On the need for predictable floating-point arithmetic in the programming languages Fortran 90 and C / C++

DENNIS VERSCHAEREN, ANNIE CUYT AND BRIGITTE VERDONK
University of Antwerp (Campus UIA)
InfoProc
Department of Mathematics and Computer Science
Universiteitsplein 1
B-2610 Antwerp - Belgium
E-mail: Dennis.Verschaeren@uia.ua.ac.be
E-mail: Annie.Cuyt@uia.ua.ac.be
E-mail: Brigitte.Verdonk@uia.ua.ac.be

Abstract. During the past decade the IEEE 754 standard for binary floating-point arithmetic has been very successful. Many of today's hardware platforms conform to this standard and compilers should make floating-point functionality available to programmers through high-level programming languages. They should also implement certain features of the IEEE 754 standard in software, such as the input and output conversions to and from the internal binary representation of numbers. In this report, a number of Fortran 90 and C / C++ compilers for workstations as well as personal computers will be screened with respect to their IEEE conformance. It will be shown that most of these compilers do not conform to the IEEE standard and that shortcomings are essentially due to their respective programming language standards which lack attention for the need of predictable floating-point arithmetic.

Categories and subject descriptors: G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*; D.3.0 [**Programming Languages**]: General—*Standards*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

AMS subject classifications: 68N15, 68N20

General terms: floating-point, arithmetic

Additional key words and phrases: IEEE floating-point standard, Fortran 90, C, C++

1 – Introduction

The IEEE standard for floating-point arithmetic describes how floating-point numbers should be handled in hardware and software. The advantages and disadvantages of the IEEE standard will not be discussed, but standardization of floating-point arithmetic is needed for writing portable source code and generating reliable output. In this paper, the discussion will be restricted to the programming languages Fortran 90 and C / C++, as the majority of available mathematical software is written in Fortran or C. Fortran 90 is the modern descendent of Fortran 77, and according to [Metcalf & Reid 1996]: "(...) has always been the principal language used in the fields of scientific, numerical, and engineering programming (...)". On the other hand, the languages C and especially C++ are increasingly used in scientific computing. It will be shown that most of the popular PC and workstation compilers which were tested do not conform to the IEEE standard, even though the underlying hardware platforms are IEEE-conforming.
Section 2 highlights some important aspects of the IEEE floating-point standard. Sections 3 through 6 then compare the IEEE requirements with the specifications for floating-point support in the international standards of the programming languages Fortran 90 and C, and in the current C++ draft standard, which is based on the C programming language for its floating-point part. This discussion also includes floating-point support in the upcoming Fortran 2000 standard and the final proposal of the Numerical C Extensions Group (NCEG) to be incorporated into a revision of the C standard. Section 7 summarizes some of the obtained results when screening popular Fortran 90 and C / C++ compilers. A full report, with an appendix discussing these test results in more detail, and the test programs can be found at URL ftp://hwins.uia.ac.be/pub/cant/SIGPLAN/.

2 – The IEEE floating-point standard

The [IEEE 1985] and [IEEE 1987] standards for binary and radix-independent floating-point arithmetic respectively, are defined as: "(...) a family of commercially feasible ways for new systems to perform floating-point arithmetic". Both standards differ only slightly: the latter standard can be seen as a generalization of [IEEE 1985] in the sense that it allows the radix or base of the floating-point set under consideration to be 2 (binary) or 10 (decimal). In what follows, we will refer to the IEEE standard when considering [IEEE 1985] for binary implementations. The following paragraphs highlight aspects of the IEEE standard which are relevant to floating-point functionality in high-level programming languages.

2.1 - Formats

The IEEE standard: "(...) defines four floating-point formats in two groups, basic and extended, each having two widths, single and double". An implementation conforming to this standard always supports single format, which is the narrowest format. A second, wider basic format, is called double format. The two extended formats, single-extended and double-extended, can be encoded in an implementation-dependent way. The extended format corresponding to the widest basic format should be supported (where the use of the word "should" means that it is not obligatory in order to conform to the IEEE standard).

2.2 – Rounding

A second paragraph in the IEEE standard requires implementations to provide four rounding modes which take: "(...) a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's format (...)". An implementation shall provide round to nearest as the default rounding mode, and three user-selectable directed rounding modes: round to $+\infty$, round to $-\infty$ and round to 0. Rounding affects all operations, including conversions, except comparisons and remainder (see 2.3).

