. Obviously, this can cause different implementations of the IEEE Standard to produce different results.

4. What operations are available. The IEEE Standard requires implementations to provide certain operations involving floating-point operands or results. However, except for conversion to supported floating-point formats, the IEEE Standard does not go so far as to specify in what format floating-point results are to be delivered. In other words, subject to a few constraints, implementors can choose in what format operand(s) are required to be for any particular operation, and in what format the result will be delivered. It is therefore conceivable that one could come up with two fully-conforming implementations of the IEEE Standard such that none of the operations provided by one of them is exactly identical to the operations provided by the other implementation of the IEEE Standard. (One example would be an implementation that supports only the single format, and

another in which all floating-point results, except for conversion to the single format, are delivered in the single extended format.)

5. How underflow is detected. If the underflow trap is disabled, the underflow exception is signaled when both tininess and loss of precision is detected. An implementor can choose between two different ways of detecting tininess and two different ways of detecting loss of precision. While results will be the same regardless of how tininess and loss of precision is detected, these different detection methods can cause underflow to be signaled under slightly different circumstances, depending on the implementation. This can lead to different results if they can change in some way depending on whether the underflow exception has been signaled.

6. Whether it is possible to enable traps. Implementations are not required to allow traps to be enabled, but if traps can be enabled, then results can change if a trap handler is invoked.

7. What information is made available to trap handlers, should they be enabled. If it is possible to enable traps, certain information must be made available to trap handlers, depending on what kind of exceptional situation arises, but implementations of the IEEE Standard are encouraged to provide information beyond that required.

Some implementations of the IEEE Standard provide additional features not covered by the IEEE Standard. An example of such a feature would be the fused multiply-add operation, which takes the product of two operands and adds it to a third, with rounding occurring just once (see chapter 7). Obviously, if features such as these are used, bit-for-bit identical results might not be achieved.

9.3.2 Are bit-for-bit identical results achievable in Java?

Now that all the potential reasons why different implementations of the IEEE Standard can produce different results have been identified, one can consider whether Java's claim that bit-for-bit identical results are achievable is credible:

1. Data formats. Java requires implementations to support two floating-point formats: the single and double formats. Implementations are not allowed to support any other predefined floating-point formats.

2. NaNs. Java does not distinguish between signaling and quiet NaNs, as the IEEE Standard requires, nor does the definition of the Java language specify what bit pattern is produced in situations in which the result of an operation is a NaN[3]. Consequently, it is conceivable that if results involve NaNs, or if results depend on the particular bit pattern of a NaN that emerges as an intermediate result, they might not be bit-for-bit identical across

[3]However, Java does specify what bit pattern is returned when the floatToIntBits or doubleToLongBits method in java.lang.Float or java.lang.Double, respectively, is called with a NaN argument.

different implementations of Java. (However, it is admittedly not obvious how one could go about determining what bit pattern is being used for a particular NaN.)

3. Whether (rounding) precision modes are implemented. Java does not make provision for precision modes, nor is there a need for such a provision, since implementations can deliver floating-point results to destinations in either of the two supported formats. (Note that rounding modes are not supported in Java—the results of all floating-point operations must be rounded to the nearest representable value.)

4. What operations are available. All implementations must provide the operations the Java Virtual Machine (JVM) [61] is capable of executing; all floating-point arithmetic in Java programs must be mapped to the operations the JVM provides.

5. How underflow is detected. Java does not provide access to the (sticky) status flags that the IEEE Standard requires implementations to make available. Therefore, how tininess and loss of precision is detected is irrelevant.

6. Whether it is possible to enable traps. Java does not provide a means for floating-point traps to be enabled.

7. What information is available to trap handlers. As far as Java is concerned, it does not matter what information is available to trap handlers, since traps cannot be enabled, and, consequently, trap handlers cannot be invoked.

It would thus appear that insofar as floating-point arithmetic is concerned, Java's claim that bit-for-bit identical results are achievable is credible, provided all NaNs are treated identically regardless of what their bit patterns are, and that different language processors do not map floating-point arithmetic in source programs to operations provided by the JVM in a way that alters the results obtained (which indeed the Java language specification does not allow).

However, one must also consider the practicality of the constraints the Java language imposes upon its implementations. For example, some of the operations provided by the JVM do not map efficiently to operations provided by the processor on which the JVM is actually running. In particular, on processors conforming to the Intel x86 processor, basic floating-point arithmetic in Java can be about an order of magnitude slower than the most straightforward way of performing floating-point arithmetic on these processors. This is because on these processors, in order to faithfully simulate an arithmetic engine that fully supports the double format as the widest supported floating-point format, a time consuming check is required in most cases after each individual floating-point operation if one is to avert the possibility of double rounding (see chapter 6). In fact, Sun extended the Java language in order to allow implementations to optionally provide better performance for floating-point arithmetic [4]. Of course, this could cause implementations to no longer produce bit-for-bit identical results, unless the new keyword *strictfp* were used at appropriate places.

In summary, while it is possible for Java programs to produce bit-for-bit identical results across a variety of processors, this has been achieved only by designing a particular virtual

machine, and then requiring all implementations of Java to faithfully simulate the operations of which this virtual machine is capable. As expected, the price to be paid for this level of portability is relatively poor performance in the case of the actual processor being used not matching this virtual machine closely. So for most practical purposes, the development of Java does not disprove the commonly-held belief that the IEEE Standard does not allow for bit-for-bit identical results.

Chapter 10

Supporting the IEEE Standard in Ada and Java

Figuring out how to support the IEEE Standard in a high-level language is not a trivial task. This is due partly to the number and nature of the features specified in the IEEE Standard, and partly to the fact that the floating-point aspects of most languages did not receive careful consideration when they were designed, or, at best, were not designed with the IEEE Standard in mind. As a result, a certain amount of creativity is required in order to shoehorn at least the most important features of the IEEE Standard into a given language.

This chapter sketches how the IEEE Standard could be supported in two significantly different high-level languages: Ada and Java. Although many ideas on how to achieve this support is similar, this case study illustrates how, as in the case of adding an extra bathroom to a house, one design definitely does not fit all. Instead, careful consideration is needed to find the best way to support each feature in a given language.

The main considerations in deciding how to support these features were:

1. how each feature of the IEEE Standard is likely to be used—it ought to be possible to make use of each feature in the way programmers are likely to want to use them;

2. completeness—as many features of the IEEE Standard as possible should be accessible;

3. flexibility and generalness—these features should not be accessible under only very limited circumstances;

4. level of control—facilities for accessing these features should satisfy the needs of both naive and expert numerical programmers;

5. simplicity—the rules governing the behavior of a system in which these features are accessible should not be overly complex or difficult to remember; and

6. performance should not be unduly impacted simply because these features are used or even available.

Note that this chapter assumes material discussed in previous chapters is familiar to the reader.

10.1 Ada

Unlike some other languages, the definition of the Ada language [11] does not preclude satisfactory support for the IEEE Standard. Because of Ada's richer set of facilities, such support can be more convenient than in other languages—one does not need to resort to changing the definition of the language in order to avoid making use of the features of the IEEE Standard excessively cumbersome.

This section sketches how the IEEE Standard could be supported in the Ada language. The appendix fills in some of the details not presented here. The phrase "this specification" refers to the specification in the appendix.

10.1.1 Data formats

The package Standard_FP_Arithmetic, whose declaration is implementation defined, names the data formats that the implementation supports. The IEEE Standard requires support for the single format, and this specification additionally requires support for the single extended format, which, in practice, can be equivalent to the double, or even double extended, format.

In addition to providing names for the formats described in the IEEE Standard, this specification provides aliases for the format whose use tends to result in improved floating-point performance, for the widest format supported, and for the widest format supported in hardware (in case a wider format is supported in software).

10.1.2 Rounding and rounding precision modes

Subprograms are provided to get and set the rounding mode. In addition, declaring objects of a certain limited controlled type declared in Standard_FP_Arithmetic allows one to specify the rounding mode that should be in effect in a given block of code. Once control reaches outside this block, the rounding mode in effect automatically reverts back to how it was set before entering this block.

A pragma allows the programmer to identify subprograms that modify the rounding mode. The language processor may assume that subprograms not so identified do not modify the rounding mode.

There is no need to provide facilities that explicitly control the rounding precision mode, due to the facilities for expression evaluation, and the availability of subprograms that perform arithmetic operations in any supported precision.

10.1.3 Operations

In addition to requiring all the language's operators to conform to the IEEE Standard where appropriate, this specification provides a package called Generic_FP_Operations, which declares

functions that are guaranteed to conform to the IEEE Standard. These can be used, for example, to ensure that double rounding does not occur, or to control the precision with which a particular operation is carried out. In addition, this allows the rounding mode to be specified on an operation-by-operation basis, as well as to determine whether an exceptional situation arose while performing a particular operation.

A nested generic package that can be instantiated using any integer subtype as its parameter provides operations that convert between floating-point and integer values.

The functions that convert between binary and decimal representations of a value behave similarly to the attributes image, wide_image, value, and wide_value, except that the functions fully conform to the IEEE Standard.

Comparison is supported by the predefined operators as well as predicates, some of which do not correspond to any predefined operators. Although only 8 predicates are provided, the remaining 18 that the IEEE Standard describes can be generated using logical negation, or by enabling (or disabling) signaling of the invalid operation exception if one or more of the operands is a NaN. The alternative of having functions that return a bit string indicating which relationship holds between two values was not chosen because not all processors are capable of generating such bit strings quickly.

Besides the required and optional operations described in the IEEE Standard, this specification makes available the fused multiply-add instruction that some processors provide. (This instruction multiplies an operand with the sum of two other operands, and rounds only once after the product has been computed.) A Boolean constant indicates whether the target supports this operation directly in hardware, so an alternative algorithm can be used instead if such support is not available.

Another additional function is provided to determine what the sign bit of a value is, regardless of what that value represents.

10.1.4 Exceptional situations

Special values

Special values, such as infinities and NaNs, are available as attributes. For example, writing S'Infinity, where S is a floating-point type, yields an infinite value of type S. This allows these values to be used as compile time constants; no arithmetic with its attendant side effects is required in order to generate these values[1], nor do functions need to be called. For convenience, the package Generic_FP_Operations also declares these values.

Negative zero, that is, zero with a negative sign bit, is specified by using the unary minus operator in front of a floating-point zero (as in "`-0.0`"). This works because in implementations of Ada that are compatible with the IEEE Standard, the attribute Signed_Zeros is true, that is, the implementation follows the rules specified in the IEEE Standard regarding the sign of zeros.

[1]Note that Constraint_Error would be raised if one were to write `0.0/0.0`, so this is not a suitable synonym for NaN. For the same reason, `1.0/0.0` is not a suitable synonym for infinity.

Status flags

Besides subprograms that return or modify the state of the status flags, a limited controlled type is provided so that modification of the status flags can be limited to a desired lexical scope. This feature can be used, for example, to determine what floating-point exceptions were signaled in a given block of code: the status flags can be cleared at the beginning of the block and checked at the end of the block, at which time the status flags revert back to how they were set before entering the block of code.

Alternatively, it is possible to ensure that a specific status flag remains set, even after leaving a block of code. This allows one to make subprograms behave like atomic operations from the standpoint of how the status flags are set: the state of the status flags before a computation began can be restored at the end, and in addition, one can specify that some other flag or flags also remain set, but not the ones corresponding to any spurious exceptions that may have been signaled during the course of the computation. For example, one could envision implementing the function $\exp(x)$ where $x < 0$ as $\exp(x) = 1/(e^{-x})$. If the magnitude of x were large, overflow could be signaled while computing $e^{|x|}$. In such a case, one would ideally want $\exp(x)$ to return zero[2], while signaling the underflow and inexact exceptions. It would not be correct to also signal the overflow exception.

Retrospective diagnostics (that is, a summary of what exceptions were signaled during the course of a computation, reported after the computation has finished) is also provided via a limited controlled type.

Floating-point trap handling

Normally, predefined operators are not allowed to raise arbitrary exceptions, except for the specific cases mentioned in the language definition. This specification offers three alternatives specifically for getting around this restriction:

1. As mentioned above, functions declared in Generic_FP_Operations can be called to perform basic arithmetic, comparison, or conversion operations in order to determine whether any exceptional situations of interest arises while performing these operations. If so, an exception can be raised explicitly.

2. An arbitrary exception can be raised as part of finalization of a controlled object that triggers retrospective diagnostics if any status flag among those specified would remain set after finalization of that object is complete.

3. The notion of trap handlers is introduced as an extension to the language. Unlike interrupt handlers, which are invoked asynchronously typically in response to some external event, trap handlers are invoked as a result of executing user-written code, and they behave very much like implicit subprogram calls—after the trap handler finishes execution, control normally returns to the point in the program that caused the trap handler to be invoked.

[2]If the magnitude of x were within a certain range, the $\exp(x)$ should return a denormalized number instead of zero, but in order to do this, a different algorithm from that outlined here would have to be used.

In addition, trap handlers are allowed to raise arbitrary exceptions in the context of the point in the program in which they are invoked. (An example of a situation in which being able to write a trap handler could be useful is managing page faults in a nonstandard way.)

The last of these alternatives merits further discussion.

Since the ability to enable trap handling is an optional feature of the IEEE Standard, this specification does not require trap handlers to actually be available, though it must be possible to compile programs that make use of trap handlers. A Boolean constant may be tested at run time to determine if trap handlers are actually available.

Although this specification provides subprograms that can prove useful when writing trap handlers, several prewritten trap handlers Kahan advocates in [54] are also provided to reduce the need to write one's own trap handlers. (This is an important feature because trap handlers are likely to be processor and platform specific.) These prewritten trap handlers allow one to count how many times a given type of exceptional situation arose, to specify that a certain value be substituted for the result of operations in which a given type of exceptional situation arises, to raise an arbitrary exception, or to stop execution of the program.

A certain subprogram can be called to associate a trap handler with or disable trap handling for a given type of exceptional situation.

This specification provides a controlled type that allows one to limit the scope over which a trap handler is in effect to a given block of code.

10.1.5 Expression evaluation

All floating-point expressions are evaluated to at least a certain minimum precision that is implementation dependent. However, if the precision of any operand is wider than this, the corresponding operation is carried out in this wider precision. This accommodates floating-point arithmetic engines that perform better when operations are carried out at less than maximum precision (for example, when lower precisions are supported in hardware), and avoids requiring a single, specific scheme for expression evaluation that might penalize performance on certain types of processors.

Although this specification does not guarantee bit-for-bit identical results regardless of the processor used, this can be nearly achieved by using a pragma to require the minimum precision used for expression evaluation to be that corresponding to the single format. (This same pragma allows one to alternatively require the minimum precision to be any other precision the implementation supports.) Of course, specifying a minimum precision different from the default minimum precision could adversely impact performance.

10.1.6 Pragmas related to floating-point arithmetic

Other pragmas, besides the one mentioned above, can be used to control how floating-point arithmetic is performed. Using these pragmas allow one, for example, to require that no results be rounded more than once, or to discard precision beyond what is normally available, given the format associated with the floating-point data types used.

Still other pragmas that are available may increase performance possibly at the expense of producing dubious results given the right conditions. For example, one can specify whether the fused mulitiply-add instruction that some processors provide can be used, or whether floating-point side effects (such as whether floating-point exceptions might be signaled) can be disregarded when making decisions as to specific code improvements or optimizations. It is even possible to allow arithmetic to be performed in a way that might not always conform to the IEEE Standard, or to provide hints to the language processor as to what rounding mode is in effect.

10.2 Java

Unfortunately, it is not possible to fully support the IEEE Standard in Java [38] in a way that is convenient to use without changing the definition of the language, since support for some of the IEEE Standard's features, such as dynamic rounding modes, is specifically forbidden. However, this has not stopped people from proposing how to improve the language's facilities for numerical computing.

The three major "dialects" of Java that are specifically intended to improve Java's suitability for numerical computing are, in order of the magnitude of their changes to Java's language definition, Ivory [37], RealJava [21], and Borneo [22]. Ivory's main goal is to allow Java programs to run faster on implementations for processors conforming to the Intel x86 architecture; RealJava is an adaptation of C9X's ideas on how to support the IEEE Standard [75], but in the context of Java; and the author of Borneo overcomes the fear of changing the Java language too drastically in his journey beyond simply making the IEEE Standard's features available in Java. These three "dialects" will be mentioned from time to time in the remainder of this section, which briefly sketches how Java could be changed to better support the IEEE Standard, without making too many intrusive changes to the language.

10.2.1 Data formats

Java supports the single and double formats. While this certainly fulfills the IEEE Standard's requirements, some processors support other formats, such as the double extended format. On such processors, it is important to be able to access the additional format(s); otherwise, any additional accuracy the processor is capable of providing would be difficult to obtain and control, and performance could be negatively impacted, due to, for example, the need to use algorithms that are less than optimal.

On the other hand, adding an additional floating-point data type to the language triggers cascading modifications to other parts of the language, as Darcy points out [22]. These include adding a new class to the java.lang hierarchy analogous to java.lang.Float and java.lang.Double, adding new syntax for numeric literals of that type, changing the rules for type promotion and expression evaluation, and adding support for this type in other classes, such as java.lang.Math, which contains methods representing the trigonometric functions, among other things. Changes to the language would also need to be reflected in the specification for the Java virtual machine:

at least one, and more likely more than one, opcode would need to be added for each new floating-point data type.

Another reason for hesitation is Java's insistence on the motto "write once, run anywhere"—all Java language processors and virtual machines would need to support any floating-point data types added to the language. It is not an option to allow some language processors or virtual machines to reject or refuse to run a Java program simply because a primitive data type is not supported, especially if some other implementation were to support the data type in question. So if the underlying hardware were not to have support for a given additional floating-point type or types, support would have to be provided in software, and performance could suffer so drastically, depending on the rules for expression evaluation, as to make the language of no more than theoretical interest for those interested in numerical computing.

One potential way that several have proposed to reduce the potential performance impact of adding a new floating-point data type is to make its corresponding format implementation dependent [21, 22, 29], in much the same manner as is done with the C language's long double type [16]. Of course, if such a type were added to Java, programs making use of such a type would not qualify for the label "write once, run anywhere." However, given the drawbacks of other potential ways of providing support for additional IEEE-Standard-compliant data types, this is possibly the best solution in the context of the Java language. Providing additional data types beyond this, as RealJava and Borneo do [21, 22], is more difficult to justify.

10.2.2 Rounding and (rounding) precision modes

The Java language does not make any provision for using different rounding modes. In fact, arithmetic operators, when applied to floating-point operands, are explicitly required to round to nearest. Such restrictions should be relaxed—given the current definition of the language, calling a function or procedure to change the rounding mode should theoretically have no effect, since the behavior of arithmetic operators does not depend on what functions or procedures have been called.

It is not obvious what more can be done to make control over rounding modes more convenient other than to provide methods to retrieve or change the rounding mode. For example, using the facilities the language provides, the rounding mode cannot be saved and restored automatically by simply declaring something, or calling some function or procedure. Although classes can be initialized and finalized, the exact point in which an instance of a class is finalized is not definite—this depends on when the garbage collector reclaims the storage of the object (and thus is an area where the objective of obtaining bit-for-bit identical results was compromised). A somewhat inconvenient programming technique that can be used, however, is to save the rounding mode in a *try* block, and restore the saved mode in the corresponding *finally* block. The alternative is to extend the language with a special declaration that causes the language processor to save the rounding mode at the beginning of a block, and restore the saved mode at the end of the block, as is done in Borneo [22]. The problem with this approach is that it is too reminiscent of the notion of "programming by pragmas"—such declarations by themselves directly cause significant amounts of code to be executed.

Not all implementations of the IEEE Standard need to implement (rounding) precision

modes. Therefore, it is not as appropriate to provide explicit support for this feature, especially since it is possible to offer control over this feature through other means, such as providing functions that perform basic arithmetic with well-defined semantics—see below. Also note that even Java 2's relaxed floating-point semantics [4] offer very few opportunities to change the rounding precision mode in a floating-point arithmetic engine that has this feature.

10.2.3 Operations

Java 2's relaxed floating-point semantics [4] and implementation of the expression evaluation schemes suggested below make it important to provide subprograms that perform basic arithmetic with well-defined semantics, similar to what would be obtained in *strictfp* mode, with rounding guaranteed to occur just once. An additional parameter could be provided to allow the programmer to specify the rounding mode for individual operations. These subprograms could appear, for example, as methods in java.lang.Float and java.lang.Double.

Support for the operations the IEEE Standard requires or recommends, but missing from the language definition, could be provided in the same way. These operations include converting floating-point values to integers (rounding according to the rounding mode in effect, or at least to nearest), determining if a value is finite, scaling a number by an integral power of two, and all 26 predicates related to comparison of two values, among others.

Although the IEEE Standard does not require or recommend this, support should be provided in the same manner for the fused multiply-add instruction that some processors have. (This instruction adds an operand to the product of two other operands, and rounds only once after the sum has been computed.) A Boolean constant should also be provided to indicate whether a given implementation supports this operation directly in hardware.

An alternative to providing additional methods in the java.lang hierarchy would be to provide these methods in some other class. This approach would have the advantage of not needing to change the language definition in order to make these operations available.

10.2.4 Exceptional situations

Special values

Though perhaps not so critical in practice, support should be provided for signaling NaNs. This would, for example, entail adding an additional constant in the classes java.lang.Float and java.lang.Double, establishing a string to be produced by methods such as toString, and establishing a bit pattern to represent signaling NaNs for use in the methods floatToIntBits, intBitsToFloat, doubleToLongBits, and longBitsToLong.

Special syntax also needs to be established so that strings representing (signaling or quiet) NaNs can be passed to methods such as Float, Double, and valueOf.

Of course, it would be even better if the syntax for numeric literals were extended so that values such as infinities and NaNs could be written explicitly as numeric literals, rather than needing to call special methods, or writing a numeric literal representing a number so large, that it would overflow the double format, as is currently required to represent infinities in methods

such as valueOf. Another advantage of extending the syntax of numeric literals is that the output of methods such as toString could then be directly used as input to methods such as valueOf.