2.3 – Operations

According to the IEEE standard, all conforming implementations provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, convert between different floating-point formats, convert between floating-point and integer formats, round to a floating-point integer value, convert between internal floating-point representations and decimal strings and compare. It is important to note that the remainder is defined: "(...) by the mathematical relation $r = x - y \times n$, where n is the integer nearest the exact value x/y (...)". Each operation must be performed as if it first produced an intermediate, infinitely precise result which is then correctly rounded to fit its destination's format, except binary to decimal and decimal to binary conversion for which exact rounding is only required for a large but limited subset of values. Nowadays, it is possible to convert from binary to decimal and decimal to binary in ways that always yield the correctly rounded result, with little time penalty in common cases [Gay 1990]. Implementations should also provide an unordered operator, which returns true if at least one operand is NaN (Not a Number).

2.4 – Infinity, NaNs, and Signed Zero

All formats have to provide distinct representations for $+0$, -0, $+\infty$, $-\infty$, at least one quiet NaN and at least one signaling NaN. Infinities have the same meaning as in mathematics. They are interpreted in the affine sense, that is $-\infty <$ (every finite number) $< +\infty$. Infinities are created from finite operands by overflow and division by zero. Invalid operations (e.g. 0/0) and operations involving NaNs shall deliver a quiet NaN as their result. Moreover, invalid operations and operations involving signaling NaNs also raise the invalid exception (see 2.5). High-level languages have to support these special representations and should provide strings for input/output purposes.

2.5 – Exceptions

There are five types of floating-point exceptions which must be signaled when detected: invalid operation, division by zero, overflow, underflow and inexact. These exceptions entail: "(...) setting a status flag, taking a trap, or possibly doing both. With each exception should be associated a trap under user control. The default response to an exception is to proceed without a trap" (see 2.6). A status flag is set on: "(...) any occurrence of the corresponding exception when no corresponding trap occurs". The user must be able to test and alter each status flag individually. The only exceptions which can coincide are inexact with overflow and inexact with underflow.

2.6 – Traps

The IEEE standard recommends, but does not mandate, an alternative response to an exception, namely trapping. Trap handlers, interrupt handlers or signal handlers are all general terms for what in this context can be called floating-point exception handlers. When a floating-point exception whose trap is enabled, is signaled, the execution of the program in which the exception occurs, is suspended, and the exception handler which was previously specified by the user, is activated. An exception handler behaves like a subroutine which can return a value instead of the exceptional operation's result (such as ∞ or NaN). Exception handling is often overlooked (or avoided) by programming language standards and cannot be seen independently from a general interrupt handling scheme.

3 – The ISO/IEC Fortran 90 standard

According to [Metcalf & Reid 1996]: "Fortran's superiority has always been in the area of numerical, scientific, engineering, and technical applications". The ISO/IEC Fortran standard [ISO/IEC 1991], informally known as Fortran 90, was published in 1991. With the announced Fortran 2000 standard currently under discussion (see 4), this section reviews why the Fortran 90 standard does not assure IEEE compatibility.

3.1 – Formats

The ways in which numbers are stored internally by a computer are not the concern of the Fortran standard. The standard specifies: "The real type has values that approximate the mathematical real numbers. A processor must provide two or more approximation methods that define sets of values for data of type real". There is a default real type and a processor-dependent number of other kinds of type real, where kind is a non-negative integer value which is used to identify the various kinds of type real. Floating-point formats are not specified. To support portability between different processors, the inquiry function `SELECTED_REAL_KIND(A,B)` returns the kind value of a real type on the current processor with at least `A` significant decimal digits and a decimal exponent range of at least `-B` to `+B`.

3.2 – Rounding

The standard explicitly excludes: "The physical properties of the representation of quantities and the method of rounding, approximately, or computing numeric values on a particular processor".