Status flags

Access to status flags could be provided in a way analogous to that of rounding modes, with methods to get, set, or clear the status flags. Methods could also be provided to save and restore the status flags, but again, this could not be done automatically at the beginning and end of a block of code without extending the language. The technique of saving the status flags in a *try* block, and restoring (or updating) them in the corresponding *finally* block could be used instead; this would allow one to crudely simulate the basic arithmetic operations as far as returning a value and simultaneously signaling one or more floating-point exceptions.

It might be appropriate to provide retrospective diagnostics by instanstiating a class that reports the diagnostics as part of its finalization, assuming retrospective diagnostics is only desired at the end of programs.

As Darcy mentions in his thesis [22], if access to the status flags were allowed, certain code improvements that are currently allowed would need to be forbidden. This includes, for example, constant folding and code hoisting, since these optimizations can change which status flags are set.

Floating-point trap handling

Java does not allow exceptions to be thrown or floating-point trap handlers to be invoked as a byproduct of performing basic arithmetic operations. Since the IEEE Standard does not require implementations to make it possible to enable floating-point traps, one could be tempted to simply not provide support for this feature. Certainly, it would be easier to accept the excuse that support for this feature does not fit well with the rest of the language. But if this excuse were to not prove sufficient, one might let the programmer choose between a few different well-defined trap handlers, such as the ones Kahan suggests [54]. A full-blown trap handling mechanism is probably overkill in the context of the Java language.

Expression evaluation

As others have remarked [56], Java's rules for floating-point expression evaluation are unfortunately too closely tied to the rules for determining the semantic type of floating-point expressions. These rules do not sufficiently protect naive users who may be unaware of their need to use greater precision in their computations (particularly for intermediate expressions), and does not allow one to exploit the extra accuracy that some floating-point arithmetic engines provide. Though still not optimal, rules for floating-point expression evaluation similar to those of the C language [16] would have been better [56]. Some had hoped Java 2's relaxed floating-point semantics would bring an improvement in this area, but alas, this did not happen—the only issue the new rules addressed was poor performance on processors conforming to Intel's x86 architecture [76].

In any case, at this point it would be difficult to significantly change Java's default rules for floating-point expression evaluation. What might make sense is to add an alternative set of rules requiring intermediate expressions to be evaluated using the full precision the underlying arithmetic engine is capable of providing without significantly reducing performance. In practice, it might be necessary to limit the precision used to that of the widest floating-point data type available, as this would make translating Java source code to Java byte codes more straightforward. So if a data type were added to correspond with the double extended format, as previously suggested, these alternative rules would require intermediate expressions to be evaluated using the precision corresponding to this new data type; otherwise, intermediate expressions would be evaluated using double precision. Some syntax would need to be invented to enable these alternative rules.

This same new syntax could also be used to enable the use of the fused multiply-add instruction that some processors provide. If one wants to ensure this instruction is used in a particular place, one could call a method made available for this purpose, as previously suggested. Conversely, if one does not wish this instruction to be used in a particular situation, one could similarly call a different method to perform the desired operation.

Chapter 11

Conclusion

As can be seen, contrary to what some language designers seem to believe (at least judging from the languages they design) fully supporting the IEEE Standard in a high-level language is much more than simply making available one or two floating-point data formats, and providing a set of operations on data stored in these formats. Fully supporting the IEEE Standard involves making available a number of features described in the IEEE Standard, and satisfying several specific, concrete criteria, as summarized in chapter 2.

The foregoing chapters show that, although very few languages, if any, fully support the IEEE Standard, not only is it possible with an appropriate amount of ingenuity and dedication to supplement a language in such a way that it fully supports the IEEE Standard (as is the case with Ada [11]), but it is also possible to satisfy the aforementioned criteria in the process—see chapter 10 and the appendix. This work also shows that in some cases, certain fundamental changes need to be made to language definitions (such as Java's [38]) in order for them to fully support the IEEE Standard, even if superficially they appear to be IEEE-Standard friendly. (Again, see chapter 10.)

11.1 How well current languages support the IEEE Standard

At this point, it would be instructive to compare how well different languages, such as C, Fortran, Ada, and Java, support the IEEE Standard. For each of these languages, Tables 11.1 and 11.2 summarize the following information:

A. which of the data formats described by the IEEE Standard can be supported in the same way as primitive data types;

B. how access to rounding and precision modes can be (or is) provided at run time;

C. whether all the operations and recommended functions are supported;

D. how special values, such as NaNs, infinities, and signed zeros are or can be supported;

E. how access to status flags can be provided;

Language Feature	C	Fortran	Java	Ada	Proposed IEEE binding for Ada
A	any 3	implementation dependent	single and double	implementation dependent	all
B	implementation dependent function calls or pragmas	implementation dependent function calls	not available	implementation dependent function calls, pragmas, or controlled types	function calls, pragmas, and controlled types
C	implementation dependent	implementation dependent	implementation dependent	implementation dependent	yes
D	implementation dependent literal strings or function calls	implementation dependent literal strings or function calls	method calls (signaling NaNs not available)	implementation dependent literal strings (for signed zeros), function calls, or attributes	literal strings (for signed zeros), attributes, and named constants
E	function calls	function calls	function calls	function calls	function calls

Table 11.1: How well current languages support the IEEE Standard

F. whether support for enabling and disabling traps can be provided;

G. whether values of floating-point expressions can be deterministic;

H. whether additional (or different) facilities to those provided by the language definition are required in order to fully support the IEEE Standard; and

I. whether there is an inherent performance penalty in the manner in which the IEEE Standard is (or could be) supported.

As a point of comparison, Tables 11.1 and 11.2 also provide information on the IEEE-Standard binding for Ada described in the appendix.

Language Feature	C	Fortran	Java	Ada	Proposed IEEE binding for Ada
F	requires writing processor-dependent functions	requires writing processor-dependent functions	not allowed	requires writing processor-dependent functions	yes if notion of "trap handlers" implemented
G	implementation dependent	implementation dependent	only in strict mode	implementation dependent	by default; may be disabled with a pragma
H	yes, to support special values	yes, to support special values	yes, to support signaling NaNs, rounding modes, and status flags	no	no
I	implementation dependent	implementation dependent	significant performance penalty for processors such as those conforming to Intel's x86 architecture	implementation dependent	no

Table 11.2: How well current languages support the IEEE Standard, *continued*

11.2 Related unfinished work and open issues

The recommendations contained herein have yet to be implemented in a language translator. Obviously, this is an opportunity for further work, and would serve to validate the ideas espoused here, particularly as it regards to the Ada language.

In addition, writing significant bodies of code would further validate the soundness of these ideas, and potentially expose weaknesses and opportunities for improvement. It would be especially valuable to try exploiting nontraditional facilities, such as multiple floating-point data formats, various rounding modes, and handling special computational situations, including trap handlers, the latter being an area in which more experience is sorely needed. (A unique feature that the IEEE binding for Ada in the appendix provides, and that would be interesting to try exploiting, is that of being able to write functions that simultaneously return results and signal one or more exceptions.)

Hopefully the specification for Ada given in the appendix can be incorporated in some form into the next Ada standard. Although this author's recommendations for changes to the Java language [29] in the end were not incorporated into the Java 2 platform [4], perhaps they did and will serve to influence changes to the Java language definition for the better—there should be opportunities in the future for making even more improvements to the Java language to support numerical programming better [60].

In the end, one of the most important reasons for a work such as this one is to increase language designers' and other interested parties' generally incomplete understanding of what it takes to fully support the IEEE Standard in a high-level programming language. (The fact that languages, such as Java and Haskell 98 [13], among others, continue to be designed without full support for the IEEE Standard is evidence that works such as this one are badly needed.) This increased understanding should foster more discussion on how current languages might better support the IEEE Standard, and as more and more languages support the IEEE Standard better, experience in taking greater advantage of the features the IEEE Standard has to offer will increase. This, in turn, should lead to higher quality, and even faster, software libraries and bodies of code, even when written by inexperienced numerical programmers. Now that processors' floating-point units ubiquitously support the IEEE Standard, the "holy grail," as it were, is to make the features of the IEEE Standard ubiquitously accessible to high-level language programmers. This work is a step toward achieving this important goal.

Appendix A

Supporting the IEEE Standard in Ada

The goal of this specification is to provide full support in the Ada language [11] for the IEEE Standard [47] in as natural a way as possible, and with as few extensions to the Ada language as possible. Indeed, existing language processors need not change very much in order to make it possible to implement this specification: support for a few new pragmas and attributes would need to be added, and, in general, arithmetic would need to be performed in a manner consistent with the IEEE Standard. A full implementation of this specification does require extending the Ada language with the notion of trap handlers (see section A.3.2), but given that this is an optional feature of this specification, very little is actually required beyond defining a couple types in a package called Ada.Traps.

Paragraphs that appear entirely in italics are explanatory in nature, and are not actually part of this specification.

A.1 The Package `Standard_FP_Arithmetic`

The package `Standard_FP_Arithmetic` has the following declaration:

```
with Ada.Finalization;
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Traps; use Ada.Traps;
-- Ada.Traps defines the types Trap_Handler and Trap_Occurrence
package Standard_FP_Arithmetic is
   type single_float is
     implementation-defined-floating-point-type;
type single_extended_float is
   implementation-defined-floating-point-type;
-- the types double_float and double_extended_float,
-- if supported, would be declared here
```

```
type fast_float is new
   implementation-defined-floating-point-type;
type widest_float is new
   implementation-defined-floating-point-type;
type widest_fast_float is new
   implementation-defined-floating-point-type;
type default_float is new
   implementation-defined-floating-point-type;
type Rounding_Mode is (To_Nearest, To_Zero, Up, Down,
   Current_Rounding_Mode);
procedure Rounding (Round : Rounding_Mode);
function Rounding (Round : Rounding_Mode :=
   Current_Rounding_Mode) return Rounding_Mode;
pragma Modifies_Rounding_Mode (Rounding);
type FP_Rounding_Mode_Management is new
   Ada.Finalization.Limited_Controlled with private;
type FP_Value_Types is (Negative_Infinity,
   Negative_Normal, Negative_Denormal, Negative_Zero,
   Positive_Zero, Positive_Denormal, Positive_Normal,
   Positive_Infinity, Quiet_NaN, Signaling_NaN);
type FP_Exception is (Invalid_Operation,
   Division_By_Zero, Overflow, Underflow, Inexact);
-- FP_Exception may include additional implementation-defined
-- floating-point-related exceptions
type FP_Exception_Array is array(FP_Exception) of Boolean;
pragma Pack (FP_Exception_Array);
procedure FP_Status_Flags
  (Mask : FP_Exception_Array := (others => True);
   Status_Flags : FP_Exception_Array := (others => False));
function FP_Status_Flags
  (Mask : FP_Exception_Array := (others => False);
   Status_Flags : FP_Exception_Array := (others => False))
  return FP_Exception_Array;
type FP_Status_Flags_Management is new
   Ada.Finalization.Limited_Controlled with private;
procedure Update_FP_Status_Flags
  (Saved_Status_Flags : FP_Status_Flags_Management;
   New_Status_Flags : FP_Exception_Array);
type FP_Retrospective_Diagnostics is new
   FP_Status_Flags_Management with private;
procedure Raise_Exception
  (Status_Flags : FP_Retrospective_Diagnostics;
```

```
   FP_Exceptions : FP_Exception_Array :=
     (Invalid_Operation => True, Division_By_Zero => True,
      Overflow => True, others => False);
   Exception_To_Raise : Exception_Id :=
     Constraint_Error'Identity);
FP_Trap_Handling_Supported : constant Boolean := true_or_false;
type FP_Trap_Management is new
   FP_Status_Flags_Management with private;
procedure Install_FP_Trap_Handler
  (FP_Exceptions : FP_Exception;
   Handler : access Trap_Handler := null;
   Counter : access Integer := null;
   Substitute_Value : access widest_float := null;
   Copy_Sign_Of_Result : Boolean := False;
   Exception_To_Raise : Exception_Id :=
     Constraint_Error'Identity);
procedure Install_FP_Trap_Handler
  (FP_Exceptions : FP_Exception_Array;
   Handler : access Trap_Handler := null;
   Counter : access Integer := null;
   Substitute_Value : access widest_float := null;
   Copy_Sign_Of_Result : Boolean := False;
   Exception_To_Raise : Exception_Id :=
     Constraint_Error'Identity);
function Install_FP_Trap_Handler
  (FP_Exceptions : FP_Exception_Array := (others => False);
   Handler : access Trap_Handler := null;
   Counter : access Integer := null;
   Substitute_Value : access widest_float := null;
   Copy_Sign_Of_Result : Boolean := False;
   Exception_To_Raise : Exception_Id :=
     Constraint_Error'Identity)
  return FP_Exception_Array;
procedure Tally_Exception (Trap : Trap_Occurrence);
pragma Trap_Handler(Tally_Exception);
procedure Presubstitute (Trap : Trap_Occurrence);
pragma Trap_Handler(Presubstitute);
procedure Raise_Exception (Trap : Trap_Occurrence);
pragma Trap_Handler(Raise_Exception);
procedure Abort_Task (Trap : Trap_Occurrence);
pragma Trap_Handler(Abort_Task);
type FP_Operations is (Unknown, Addition,
```

Name of IEEE Data Format	Name of Corresponding Floating-Point Data Type
Single	`single_float`
Single extended	`single_extended_float`
Double	`double_float`
Double extended	`double_extended_float`

Table A.1: Floating-point data formats and their corresponding data types

```
   Subtraction, Multiplication, Division, Remainder,
   Square_Root, FP_Conversion, Integer_Conversion,
   Decimal_Conversion, Comparison, Multiply_Add);
-- FP_Operations may include additional implementation-defined
-- operations
function Get_Operation (Trap : Trap_Occurrence)
   return FP_Operations;
procedure Get_Operands
  (Trap : Trap_Occurrence;
   Left, Middle, Right : out widest_float;
   Left_Valid, Middle_Valid, Right_Valid : out Boolean);
procedure Get_Result
  (Trap : Trap_Occurrence; Result : out widest_float;
   Result_Valid : out Boolean);
procedure Set_Result
  (Trap : Trap_Occurrence;
   Result : widest_float;
   Result_Set : out Boolean);
private
   ... -- not specified by this specification
end Standard_FP_Arithmetic;
```

The meaning of these types and functions are explained in the following subsections.

A.1.1 Data Formats

Table A.1 contains the names of the formats described in the IEEE Standard, along with the names of corresponding floating-point data types, some of which may be declared in the package `Standard_FP_Arithmetic`.

Although the IEEE Standard only requires support for the single format, this specification requires all conforming implementations to support the single extended format as well. (That is, the data types single_float and single_extended_float must both be declared in `Standard_FP_Arithmetic`.) The latter format does not need to be distinct from either the double or double extended format, if supported. However, values which are representable in the single extended

format must also be (exactly) representable in the double format, if supported. In addition, such values must be (exactly) representable in the double extended format, if supported. Note that the single extended format is not required to be supported by the underlying hardware.

If the double or double extended format is supported, the data type double_float or double_extended_float, respectively, is declared; if the data type double_float or double_extended_float is declared, it denotes the double or double extended format, respectively.

In addition to the data types above, the package `Standard_FP_Arithmetic` declares the data types fast_float, widest_float, widest_fast_float, and default_float. The first of these corresponds to the widest IEEE-Standard-conforming floating-point data format among those whose use generally leads to the fastest performing programs. The widest_float data type corresponds to the widest supported IEEE-Standard-conforming floating-point data type. If any of the floating-point data formats described in the IEEE Standard is supported by the underlying hardware, the widest_fast_float data type corresponds to the widest such data format. Otherwise, the widest_fast_float data type is identical to the widest_float data type. The default_float data type corresponds to the default precision mode (see section A.3.1); this data type should not correspond to a format narrower than the single extended format.

If one or more of the floating-point data types declared in the package Standard have the same characteristics as data types in `Standard_FP_Arithmetic`, the latter should be subtypes of the corresponding ones in Standard.

A.1.2 Operations of Floating-Point Types

The following attributes are defined for every floating-point type S declared in `Standard_FP_Arithmetic`, or derived from a floating-point type declared therein:

`S'Infinity` yields an infinite value of type S.

`S'Quiet_NaN` yields a value of type S that does not represent any number, or cause any exceptions to be signaled if used as an operand of a basic arithmetic operation.

`S'Signaling_NaN` yields a value of type S that does not represent any number, and causes the invalid operation exception to be signaled if used as an operand of a basic arithmetic operation.

`S'Fast_Multiply_Add` yields the value True if the fused multiply-add operation can be performed with operands of type S in about the same amount of time as a multiplication or an addition, and False otherwise.

Note that Constraint_Error is raised when evaluating a constant initialization expression such as 0.0/0.0. This makes it difficult to devise some way of declaring constants such as infinity or NaN. An alternative to using attributes would have been to use functions to generate these constants, but then these constants would not be usable in static expressions.

A.1.3 Rounding Modes

The first four of the enumeration literals in the declaration of the enumeration type Rounding_Mode correspond to the rounding modes defined in the IEEE Standard. The enumeration literal Current_Rounding_Mode represents the rounding mode currently in effect. The actual order in which these enumeration literals appear is implementation dependent.

The procedure Rounding sets the rounding mode to that specified by its parameter. (Note that if the specified rounding mode is Current_Rounding_Mode, the rounding mode is not actually set.) The function Rounding behaves similarly, except it additionally returns the rounding mode in effect at the time it was invoked.

Whenever an object of type FP_Rounding_Mode_Management is initialized (via the primitive procedure Initialize), the rounding mode in effect at the time of initialization is saved, and this saved rounding mode is restored at the time of finalization when the object's Finalize procedure is called.

A.1.4 Status Flags

The first five of the enumeration literals in the declaration of the enumeration type FP_Exception correspond to the exception types defined in the IEEE Standard. FP_Exception may include additional implementation-defined enumeration literals representing floating-point exceptions outside the scope of the IEEE Standard. For example, in some implementations, an additional enumeration literal might represent integer overflow in conversions of floating-point values to integer formats. The actual order in which these enumeration literals appear is implementation dependent.

The procedure FP_Status_Flags sets or clears the status flags corresponding to the entries in the argument Mask whose values are true, depending on whether the corresponding entries in the argument Status_Flags are true or false, respectively. The function FP_Status_Flags behaves similarly, except it additionally returns an array in which each entry indicates whether the corresponding status flag was set or not at the time the function was invoked.

The declarations related to the management of status flags are discussed in section A.3.2.

A.1.5 Trap Handlers

The declarations related to trap handling are discussed in section A.3.2.

A.2 Floating-Point Operations

The functions declared in the generic package Generic_FP_Operations mostly represent the operations described in the IEEE Standard.

A.2.1 The Package Generic_FP_Operations

```
with Standard_FP_Arithmetic; use Standard_FP_Arithmetic;
```

```
generic type Real is digits <>;
package Generic_FP_Operations is
  Infinity : constant Real := Real'Infinity;
  Quiet_NaN, QNaN, NaN : constant Real := Real'Quiet_NaN;
  Signaling_NaN, SNaN : constant Real := Real'Signaling_NaN;
  function Add
    (Left, Right : Real'Base;
     Round : Rounding_Mode := Current_Rounding_Mode;
     Produce_Model_Number, Round_Once : Boolean := False;
     Exceptions_Signaled : access FP_Exception_Array := null)
    return Real'Base;
  function Subtract
    (Left, Right : Real'Base;
     Round : Rounding_Mode := Current_Rounding_Mode;
     Produce_Model_Number, Round_Once : Boolean := False;
     Exceptions_Signaled : access FP_Exception_Array := null)
    return Real'Base;
  function Multiply
    (Left, Right : Real'Base;
     Round : Rounding_Mode := Current_Rounding_Mode;
     Produce_Model_Number, Round_Once : Boolean := False;
     Exceptions_Signaled : access FP_Exception_Array := null)
    return Real'Base;
  function Multiply_Add
    (Left, Middle, Right :
     Real'Base; Round : Rounding_Mode :=
     Current_Rounding_Mode; Produce_Model_Number :
     Boolean := True; Round_Once : Boolean := False;
     Exceptions_Signaled : access FP_Exception_Array := null)
     return Real'Base;
  function Divide (Left, Right : Real'Base; Round :
     Rounding_Mode := Current_Rounding_Mode;
     Produce_Model_Number, Round_Once : Boolean :=
     False; Exceptions_Signaled : access
     FP_Exception_Array := null) return Real'Base;
  function Remainder (Left, Right : Real'Base;
     Exceptions_Signaled : access FP_Exception_Array := null)
     return Real'Base;
  function Sqrt (Right : Real'Base; Round :
     Rounding_Mode := Current_Rounding_Mode;
     Produce_Model_Number, Round_Once : Boolean :=
     False; Exceptions_Signaled : access
```

```
   FP_Exception_Array := null) return Real'Base;
generic type Another_Real is digits <>;
package Generic_Float_To_Float_Conversion is
     function Convert_To_Float (Right :
        Another_Real'Base; Round : Rounding_Mode :=
        Current_Rounding_Mode; Exceptions_Signaled :
        access FP_Exception_Array := null) return Real'Base;
function Convert_To_Float (Right : Integer; Round :
   Rounding_Mode := Current_Rounding_Mode;
   Produce_Model_Number, Round_Once : Boolean :=
   False; Exceptions_Signaled : access
   FP_Exception_Array := null) return Real'Base;
function Round_To_Integer (Right : Real'Base; Round :
   Rounding_Mode := Current_Rounding_Mode;
   Produce_Model_Number : Boolean := False;
   Exceptions_Signaled : access FP_Exception_Array := null)
   return Real'Base;
function Round_To_Integer (Right : Real'Base; Round :
   Rounding_Mode := Current_Rounding_Mode;
   Exceptions_Signaled : access FP_Exception_Array := null)
   return Integer;
generic type Int is range <>;
package Generic_Float_Integer_Conversion is
  function Convert_To_Float
    (Right : Int;
     Round : Rounding_Mode := Current_Rounding_Mode;
     Produce_Model_Number, Round_Once : Boolean := False;
     Exceptions_Signaled : access FP_Exception_Array := null)
    return Real'Base;
  function Round_To_Integer
    (Right : Real'Base;
     Round : Rounding_Mode := Current_Rounding_Mode;
     Exceptions_Signaled : access FP_Exception_Array := null)
    return Int;
end Generic_Float_Integer_Conversion;
function Wide_Image (Right : Real'Base; Round :
   Rounding_Mode := Current_Rounding_Mode;
   Exceptions_Signaled : access FP_Exception_Array := null)
   return Wide_String;
function Image (Right : Real'Base; Round :
   Rounding_Mode := Current_Rounding_Mode;
   Exceptions_Signaled : access FP_Exception_Array := null)
```

```
      return String;
   function Wide_Value (Right : Wide_String; Round :
      Rounding_Mode := Current_Rounding_Mode;
      Produce_Model_Number, Round_Once : Boolean :=
      False; Exceptions_Signaled : access
      FP_Exception_Array := null) return Real'Base;
   function Value (Right : String; Round : Rounding_Mode
      := Current_Rounding_Mode; Produce_Model_Number,
      Round_Once : Boolean := False; Exceptions_Signaled
      : access FP_Exception_Array := null) return Real'Base;
   function Equal (Left, Right : Real'Base;
      Signal_Invalid_If_Unordered : Boolean := False;
      Exceptions_Signaled : access FP_Exception_Array := null)
      return Boolean;
   function Not_Equal (Left, Right : Real'Base;
      Signal_Invalid_If_Unordered : Boolean := False;
      Exceptions_Signaled : access FP_Exception_Array := null)
      return Boolean;
   function Greater_Than (Left, Right : Real'Base;
      Signal_Invalid_If_Unordered : Boolean := True;
      Exceptions_Signaled : access FP_Exception_Array := null)
      return Boolean;
   function Greater_Than_Or_Equal (Left, Right :
      Real'Base; Signal_Invalid_If_Unordered : Boolean
      := True; Exceptions_Signaled : access
      FP_Exception_Array := null) return Boolean;
   function Less_Than (Left, Right : Real'Base;
      Signal_Invalid_If_Unordered : Boolean := True;
      Exceptions_Signaled : access FP_Exception_Array := null)
      return Boolean;
   function Less_Than_Or_Equal (Left, Right : Real'Base;
      Signal_Invalid_If_Unordered : Boolean := True;
      Exceptions_Signaled : access FP_Exception_Array := null)
      return Boolean;
   function Less_Or_Greater_Than (Left, Right :
      Real'Base; Signal_Invalid_If_Unordered : Boolean
      := True; Exceptions_Signaled : access
      FP_Exception_Array := null) return Boolean;
   function Unordered
     (Left, Right : Real'Base;
      Signal_Invalid_If_Unordered : Boolean := False;
      Exceptions_Signaled : access FP_Exception_Array := null)
```

```
      return Boolean;
    function Is_Finite (Right : Real'Base) return Boolean;
    function Is_NaN (Right : Real'Base) return Boolean;
    function Classify (Right : Real'Base) return FP_Value_Types;
    function Positive (Right : Real'Base) return Boolean;
end Generic_FP_Operations;
```