3.3 – Operations

Fortran 90 does not provide all basic IEEE operations. It is possible to add, subtract, multiply, divide and extract the square root of any real number, but the IEEE remainder operation is missing. Instead, Fortran 90 defines two other functions:

- `MOD(A,P) = A - INT(A/P) * P` where `INT` rounds to 0
- `MODULO(A,P) = A - FLOOR(A/P) * P` where `FLOOR` rounds to $-\infty$

The result is processor-dependent if `P` equals 0. IEEE requires the division `A/P` to be rounded to the nearest integer while, if $P = 0$, the operation is invalid and NaN must be returned or a trap invoked. The following function converts floating-point numbers between all supported formats, and between integer formats and floating-point formats, but as stated above (see 3.2), rounding is not specified:

- `REAL(A,KIND)` converts any real or integer `A` to a real of the given `KIND`

For conversions of floating-point formats to integer formats, the standard provides separate functions for each of the four rounding modes:

- `NINT(A,KIND)` converts a real `A` to the nearest integer of the given `KIND`
- `INT(A,KIND)` converts a real `A` to the integer of the given `KIND`, rounded to 0
- `CEILING(A)` converts a real `A` to the default integer type, rounded to $+\infty$
- `FLOOR(A)` converts a real `A` to the default integer type, rounded to $-\infty$

Note that the last two functions only provide conversions to the default integer type, which is not predetermined by the standard. Two functions round floating-point numbers to integral floating-point values (rounding to $+\infty$ or $-\infty$ is not available):

- `AINT(A,KIND)` converts a real `A` to the integral real of the given `KIND`, rounded to 0
- `ANINT(A,KIND)` converts a real `A` to the nearest integral real of the given `KIND`

Fortran, like any programming language, allows decimal input of floating-point values. The Fortran standard mentions that: "The significand may be written with more digits than a processor will use to approximate the value (...)", but the rounding algorithm for either input or output is not specified by the standard. The Fortran standard provides the six required IEEE comparisons. It does not define an unordered predicate.

3.4 – Infinity, NaNs, and Signed Zero

The Fortran standard does not discuss any special representations.

3.5 – Exceptions, Traps

Exception handling is described as future work.

4 – Fortran 95 and Beyond

Fortran 95 is seen as a relatively minor revision of Fortran 90 with the primary emphasis on clarifications, corrections and interpretations. Especially John Reid [Reid 1996] has tried hard to get an exception handling feature, as well as other aspects of the IEEE standard, into the Fortran standard. We will discuss the current draft [ISO/IEC 1996] of a final report announced as [ISO/IEC 1997], which is expected to appear in 1997. It will enable compiler vendors to add exception handling to their compilers before its formal standardization as part of the Fortran 2000 standard.
In order to allow implementations on different hardware platforms to provide maximum value to users, the [ISO/IEC 1996] report still does not require IEEE conformance. The modules `IEEE_EXCEPTIONS`, `IEEE_ARITHMETIC` and `IEEE_FEATURES` provide support for floating-point exceptions and IEEE arithmetic. They are not required for vendors with no IEEE hardware.

4.1 – Formats

The inquiry function `IEEE_SUPPORT_DATATYPE([X])` (where `[X]` denotes that the argument `X` is optional) in the module `IEEE_EXCEPTIONS` returns true if [ISO/IEC 1996]: "(...) the processor supports IEEE arithmetic for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X` (...) Here, support means employing an IEEE data format and performing the operations $+$, $-$, and $\times$ as in the IEEE standard whenever the operands and result all have normal values". It is sufficient to implement the arithmetic operators in only one of the four IEEE rounding modes (see also 4.2). The Fortran report does not precise which rounding mode.

4.2 - Rounding

The function IEEE_SUPPORT_ROUNDING(ROUND_VALUE[,X]) returns true if a call to the function IEEE_SET_ROUNDING_MODE(ROUND_VALUE) changes the current rounding mode to ROUND_VALUE. The function IEEE_GET_ROUNDING_MODE(ROUND_VALUE) can be used to inquire which of the four IEEE rounding modes (or IEEE_OTHER if the rounding mode is not IEEE-conforming) is in operation.

4.3 - Operations

The inquiry function IEEE_SUPPORT_DATATYPE([X]) does not guarantee (see 4.1) that all operations are executed as defined in the IEEE standard. Supplementary functions, IEEE_SUPPORT_DENORMAL([X]), IEEE_SUPPORT_DIVIDE([X]) and IEEE_SUPPORT_SQRT([X]), inquire if IEEE denormal numbers are supported and if division and square root are implemented in the IEEE sense. The IEEE remainder function is not mentioned in the report, nor is it possible to inquire if conversions are available that take into account IEEE rounding. The module IEEE_ARITHMETIC contains the unordered predicate IEEE_UNORDERED(X,Y) for arguments X and Y such that IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) return true.

4.4 - Infinity, NaNs, and Signed Zero

Although not explicitly mentioned, signed zeros should be supported if operations are executed in the IEEE sense (see 4.3). Two inquiry functions, IEEE_SUPPORT_INF([X]) and IEEE_SUPPORT_NAN([X]), return true if the processor supports special representations for infinities and NaNs for all reals (X absent) or only in the same format as the argument X.

4.5 - Exceptions

The module IEEE_EXCEPTIONS contains the functions IEEE_SUPPORT_FLAG(FLAG) to inquire if the processor supports the requested exception FLAG. The status of supported exceptions can be requested and restored individually by the functions IEEE_GET_FLAG(FLAG,FLAG_VALUE) and IEEE_SET_FLAG(FLAG,FLAG_VALUE), and as a whole by the functions IEEE_GET_STATUS(STATUS_VALUE) and IEEE_SET_STATUS(STATUS_VALUE).

4.6 - Traps

The function IEEE_SUPPORT_HALTING(FLAG) in the module IEEE_EXCEPTIONS returns true if: "(...) the processor supports the ability to control during program execution whether to abort or continue execution after an exception". The function IEEE_GET_HALTING_MODE(FLAG,HALTING) gets the halting mode of an exception and IEEE_SET_HALTING_MODE(FLAG,HALTING) sets continuation or trapping on exceptions. The report remarks that: "Halting is not precise and may occur some time after the exception has occurred". The initial trapping mode is processor-dependent, whereas IEEE requires the default behavior to proceed without a trap. The report does not define how to specify a trap handler under user control.

5 - C / C++

The programming language C [Kernighan & Ritchie 1988] and its successor C++ [Stroustrup 1991] have always been used for a wide variety of applications. More and more scientific libraries are being developed in C++ because of its object-oriented features. In the following paragraphs we compare the IEEE standard with the floating-point functionality defined in the international standard for the programming language C [ISO/IEC 1990]. Since C++ is completely based on the language C for its floating-point part, this comparison also holds for the C++ standard in process of formation [ANSI/ISO 1995].

5.1 - Formats

The ISO C standard [ISO/IEC 1990] states: "There are three floating-types, designated as float, double, and long double. The set of values of the type float is a subset of the set of values of the type double, the set of values of the type double is a subset of the set of values of the type long double". There is no guarantee that these types correspond to IEEE single, double or extended formats. A header file <float.h> is required in which the characteristics of floating-point types are defined in terms of a number of parameters.

5.2 - Rounding

Rounding is not discussed, except for the parameter FLT_ROUNDS in <float.h> which characterizes the rounding mode for floating-point addition only.

5.3 - Operations

It is possible to add, subtract, multiply and divide. The remainder operation '%' is only applicable to integer types and whether, for example, $-23/4$ returns -5 or -6, is implementation-dependent. The header file <math.h> declares mathematical routines which take double arguments and return double values. Float and long double arguments to these functions are automatically converted to double representations, and a double result is always converted when assigned to a float or long double variable. An IEEE remainder function is not available: "The fmod function returns the value $x - i \times y$, for some

integer i such that, if y is nonzero, the result has the same sign as x and the magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the `fmod` function returns zero is implementation-defined". The `sqrt` function computes the nonnegative square root, but a domain error occurs if the argument is negative, whereas IEEE requires an invalid exception. The C standard also gives future library directions. The names of all existing functions in the `<math.h>` header file, suffixed with f or l, are reserved for corresponding functions with float and long double arguments and return values. In addition to the double versions of the math functions in C, the C++ draft explicitly adds the float and long double overloaded versions of these functions. All conversions between internal formats which cannot be represented exactly, deliver either the nearest higher or nearest lower value, chosen in an implementation-defined manner. Hence, they are not necessarily correctly rounded. The header file `<math.h>` contains two functions to round double to integral double values:

- `double ceil (double x)` rounds `x` to the smallest integral value not less than `x`
- `double floor (double x)` rounds `x` to the largest integral value not greater than `x`

The `strtod` function in the header file `<stdlib.h>` is responsible for all decimal to binary conversions. If its string argument has the expected form, `strtod` converts it to double. All six required comparisons are available, only the unordered predicate is absent.

5.4 – Infinity, NaNs, and Signed Zero

Special representations are not discussed.

5.5 – Exceptions

The C standard clearly states that: "If an exception occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined".