The package FP_Operations defines the same subprograms as Generic_FP_Operations, except that the type fast_float (which is declared in `Standard_FP_Arithmetic`) is systematically substituted for Real'Base throughout. In addition, for every distinct floating-point data type declared in `Standard_FP_Arithmetic` and wider than fast_float, the package FP_Operations defines a version of each function declared in Generic_FP_Operations that returns a floating-point value and has at least one floating-point argument, with the return type being this wider type, and with the type fast_float being otherwise systematically substituted for Real'Base throughout.

Nongeneric equivalents of Generic_FP_Operations for each of the floating-point types declared in `Standard_FP_Arithmetic` other than fast_float are defined similarly to FP_Operations, with the names Single_FP_Operations, Double_FP_Operations, etc.

Some of the functions in instantiations of Generic_FP_Operations, as well as some of the functions in nongeneric equivalents of Generic_FP_Operations, have a corresponding predefined operator. An expression of the form X op Y, where the base type of X and Y is a floating-point type declared in `Standard_FP_Arithmetic` and op is a binary operator, is equivalent to a function_call of the form operator(X, Y), where operator is the name of the function corresponding to op. Similarly, an expression of the form op Y, where the base type of Y is a floating-point type declared in `Standard_FP_Arithmetic` and op is a unary operator, is equivalent to a function_call of the form operator(Y), where operator is the name of the function corresponding to op.

If functional notation is used, the actual accuracy of the result of an operation depends on the precision of the result type, and may (but is not required to) also depend on the precision mode in effect (as indicated by the use or absence of the pragma Minimum_FP_Precision), provided the pragma Produce_Model_Number is not enabled, and the value of the parameter Produce_Model_Number is false.

A.2.2 Required Functions and Predicates

Generic_FP_Operations declares functions similar to the following one:

```
function Some_Operation
  (Left, Right : Real'Base;
   Round : Rounding_Mode;
   Produce_Model_Number, Round_Once : Boolean;
   Exceptions_Signaled : access FP_Exception_Array)
  return Real'Base;
```

In declarations similar to this one, Left and Right are the "operands" of the function (some functions may have just one "operand"); Round indicates the desired rounding mode for the operation; Produce_Model_Number indicates whether the result of the function must be restricted to a model number of the (base) type, that is, whether extra precision and range in the result is undesirable; Round_Once indicates whether the value returned by the function must not differ from that which would be obtained if rounding were performed only once in the process of arriving at the result; Exceptions_Signaled, if nonnull, indicates, on input, which exceptions are of interest, and, on output, among those of interest, which exceptions, if any, were signaled as the operation was performed.

For many operations, there is a version with the parameter Exceptions_Signaled and a version without this parameter.

If a trap is enabled for any of the exceptions of interest (as indicated on input by the parameter Exceptions_Signaled) at the point in which such a function is called, that trap is not invoked as a byproduct of the indicated operation having been performed. The status flag corresponding to any exception that is not of interest is updated normally, as specified in the IEEE Standard.

The function Multiply_Add, which does not correspond to any function described in the IEEE Standard, returns the sum of Right and the product of Left and Middle. If Round_Once is True, then Multiply_Add adds the exact product of Left and Middle to Right, and does not perform any rounding until the sum has been computed. Otherwise, the result returned may be any value that could be returned by Add if one were to first call Multiply identically to Multiply_Add (except for the omission of Right), and then call Add with the value returned by Multiply as the parameter Left, and with the values of all the other parameters the same as the corresponding ones to Multiply_Add. Whether one or more exceptions can be signaled due solely to the computation of the product of Left and Middle when the mathematical value of this product is finite is implementation dependent.

The functions Image, Wide_Image, Value, and Wide_Value behave similarly to the corresponding identically named attributes, except that rounding is performed according to the specified rounding mode, regardless of the magnitude of the value being represented. (Note that in some cases, this implies greater accuracy than that required by the IEEE Standard.) These functions must otherwise conform to the specifications in the IEEE Standard related to conversions between decimal and floating-point formats, regardless of what kind of floating-point format is involved (that is, basic or extended). In addition, the number of digits before the uppercase E in the string returned by Image or Wide_Image is

$$\lceil \texttt{Real'Base'Model\_Mantissa} \log_{10} \texttt{Real'Base'Machine\_Radix} \rceil + \text{sign}(\texttt{Real'Base'Machine\_Radix} \bmod 10),$$

where sign(...) is either 0 or 1, depending on whether Real'Base'Machine_Radix is 10 or not. If the value of Right is infinite, then Image and Wide_Image produce the sequence of characters INFINITY with a single leading character that is either a minus sign or a space, as appropriate; if the value of Right is a quiet or signaling NaN, then Image and Wide_Image produce the sequence of characters QNAN or SNAN, respectively, with a single leading character that is

either a minus sign or a space, as appropriate. Similarly, if the sequence of characters of Right (ignoring leading or trailing spaces) is INFINITY, NAN, QNAN, or SNAN (ignoring upper/lower case distinctions), optionally preceded by a minus sign, then Value and Wide_Value return an appropriately signed value that is infinite, a quiet NaN, a quiet NaN, or a signaling NaN, respectively.

In functions that compare two values, such as Equal and Greater_Than, the parameter Signal_Invalid_If_Unordered indicates whether the invalid operation exception should be signaled if the value of at least one of the parameters Left or Right is a NaN. Note that the function Not_Equal actually stands for "less than, greater than, or unordered," and is thus not equivalent to the function Less_Or_Greater_Than.

A.2.3 Recommended Functions and Predicate

Access to most of the recommended functions and predicates in the appendix of the IEEE Standard are already provided by either the package Generic_FP_Operations and its nongeneric equivalents, or by other features of the language. In particular, the attributes Copy_Sign, Scaling, Exponent, and Adjacent have similar functionality to that of the functions copysign, scalb, logb, and nextafter, respectively, which are described in the appendix of the IEEE Standard. The functions Is_Finite, Is_NaN, Unordered, and Classify conform to the specifications in the appendix of the IEEE Standard for the functions finite, isnan, unordered, and class. In implementations of this specification, the unary adding operator - conforms to the specification for negation in the appendix of the IEEE Standard.

Note that if one wants behavior identical to that of Is_NaN or Is_Finite, except that one wants a guarantee that the invalid operation exception will be signaled whenever the argument is a signaling NaN, one can use the function Unordered, or the operator abs in combination with the function Less_Than, respectively.

The function Positive returns the value True or False, depending on whether the sign of its argument is positive or negative, even if its argument is a NaN or zero. Whether the invalid operation exception is signaled when the argument is a signaling NaN is implementation dependent.

A.3 Model of Floating-Point Arithmetic

This specification extends the model of floating-point arithmetic described in section 2.1 of Annex G of the Ada 95 Reference Manual [11], especially in regards to exceptional situations and operations involving exceptional values. In other situations, that is, unexceptional situations and those that do not involve exceptional values, the model of floating-point arithmetic described in the Ada 95 Reference Manual still applies. However, even in these kinds of situations, implementations of this specification must conform to a stricter model, as detailed in the following sections, whenever the "strict mode" is in effect.

A.3.1 Floating-Point Evaluation Format

The accuracy of the result of an operation depends on what evaluation format is used. This, in turn, depends on the precision mode in effect at the point in which the operation appears, as well as the data type(s) associated with the operand(s). In any given implementation, there are as many precision modes as there are distinct IEEE-Standard-conforming floating-point data types. In any given region of code, unless the programmer specifies otherwise, the default precision mode, corresponding to the default_float data type declared in `Standard_FP_Arithmetic`, is in effect.

The precision mode that is in effect in a given region of code determines the minimum accuracy with which floating-point arithmetic operations in that region will be computed. If the precision(s) of the operand(s) of a given arithmetic operation in that region is (are) not wider than that corresponding to the precision mode in effect, then the operation is performed as if the operand(s) were first converted to the format corresponding to the precision mode in effect, and according to the specifications of the IEEE Standard given a destination whose precision is that corresponding to the precision mode in effect. (The allowable range of the result, however, may be wider than that of the format corresponding to the precision mode in effect.) For example, if the precision mode in effect corresponds to the double format, then any operands of a given arithmetic operation narrower than double are implicitly converted to double, and the format of the result is a format not narrower than the double format. The result is computed as specified by the IEEE Standard, with the precision being that of the double format. In many implementations, the result will always be a value that is representable in the double format, but in some implementations, the result, if finite, may be outside the range of finite values that are representable in the double format.