5.6 – Traps

On the occurrence of signals or interrupts, it is not guaranteed that the functions in the standard library (including `<math.h>`) are reentrant. The header file `<signal.h>` contains functions to install a user-specified routine on the occurrence of so-called signals such as floating-point exceptions.

6 – Numerical C Extensions Group (NCEG)

This section discusses the final proposal of NCEG [Thomas 1996]. The proposal is intended to integrate floating-point C extensions into the current revision of the C standard. Although the detailed proposal contains different specifications for IEEE and non-IEEE platforms, the discussion will be restricted to IEEE implementations.

6.1 – Formats

The C types float and double correspond to IEEE single and double formats, but the long double type matches an IEEE extended format, else a non-IEEE extended format, else the IEEE double format. Any non-IEEE extended format has more precision than double precision and at least the range of IEEE double format. The recommended practice is to match an IEEE extended format.

6.2 – Rounding

The header file `<fenv.h>` declares the necessary types and functions to provide access to the floating-point environment, including control of the IEEE rounding directions.

6.3 – Operations

All basic IEEE operations are available for all formats. The non-IEEE `fmod` function is still available for compatibility reasons. The `rem` function in `<math.h>` provides the IEEE remainder operation. Floating-point format conversions are performed according to the IEEE rules. The following two functions, defined in `<math.h>`, convert floating-point arguments to integers (only for the type long int), and to integral floating-point values, using the current rounding direction:

- `long int rinttol(double x)` rounds `x` to the nearest long integer
- `double rint(double x)` rounds `x` to an integral double floating-point value

For each mathematical function defined in `<math.h>`, the corresponding functions with float and long double arguments and return values are also provided. The proposal defines a constant `DECIMAL_DIG` in `<math.h>`, so that all conversions between any supported IEEE format and decimal numbers with `DECIMAL_DIG` or fewer significant digits are correctly rounded. All comparisons are available, including an unordered predicate.

6.4 – Infinity, NaNs, and Signed Zero

Infinities, NaNs, and signed zeros are explicitly supported. Decimal to binary conversions and vice versa support infinities, NaNs and signed zeros in accordance with the IEEE standard. But a proposal that the conversion function `strtod` be allowed to return a NaN for invalid numeric input, was withdrawn because of the incompatibility with the C standard, which demands that this function returns 0 for invalid numeric input.

6.5 – Exceptions

The header file <`fenv.h`> defines a type `fenv_t` representing the entire floating-point environment. This environment includes exception status flags, dynamic rounding modes and: "(...) other similar floating-point state information". Although it is possible to save and restore this whole floating-point environment as one entity, its exact content is implementation-defined. The status of floating-point exceptions can be saved and cleared individually and collectively. Individual functions to change the trapping or non-trapping behavior of floating-point exceptions are not mentioned explicitly (see 6.6).

6.6 – Traps

Trap handlers are outside the scope of the NCEG document.

7 – On the IEEE conformance of some popular compilers

So far, it has been shown that the programming languages Fortran and C are converging slowly towards more compatibility with the IEEE standard for floating-point arithmetic. Even so, the current standards of these languages do not address the need for predictable floating-point arithmetic. The next paragraph discusses tools which were used to test the IEEE conformance of some popular Fortran 90 and C / C++ compilers. Subsequent paragraphs present a summary of the test results for each compiler. A full report, with an appendix discussing these test results in more detail, can be obtained at URL ftp://hwins.uia.ac.be/pub/cant/SIGPLAN/SIGPLANreport.ps. This appendix also describes compiler-specific extensions for setting the rounding direction at runtime, detecting floating-point exceptions and changing the trapping or non-trapping behavior of exceptions. The test programs which were used to screen the compilers in this section can be found at URL ftp://hwins.uia.ac.be/pub/cant/SIGPLAN/test-programs/.