If the precision of at least one operand of a given arithmetic operation is wider than that corresponding to the precision mode in effect, then the operation is performed according to the specifications of the IEEE Standard given a destination whose precision is that of the widest operand of the given operation. (Again, the allowable range of the result may be wider than that of the widest operand of the given operation.)

A.3.2 Exception Handling

The goals of this specification as far as exception handling is concerned are as follows:

1. *Provide full access to the required status flags in such a way that one or more of them can be simultaneously set, sensed, cleared, copied, updated, and restored to a previous value.*

2. *Make it convenient to determine what exception(s) were signaled, if any, during the course of a computation. (If the previous goal is achieved, then this goal is likely to be achieved as well.)*

3. *Allow for (that is, not necessarily require but at least encourage) retrospective diagnostics upon completing execution of programs, along with the means to disable this.*

4. *Provide a way for functions to return a specified value and simultaneously signal that one or more exceptional situations have occurred. This would allow a function to behave as if it were a single arithmetic operation like those described in the IEEE Standard.*

5. *Make it particularly easy to get the default behavior specified in the IEEE Standard (all traps disabled), as well as a "debugging" mode in which occurrences of the invalid operation exception, divide by zero, and overflow are not easy to ignore (for example by raising some exception that is unlikely to be ignored by default). (The predefined operators are not allowed to raise exceptions when the underflow or the inexact exception is signaled, and in any case, both of these exceptional situations are less likely to be of interest when debugging a program.) The latter mode must not be restricted to processors that have the ability to trap precisely when exceptional situations occur—it must be possible to fully implement this mode in software by checking the status flags at appropriate points in a program. (It should ideally be possible to conveniently specify that either of these modes is to be in effect when executing any code in a given program unit, without needing to specify this individually for every subprogram in the given unit, but this specification does not address this.) Provision should be made for optionally making the "debugging" mode be in effect even in called subprograms (in which case an implementation realized purely in software would be insufficient if precise trapping is required, since recompilation may not be an option).*

6. *Provide an optional means to count how many times a given exception has been signaled, an optional means to "presubstitute" the result of specified operations with a given value when exceptional situations arise (see [54]), an optional means to raise an arbitrary exception when a given exception is signaled, as well as a way to detect whether these facilities are available in a given implementation of this specification. (If achieving the following goal is also important, providing these kinds of facilities is likely to be challenging when targeting processors that lack precise traps.)*

7. *Avoid inherently requiring excessive overhead when making use of facilities related to exception handling, since exceptions are likely to be signaled rarely in practice. Another reason this goal is important is that making use of at least some of these facilities will either require access to the processor's status flags, or involve enabling and disabling traps, and on many processors, accessing the register that contains the status flags or controls whether traps are enabled takes several clock cycles and disrupts the floating-point pipeline. If this goal is not met, features related to exception handling will often be avoided in practice, even in cases in which it would have been desirable to use these features. As with rounding modes, one technique that could be used to reduce this overhead is to keep a copy of which traps are enabled in some thread-specific memory location. This copy could be checked, for example, to determine if it is actually necessary to modify the register controlling which traps are enabled. So a goal of this specification is to allow "caching" this information in this manner (but not require this "caching"), and make it clear in what situations the cached value can become stale. Another goal of this specification is to avoid forbidding*

all code improvements which may alter what exceptions are signaled, and when they are signaled.

This is an impressive set of goals!

When a program begins execution, all user-visible floating-point traps are disabled, and all floating-point status flags are cleared.

Trap Handlers

Facilities in this specification related to the handling of floating-point exceptions (as described in the IEEE Standard) require the Ada language to be extended with a feature referred to as trap handlers. For the purposes of this specification, a trap handler is a protected procedure very similar to an (Ada) interrupt handler, except that it is invoked synchronously, that is, as a consequence of attempting to execute certain machine instructions, and its profile has one parameter of type Trap_Occurrence. In addition, a trap handler has access to library level subprograms and objects, and can raise arbitrary exceptions like any normal subprogram. This specification assumes there is a package analogous to Ada.Interrupts called Ada.Traps, which declares the same kinds of entities as the former. This specification also makes use of a pragma called Trap_Handler, which is assumed to be similar to the pragma Interrupt_Handler.

If the value of FP_Trap_Handling_Supported is False, then the behavior of subprograms and pragmas related to floating-point trap handling, or when objects of type FP_Trap_Management are declared, is implementation defined. However, even if this is the case, exceptions must not be raised, and execution of the program must not terminate when floating-point trap handling facilities are used, except possibly when a procedure associated with the pragma Trap_Handler is invoked directly (because then there would no trap occurrence that could be referred to).

If, on the other hand, the value of FP_Trap_Handling_Supported is True, then the behavior of subprograms and pragmas related to floating-point trap handling, or when objects of type FP_Trap_Management are declared, is as specified here.

Note that the value of FP_Trap_Handling_Supported does not affect a program's behavior when objects of type FP_Status_Flags_Management or FP_Retrospective_Diagnostics are declared, since these types are not considered to be among this specification's trap handling features.

Management of Status Flags and Traps

Whenever an object of type FP_Status_Flags_Management is initialized (via the primitive procedure Initialize), the state of the floating-point status flags is saved in this object, and all floating-point status flags are cleared. Whenever an object of this type is finalized (via the primitive procedure Finalize), the state of the status flags is restored to that saved during initialization, with the following exceptions: If the primitive procedure Update_FP_Status_Flags is called before finalization, the flags corresponding to the array elements that are set in the argument New_Status_Flags of the last call to Update_FP_Status_Flags remain set after finalization ends. Otherwise, the status flags that are set at the time finalization begins remain set after finalization ends.

Note that declaring an object of this type allows one to write functions that (almost simultaneously) return a value and signal (floating-point) exceptions, in much the same way as the basic arithmetic operations.

The same behavior occurs whenever an object of type FP_Retrospective_Diagnostics is initialized or finalized, except that in addition, if any status flags would remain set after finalization ends, even in the case of all status flags being clear at the time Initialize is called, a message is appended to the current default error file detailing which status flags would have remained set in such a case. If the primitive procedure Raise_Exception is called before finalization, then the exception Exception_To_Raise indicates is raised if any of the status flags indicated by FP_Exceptions would remain set after finalization ends.

The same behavior as when objects of type FP_Status_Flags_Management are declared occurs whenever an object of type FP_Trap_Management is initialized or finalized, except that in addition, when such objects are initialized, the set of which floating-point exception types have trap handlers associated with them, and what those trap handlers are, are saved. Also, whenever an object of this type is finalized, the set of which floating-point exception types have trap handlers associated with them, and what those trap handlers are, are restored to that saved during initialization.

Although the behavior when an object of type FP_Trap_Management is declared is implementation defined when the value of FP_Trap_Handling_Supported is False, at least the behavior when objects of type FP_Status_Flags_Management are declared must occur.

Enabling and Disabling Trap Handlers

The subprogram Install_FP_Trap_Handler can be used to associate a trap handler with one or more floating-point exceptions. The parameter Exception indicates with which exception or exceptions the trap handler Handler should be associated. If the value of Handler is null, the default response for the specified exception or exceptions is restored, that is, traps for the specified exception or exceptions are disabled. The remaining parameters are objects available for the given trap handler to use if needed. Depending on which trap handler is being installed, the parameters Counter and Substitute_Value may be checked to insure they are not null.

In addition to this, the function Install_FP_Trap_Handler returns a value that indicates which floating-point exception types, if any, have trap handlers associated with them at the time control returns to this function's caller.

Whenever a (nonnull) trap handler installed with Install_FP_Trap_Handler is invoked, the status flag corresponding to the exception being handled is cleared.

Trap Handlers Provided by `Standard_FP_Arithmetic`

The package `Standard_FP_Arithmetic` provides several useful trap handlers: Tally_Exception, Presubstitute, Raise_Exception, and Abort_Task. (The latter is not the same as the procedure with the same name in Ada.Task_Identification. Similarly, Raise_Exception is not the same as the procedure with the same name in Ada.Exceptions.)

Tally_Exception is intended to be associated with the overflow or underflow exception. It simply increments the counter specified at the time this trap handler is installed with Install_FP_Trap_Handler, and causes computation to continue with the wrapped value the IEEE Standard requires floating-point arithmetic engines to make available to trap handlers in such cases.

This trap handler is useful, for example, in computing the product of a series of numbers, since intermediate products can overflow or underflow, even if the final result is perfectly within the range of the data type used.

Presubstitute causes the Substitute_Value specified at the time this trap handler is installed with Install_FP_Trap_Handler to be substituted (possibly with the sign reversed) in place of the result that would have been produced had there been no trap handler associated with the exception that caused this trap handler to be invoked. If the value of Copy_Sign_Of_Result is True when this trap handler is installed with Install_FP_Trap_Handler, then the sign of the substituted value is that of the result that would have been produced had all trap handlers been disabled; otherwise, the sign of Substitute_Value is not changed.

Raise_Exception has the same effect as a raise_statement naming the exception specified at the time this trap handler is installed with Install_FP_Trap_Handler.

Abort_Task has the same effect as the abort_statement for the current task.

User-Defined Trap Handlers

Although details as to how trap handlers should be written are implementation defined, `Standard_FP_Arithmetic` provides several subprograms that could be useful when writing trap handlers:

The value Get_Operation returns indicates what operation caused the current trap handler to be invoked. The enumeration literal FP_Conversion refers to a conversion operation whose result is of some floating-point type; Integer_Conversion refers to a conversion from a binary floating-point format to an integer format; Decimal_Conversion refers to conversion from a binary floating-point format to a decimal format. The type FP_Operations may have other implementation-defined enumeration literals in addition to those listed in this specification.

Get_Operands provides the operand or operands of the operation that caused the current trap handler to be invoked. Upon return, the parameters Left_Valid, Middle_Valid, and Right_Valid indicate which operand or operands were actually available.

In cases in which the IEEE Standard requires floating-point arithmetic engines to make a result available to trap handlers, Get_Result provides that result. Upon return, the parameter Result_Valid indicates if the result was actually available.

Assuming normal computation is to be resumed upon return from the current trap handler, Set_Result specifies what value should be used in place of the result that would have been produced had there been no trap handler associated with the exception that caused the current trap handler to be invoked. Upon return, the parameter Result_Set indicates if it was actually possible to set the result to that specified.

A.4 Pragmas Related to Floating-Point Arithmetic

A.4.1 Pragmas Related to the Accuracy of Results

The following pragmas are related to the precision of results of floating-point arithmetic operations: Minimum_FP_Precision, Produce_Model_Number, Round_Once, and Allow_Fused_Multiply_Add. These pragmas may appear in any declarative region or as configuration pragmas. If any of these pragmas are used as configuration pragmas, the scope of their effect is all the compilation units, if any, appearing in the compilation. Otherwise, the scope of their effect is from the point in which they appear to the end of the innermost enclosing declarative region. If the scopes of more than one instance of the same pragma overlap, the one that has effect at a given point in a program is the one whose scope is the innermost one among the scopes including the given point in the program.