7.1 – Verification of floating-point arithmetic

It is clear that the number of elements in a chosen floating-point set makes it impossible to test all combinations of operations and floating-point arguments. Examples such as the famous Intel Pentium bug illustrate that even well-known hardware implementations of the IEEE standard are not necessarily error free. The compilers in the subsequent sections were screened using test sets with carefully chosen floating-point arguments, covering all the aspects (including rounding directions and exceptions) of all IEEE operations except conversions. For the latter, the test sets only screen decimal to binary conversions together with a limited number of special cases for the other conversions.
The test data for the operations $+$, $-$, $\times$, $/$ and square root are part of UCBTEST [Hough 1988]. These were extended with test sets for the IEEE remainder operation and comparisons. For decimal to binary conversions, the test data include, besides some special cases, a list of decimal numbers which are difficult to convert to the IEEE double format, as compiled by [Tydeman 1996]. It should be noted that not all decimal numbers in the test set belong to the IEEE range within which decimal to binary conversions are correctly rounded. For example, a special case is the decimal representation of the expression $1 + 2^{-1} + 2^{-24} + 2^{-52}$, which equals $1500000059604644997435229925031308084726333618164062 5 * 10^{-52}$. This number is exactly representable in double precision, yet does not belong to the IEEE range for correct decimal to binary conversions. Another decimal number $1500000059604644886412927462515654042363166809082031 25 * 10^{-53}$ corresponds to the expression $1 + 2^{-1} + 2^{-24} + 2^{-53}$, but can no longer be represented exactly in double precision. It does not need to be correctly rounded to its binary representation according to IEEE.
All test sets, except conversions, were verified against a straightforward implementation of IEEE double precision arithmetic, initially developed by [Snelgrove 1989]. This implementation was extended to include IEEE rounding and the square root, remainder and comparisons. The test vectors which were used do not guarantee to discover all errors in a floating-point implementation. Note that all compilers target IEEE-conforming hardware (SUN Sparc stations or Intel PCs). It has never been the intention to test these hardware platforms.
As will become clear from the test results, compilers can hide features from the arithmetic at hand or even introduce errors or anomalies which are not present in the underlying hardware. The tests focus on this artificial layer (figure 1) formed by compilers which translate the end-user's program to executable machine instructions running on the underlying hardware. All tests were performed in double precision and in non-trapping mode.

7.2 – Fortran 90

None of the tested Fortran 90 compilers offer the IEEE remainder operation or the unordered comparison, because they are not defined in the Fortran 90 standard.
Edinburgh Portable Compilers Ltd. (EPC) distributes a Fortran 90 compiler for SUN Sparc. Various incompatibilities with the IEEE standard were found. Arithmetic operations do not signal floating-point exceptions correctly. Decimal to binary

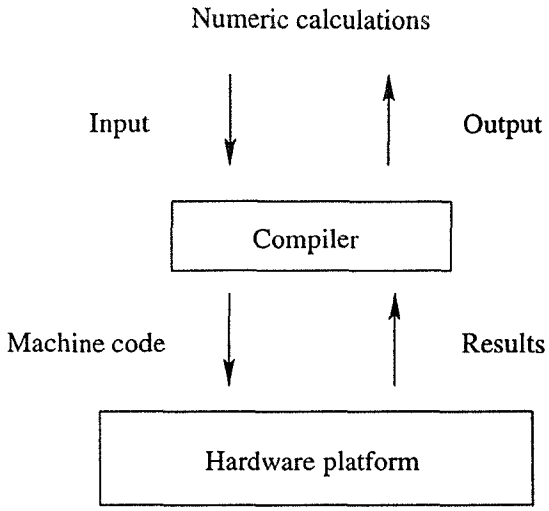

Figure 1: Compilers determine the view on the underlying hardware

and binary to decimal conversions are not always correctly rounded within the range required by IEEE, and are incorrect outside this range. Special representations are not converted correctly. Both decimal to binary and binary to decimal conversions signal the inexact exception inappropriately. The comparisons $>$, $\geq$, $<$ and $\leq$ return true instead of false on NaNs. Comparisons often raise the invalid exception without reason.
The SUN Microsystems Inc. f90 compiler for Sparc, version 1.1 (1995) generates code which executes all arithmetic operations correctly, except for the comparisons $>$, $\geq$, $<$ and $\leq$ which return true instead of false on NaNs. Decimal to binary and binary to decimal conversions outside the correctly rounded IEEE range are incorrect. Binary to decimal conversions do not preserve the sign of -0. Both decimal to binary and binary to decimal conversions signal the inexact exception inappropriately. Another remark is that the compiler option `-fnonstop`, which masks all exceptions so that the execution of generated programs is not halted on floating-point exceptions, is not the default (as IEEE requires).
A list of other compilers, LF90 version 2.00 (1996) for Intel PCs from Lahey Computer Systems Inc., FortranPlus version 1.3 (1996) for SUN Sparc and version 1.3.02 (1996) for Intel PCs from N.A. Software Ltd. and two compilers for SUN Sparc from The Numerical Algorithms Group Ltd. (NAG), NAGWare f90 version 2.2 (1996) and NAG/ACE Fortran 90 release 1.0 (1996), were not included in our tests because they do not provide Fortran 90 extensions to detect floating-point exceptions or change the rounding direction at runtime. Most of these compilers allow to call external C routines to achieve the same functionality, but it would be difficult in these cases to pin-point the source of erroneous results.