The meaning of these pragmas are explained in the following subsections.

Pragma Minimum_FP_Precision

The following pragma is related to floating-point evaluation formats:

```
Minimum_FP_Precision
   (floating_point_definition|floating_point_data_type)
```

The precision of the results of floating-point arithmetic operations that are performed at run time in the regions in which this pragma is in effect is that corresponding to the specified floating-point data type, provided the underlying arithmetic engine supports producing results in this precision in a manner that fully conforms to the IEEE Standard. (However, results may suffer rounding more than once, unless the pragma Round_Once is in effect - see below.) Otherwise, results are implementation dependent.

Using this pragma enables one to get bit-for-bit identical results on different computers more often, at the expense of possibly adversely affecting performance. When targeting processors such as those conforming to Intel's x86 architecture, an implementation of this specification could make use of that architecture's precision control bits in order to reduce the precision of floating-point arithmetic operations to that specified by an instance of this pragma, if necessary. Results of these operations might subsequently be spilled to memory and reloaded into registers at a later time. Alternatively, results of floating-point arithmetic operations could purposely be stored to memory and subsequently reloaded into registers in order to reduce their precision. In either case, results may suffer rounding more than once; this is because values stored in memory could differ from the ones that were originally held in registers, particularly if the format used to represent the value in memory is different from that of the original register.

Pragma Produce_Model_Number

The following pragma affects floating-point results produced by arithmetic operations:

```
Produce_Model_Number (true|false)
```

In any region in which this pragma is enabled, the result of any arithmetic operation that is performed at run time and whose result type is derived from a floating-point type declared in `Standard_FP_Arithmetic` is a model number of the base type of the result type. (However, results may suffer rounding more than once, unless the pragma Round_Once is in effect—see below.) In the absence of this pragma, arithmetic is performed as if this pragma were disabled.

Of course, even when this pragma is disabled, results of floating-point operations can still be model numbers; that is, results are not forbidden to be model numbers. However, if results are allowed to not be model numbers, performance can be improved in some implementations.

Pragma Round_Once

The following pragma affects floating-point results produced by arithmetic operations:

```
Round_Once (true|false)
```

In any region in which this pragma is enabled, the result of any arithmetic operation that is performed at run time and whose result type is derived from a floating-point type declared in `Standard_FP_Arithmetic` must not differ from that which would be obtained if rounding were performed only once in the process of arriving at the result. In the absence of this pragma, arithmetic is performed as if this pragma were disabled.

Of course, even when this pragma is disabled, results of floating-point operations can still be arrived at as if rounding had been performed just once; that is, results are not required to suffer more than one rounding. However, if results are allowed to be rounded more than once, performance can be improved significantly in some implementations.

Pragma Allow_Fused_Multiply_Add

The following pragma affects floating-point results produced by arithmetic operations:

```
Allow_Fused_Multiply_Add (true|false)
```

If the underlying arithmetic engine is able to multiply two operands without fully rounding the product, and add a third operand to (or subtract a third operand from) this not-fully-rounded product, the language processor must not make use of this feature in any region in which this pragma is disabled. In the absence of this pragma, arithmetic is performed as if this pragma were enabled.

Some algorithms do not produce the intended results if fused multiply-add instructions are used. On the other hand, other algorithms do not produce the intended results if fused multiply-add instructions are not used. In the latter case, this pragma is not useful; the multiply_add function should be used instead.

A.4.2 Pragmas Related to Rounding Modes

The goals of this specification as far as rounding modes are concerned are as follows:

1. *Comply with the IEEE Standard's requirements concerning rounding modes, including the requirement that the user be able to "set, sense, save, and restore" modes such as the rounding mode. This appears to imply that it must be possible to change the rounding mode dynamically during the execution of a program. (Certainly the authors of the IEEE Standard intended that this be a requirement - see [56], for example.) An important use of this ability is to attempt disproving the stability of an algorithm by running it with a different rounding mode.*

2. *Make it possible to indicate that a certain rounding mode is to be used for all arithmetic operations (except comparison and remainder) within a given subprogram or entire package, provided they are to be performed as if at run time. (That is, this goal does not apply to operations which must be performed as if at translation time.) This can simplify the task of determining whether a code fragment will produce its intended result, regardless of what the status of the processor is before execution of the code fragment in question. Also, this can allow language translators to select opcodes that specify that a fixed rounding mode be used on an instruction-by-instruction basis when targeting processors that allow this, such as implementations of the DEC Alpha architecture.*

3. *Make it possible to specify what rounding mode to use on an operation-by-operation basis. A facility that enables this would not be expected to receive widespread use, but could be useful, for example, when doing interval arithmetic. Of course, language translators could take advantage of a processor's static rounding modes, if available.*

4. *Make it possible to specify what rounding mode to use in operations appearing within a given block of code smaller than its enclosing subprogram. Note that if goal 1) above is achieved, this goal is also achieved, though a side effect would likely be that the specified rounding mode would also be in effect in any subprograms called within the given block of code.*

5. *Make it possible to indicate whether the specified rounding mode is to be used in operations that are to be performed as if at translation time, run time, or both.*

6. *Avoid inherently requiring excessive overhead when manipulating the rounding mode, since on many processors, accessing the register that contains the rounding mode in effect takes several clock cycles and disrupts the floating-point pipeline. Otherwise features having to do with the rounding mode will often be avoided in practice, even in cases in which it would have been desirable to use these features. In particular, one technique that could be used to reduce this overhead on such processors is to keep a copy of the rounding mode in some thread-specific memory location. This copy could be checked, for example, to determine if it is actually necessary to modify the register containing the rounding mode. So a goal of this specification is to allow "caching" the rounding mode in this manner (but not require this "caching"), and make it clear in what situations the cached value can become stale.*

When execution of a program begins, the rounding mode in effect is To_Nearest.
The following pragmas are related to rounding modes:

```
Use_Rounding_Mode (Rounding_Mode)
Modifies_Rounding_Mode (Subprogram_Name [, Rounding_Mode])
```

The meaning of these pragmas is explained in the following subsections.

Pragma Use_Rounding_Mode

The pragma Use_Rounding_Mode may appear in any declarative region or as a configuration pragma. If used as a configuration pragma, the scope of its effect is all the compilation units, if any, appearing in the compilation. Otherwise, if this pragma is not used as a configuration pragma, the scope of its effect is from the point in which it appears to the end of the innermost enclosing declarative region. If the scopes of more than one instance of the pragma Use_Rounding_Mode overlap, the one that has effect at a given point in a program is the one whose scope is the innermost one among the scopes including the given point in the program.

The specified rounding mode must be in effect in the regions in which this pragma is in effect when any floating-point operation is performed at translation time, whether or not the Ada 95 Reference Manual [11] requires the operation to be performed at translation time. In addition, the language translator may assume that the specified rounding mode is in effect in the regions in which this pragma is in effect, even if calls to the subprogram Rounding (or any other subprogram) appear in these regions.

If the specified rounding mode is Current_Rounding_Mode, or if this pragma is not in effect in a given region of code, then the rounding mode in effect for all floating-point operations in the region of code in question that are required to be performed at translation time must be To_Nearest, and the language processor must assume that the rounding mode in effect at run time in the region of code in question is unknown.

One way to make it possible to indicate that a certain rounding mode is to be used for all arithmetic operations within a given subprogram could have been to name the subprogram of interest in an instance of the pragma Use_Rounding_Mode. The language translator could then arrange for the rounding mode in effect to be that specified whenever the named subprogram began execution. Upon returning from such a subprogram, the rounding mode in effect at the time just before the subprogram was called could be restored. However, the caller's rounding mode probably should not be restored if, before control is transferred to the subprogram's caller, the subprogram Rounding were called with an argument other than Current_Rounding_Mode, or if any other subprogram associated with the pragma Modifies_Rounding_Mode were called.

If there were such a pragma, more efficient code might be generated if the instance of this pragma associating a given subprogram with a rounding mode were to appear in the package specification in which the given subprogram is declared, rather than in the corresponding package body. The reason is that in the former case, the language processor might be able to determine that the rounding mode in effect at the time the given subprogram is called is already that specified in this pragma, and that there is therefore no need to modify the rounding mode.

Note that associating a subprogram with the pragma Use_Rounding_Mode would seem to be at odds with (directly or indirectly) making a call from that subprogram to another subprogram that modifies the rounding mode. However, a programmer who codes such a call would most likely expect that the newly established rounding mode will be in effect unless explicitly overridden. Thus, some mechanism would be needed to determine if a subprogram associated with the pragma Use_Rounding_Mode should restore its caller's rounding mode.

One way to keep track of this would be to maintain a special thread-specific stack in which every stack element contains a "dirty" bit. Whenever a subprogram sets the rounding mode (even if the rounding mode is not actually changed), the "dirty" bit at the top of the stack would be set. If the top of the stack is marked dirty, a new stack element (with the "dirty" bit cleared) would be pushed onto this stack whenever a subprogram associated with the pragma Use_Rounding_Mode is called. In addition, a pointer to the top of this stack could be stored in the called subprogram's stack frame. A stack element would be popped off this special stack whenever a subprogram's stack pointer does not point to the top of the stack. Whenever a subprogram associated with the pragma Use_Rounding_Mode returns to its caller, its caller's rounding mode would be restored only if the "dirty" bit at the top of the stack is clear. Note that there is no overhead with this scheme when calling a subprogram that is not associated with the pragma Use_Rounding_Mode.

All this seems fairly complicated, which is why this approach was abandoned in favor of having a limited controlled type capable of saving and restoring the rounding mode at the beginning and end, respectively, of a given declarative region.

Pragma Modifies_Rounding_Mode

The pragma Modifies_Rounding_Mode may appear in any declarative region. Language processors may assume that unless a subprogram is associated with the pragma Modifies_Rounding_Mode, the rounding mode will not be changed upon returning from a call to the subprogram. If a rounding mode is present in an instance of this pragma, the language processor may assume that upon returning from a call to the specified subprogram, the rounding mode will be set to that specified in the pragma.

A.4.3 Other Pragmas

Pragmas Disregard_FP_Side_Effects

The following pragma has to do with side effects produced by floating-point operations:

```
Disregard_FP_Side_Effects (true|false)
```

This pragma may appear in any declarative region or as a configuration pragma. If this pragma is used as a configuration pragma, the scope of its effect is all the compilation units, if any, appearing in the compilation. Otherwise, the scope of its effect is from the point in which it appears to the end of the innermost enclosing declarative region. If the scopes of more than one instance of this pragma overlaps, the one that has effect at a given point in a program is

the one whose scope is the innermost one among the scopes including the given point in the program.

In any region in which this pragma is enabled, operations with an operand or result type derived from a floating-point type declared in `Standard_FP_Arithmetic` are allowed (but not required) to be performed as if there were no side effects beyond delivering a result. In particular, status flags might not be set and floating-point traps (if supported and enabled) might not be invoked at the time or as required by the IEEE Standard. However, such operations must otherwise be performed as specified by the IEEE Standard and this specification, and in a manner consistent with the definition of the Ada language. In the absence of this pragma, arithmetic is performed as if this pragma were disabled.

Pragma Allow_Nonstandard_FP_Arithmetic

The following pragma affects floating-point results produced by arithmetic operations:

```
Allow_Nonstandard_FP_Arithmetic (true|false)
```

This pragma may appear in any declarative region or as a configuration pragma. If this pragma is used as a configuration pragma, the scope of its effect is all the compilation units, if any, appearing in the compilation. Otherwise, the scope of its effect is from the point in which it appears to the end of the innermost enclosing declarative region. If the scopes of more than one instance of this pragma overlaps, the one that has effect at a given point in a program is the one whose scope is the innermost one among the scopes including the given point in the program.

In any region in which this pragma is enabled, any arithmetic operation with an operand or result type derived from a floating-point type declared in `Standard_FP_Arithmetic` is allowed (but not required) to be performed in a manner inconsistent with the IEEE Standard or this specification. However, such operations must still be performed in a manner consistent with the definition of the Ada language. In the absence of this pragma, arithmetic is performed as if this pragma were disabled.

Bibliography

[1] *68000 Family Programmer's Reference Manual (Includes CPU32 Instructions)*. Motorola Inc., 1992. Part No. M68000PM/AD.

[2] *Alpha Architecture Handbook*. Digital Equipment Corporation, Apr. 1992.

[3] *IA-64 Application Developer's Architecture Guide*. Intel Corporation, May 1999. Order No. 245188-001.

[4] Java Development Kit (JDK) version 1.2 summary of new features. 1998. Unpublished manuscript available at http://java.sun.com/products/jdk/1.2/docs/relnotes/features.html.

[5] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard Company, second edition, Sep. 1992. Part No. 09740-90039.

[6] *Pentium Processor Family Developer's Manual—Volume 3: Architecture and Programming Manual*. Intel Corporation, July 1995. Order No. 241430-004.

[7] *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics, Motorola Inc., 1993. Part No. MPC601UM/AD REV 1.

[8] *PowerPC Microprocessor Family: The Programming Environments*. IBM Microelectronics, Motorola Inc., 1994. Part No. MPCFPE/AD.

[9] *T9000 Transputer Instruction Set Manual*. SGS-THOMSON Microelectronics Limited.

[10] *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983*. United States Department of Defense, 1983.

[11] *Information technology — Programming languages — Ada, Ada Reference Manual (International Standard ISO/IEC 8652:1995(E))*. International Organization for Standardization (ISO), International Electrotechnical Commision (IEC), 1995.

[12] Apple Computer, Inc. *Apple Numerics Manual*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1988.

[13] L. Augustsson, D. Barton, and et al. Haskell 98: a non-strict, purely functional language. Feb. 1999. Unpublished manuscript available at http://www.haskell.org/definition/.

[14] W. S. Brown. A simple but realistic model of floating-point computation. *ACM Trans. Math. Softw.*, 7(4):445–480, Dec. 1981.

[15] *Information technology — Programming languages — C++ (International Standard ISO/IEC 14882:1998).* International Organization for Standardization (ISO), International Electrotechnical Commision (IEC), 1998.

[16] *Information technology — Programming languages — C (International Standard ISO/IEC 9899:1990).* International Organization for Standardization (ISO), International Electrotechnical Commision (IEC), 1990.

[17] *Programming languages — C.* International Organization for Standardization (ISO), International Electrotechnical Commision (IEC), Aug. 1998. Committee Draft available at http://wwwold.dkuug.dk/jtc1/sc22/open/n2794/.

[18] W. J. Cody and J. T. Coonen. Algorithm 722; functions to support the IEEE standard for binary floating-point arithmetic. *ACM Trans. Math. Softw.*, 19(4):443–451, Dec. 1993.

[19] J. T. Coonen. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic.* PhD thesis, University of California, Berkeley, Berkeley, California, 1984.

[20] J. T. Coonen. Fp signs. Aug. 1996. Private electronic mail message.

[21] J. T. Coonen. A proposal for RealJava. July 1997. Electronic mail message sent to the numeric-interest@validgh.com mailing list.

[22] J. D. Darcy. *Borneo 1.0: Adding IEEE 754 Floating Point Support to Java.* Master's thesis, University of California, Berkeley, Berkeley, California, May 1998. Available at http://www.cs.berkeley.edu/~darcy/Borneo/spec.html.

[23] J. W. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.*, 5(4):887–919, Dec. 1984.

[24] R. B. K. Dewar. Ada interface for the IEEE standard for binary floating-point arithmetic. Jan. 1992. Private communication via electronic mail.

[25] Programming language APL, extended — international standards organisation — committee draft 1. Jan. 1993. Unpublished manuscript available at ftp://gatekeeper.dec.com/pub/plan/apl/aplcd1.ps.

[26] C. Farnum. Compiler support for floating-point computation. *Software—Practice and Experience*, 18(7):701–709, July 1988.

[27] R. J. Fateman. High-level language implications of the proposed IEEE floating-point standard. *ACM Trans. Prog. Lang. Syst.*, 4(2):239–257, Apr. 1982.

[28] S. Feldman. Language support for floating point. In J. K. Reid, editor, *The Relationship between Numerical Computation and Programming Languages*, pages 263–273, North-Holland Publishing Company, Amsterdam, 1982.

[29] S. A. Figueroa. Comments on sun's proposal for extension of java floating point in jdk 1.2. Aug. 1998. Electronic mail message sent to the numeric-interest@validgh.com mailing list.

[30] S. A. Figueroa. *Supporting the IEEE Standard for Binary Floating-Point Arithmetic in a High-Level Language*. Master's thesis, University of Kentucky, Lexington, Kentucky, Sep. 1986.

[31] *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*. American National Standards Institute, Inc., New York, 1978.

[32] American national standard for information systems programming language Fortran, ANSI X3.9-198x, draft S8, version 112. *Fortran Forum*, 8(4), Dec. 1989.

[33] D. M. Gay. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, Nov. 1990.

[34] D. Goldberg. The design of floating-point data types. *ACM Letters on Programming Languages and Systems*, 1(2):138–151, June 1992.

[35] D. Goldberg. Re: square root and double rounding. July 1995. Private communication via electronic mail.

[36] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[37] R. A. Golliver. The Ivory brand of Java. Aug. 1997. Unpublished manuscript.

[38] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1996.

[39] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Prog. Lang. Syst.*, 18(2):139–174, March 1996.

[40] D. G. Hough. 1991 discussion on 128-bit floating-point arithmetic. July 1995. Electronic mail message sent to the numeric-interest@validgh.com mailing list.

[41] D. G. Hough. Re: satan loves extended precision. Jan. 1995. Electronic mail message sent to the numeric-interest@validgh.com mailing list.

[42] D. G. Hough. Re: square root and double rounding. June 1995. Private communication via electronic mail.

[43] D. G. Hough. Signbit. Aug. 1996. Private electronic mail message.

[44] T. E. Hull, M. S. Cohen, J. T. M. Sawshuk, and D. B. Wortman. Exception handling in scientific computing. *ACM Trans. Math. Softw.*, 14(3):201–217, Sep. 1988.

[45] *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic (International Standard ISO/IEC 10967-1:1994(E))*. International Organization for Standardization (ISO), International Electrotechnical Commision (IEC), 1994.

[46] *Binary Floating-Point Arithmetic for Microprocessor Systems*. International Electrotechnical Commision (IEC), 1989. IEC 60559:1989.

[47] *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1985. ANSI/IEEE Std 754-1985.

[48] *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1987. ANSI/IEEE Std 854-1987.

[49] *80287XL/XLT CHMOS III Math Coprocessor*. Intel Corporation, Santa Clara, California, May 1990. Order No. 290376-001.

[50] *8087 Math Coprocessor*. Intel Corporation, Santa Clara, California, Sep. 1989. Order No. 205835-007.

[51] Java Grande Forum Numerics Working Group. Improving Java for numerical computation. Oct. 1998. Unpublished manuscript available at http://math.nist.gov/javanumerics/reports/jgfnwg-01.html.

[52] W. Kahan. Nov. 1993. Private communication.

[53] W. Kahan. Analysis and refutation of the LCAS. *SIGNUM Newsletter*, 26(3):2–15, July 1991.

[54] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. May 1996. Unpublished manuscript available at http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps.

[55] W. Kahan. Miscalculating area and angles of a needle-like triangle. July 1999. Unpublished manuscript available at http://http.cs.berkeley.edu/~wkahan/ieee754status/Triangle.pdf.

[56] W. Kahan and J. D. Darcy. How Java's floating-point hurts everyone everywhere. June 1998. Unpublished manuscript available at http://http.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.

[57] R. Kirchner. Re: alpha 21064 floating point rounding modes can be real slow. Feb. 1994. Electronic message posted to the comp.arch Usenet newsgroup.

[58] W. D. Klinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101, White Plains, New York, June 1990.

[59] Limbo language definition. 1998. Unpublished manuscript available at http://www.lucent-inferno.com/Pages/Developers/Documentation/Limbo_Ref20/langdef.htm.

[60] T. Lindholm. Re: comments on sun's proposal for extension of java floating point in jdk 1.2. Aug. 1998. Electronic mail message sent to the numeric-interest@validgh.com mailing list.

[61] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1999.

[62] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991. Also available at http://www.ssw.uni-linz.ac.at/Research/Papers/Moe91a.html.

[63] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[64] E. Nikitin and M. van Acken. The OOC reference manual. July 1999. Unpublished manuscript available at http://www.uni-kl.de/OOC/OOCref/OOCref_toc.html.

[65] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*, chapter Appendix A: Computer Arithmetic, pages A–29. Morgan Kaufmann Publishers, Inc., San Mateo, California, first edition, 1990. (The author of Appendix A is D. Goldberg).

[66] D. M. Priest. Differences among IEEE 754 implementations. 1997. http://www.validgh.com/goldberg/addendum.html (to be published in the next release of Sun's *Numerical Computation Guide* as an addendum to Appendix D, David Goldberg's "What Every Computer Scientist Should Know About Floating-Point Arithmetic"—see [36]).

[67] D. M. Priest. Re: square root and double rounding. June 1995. Electronic mail message sent to the numeric-interest@validgh.com mailing list.

[68] D. M. Priest. Re: typos in NCG. May 1997. Private electronic mail message.

[69] C. Severance. An interview with the old man of floating-point. Feb. 1998. Unpublished manuscript available at http://http.cs.berkeley.edu/~wkahan/ieee754status/754story.html.

[70] SPARC International, Inc. *The SPARC Architecture Manual*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[71] G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 112–126, White Plains, New York, June 1990.

[72] SunSoft, Inc. Freely distributable math library. Source code available at http://www.netlib.org/fdlibm/.

[73] SunSoft, Inc. *Numerical Computation Guide*. Nov. 1995. Part No. 802-3254-10.

[74] SunSoft, Inc. *SunOS Reference Manual Pages*. Dec. 1996. Distributed with SunOS.

[75] J. Thomas. Floating-point C extensions. Aug. 1991. NCEG Document No. 91-028.

[76] B. Venners. The state of the Java virtual machine. Unpublished manuscript available at http://java.sun.com/events/jbe/98/features/jvm.html.

[77] B. A. Wichmann. Notes on Ada numerics for Ada 9X. Feb. 1992. Private communication via electronic mail.