7.3 - C / C++

The C / C++ compilers which we tested on SUN Sparc stations deliver almost fully correct results compared with the above Fortran 90 compilers.
SUN Microsystems Inc. provides a proprietary C++ compiler for SUN Sparc stations. We tested version 4.0.1 (1994). All arithmetic operations and conversions execute correctly. Only the unordered comparison does not signal the invalid exception for signaling NaNs.
The Free Software Foundation Inc., a non-profit organization, also develops a C++ compiler for various platforms. We tested GNU C++ version 2.7.2 (1996) for SUN Sparc stations. Their compiler produces exactly the same results as the above SUN compiler, except for one additional error: binary to decimal conversions often raise the inexact exception flag for some unknown reason.
DJGPP version 2.0 (1996) from Delorie Software contains a C / C++ compiler for Intel PC platforms based on the above GNU C++ compiler. Using this compiler, multiplication and division do not always signal the underflow flag. The remainder operation does not signal the invalid exception when the first operand is a NaN and the second operand equals zero, which is an invalid operation. Decimal to binary conversions and vice versa are not correct and signal the inexact exception flag inappropriately.
For an evaluation of Borland C++ version 4.0 (1993) and Microsoft Visual C++ version 1.52 (1993) for Windows platforms, we refer to the appendix.

8 – Conclusion

The current standards of the programming languages Fortran 90 and C still do not fully support IEEE floating-point arithmetic and pay little attention to the need for predictable floating-point arithmetic. These standards, for instance, do not define the necessary functions to establish the rounding direction or detect floating-point exceptions at runtime. As a

consequence, a number of Fortran 90 compilers do not provide this functionality and when compiler-specific extensions are available, they are seldom portable. It is clear that the use of such extensions and the lack of attention for predictable floating-point arithmetic inhibit the production of fully portable numerical libraries and programs.
The screening of some popular Fortran 90 and C / C++ compilers has shown that many of them do not conform to the IEEE standard. Results of operations and the occurrence of floating-point exceptions are not always reported correctly, although the underlying hardware is fully IEEE-conforming. The lack of conformance of these compilers can be attributed in part to their respective programming language standards. Today, the Fortran and C / C++ standards are subject to important changes, thus converging towards greater compatibility with the IEEE standard for floating-point arithmetic.

9 – Acknowledgments

For the construction of the test sets, we received valuable help and comments from several people. We thank David Hough for his feedback on UCBTEST and Jerome Coonen for his support with the IEEE test suite. Thanks also to Tom Aelbrecht and Sven Rubben who did partial implementations of the IEEE floating point simulation, and in particular to Steve Snelgrove who initially developed the program and kindly provided us with his source code. Finally, we are grateful to the following companies for allowing us to use their compiler products for testing purposes: Computing & Systems Consultants bv, Edinburgh Portable Compilers Ltd., Lahey Computer Systems Inc., N.A. Software Ltd., SUN Microsystems Inc. and The Numerical Algorithms Group Ltd. The first author is supported by a grant from the Flemish institute for the promotion of scientific technological research in industry - Flanders (Belgium) (IWT). The second and third author are respectively Research Director and Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)(F.W.O.).

References

[ANSI/ISO 1995] American National Standards Institute (ANSI). *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, 28 April 1995. ANSI doc. no. X3J16/95-0087, ISO/IEC doc. no. WG21/N0687.

[Gay 1990] David M. Gay. Correctly Rounded Binary-Decimal and Decimal-Binary Conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, 30 November 1990.

[Hough 1988] David G. Hough et al. UCBTEST, a suite of programs for testing certain difficult cases of IEEE 754 floating-point arithmetic. Restricted public domain software from http://netlib.bell-labs.com/netlib/fp/index.html.

[IEEE 1985] IEEE Task P754. *ANSI/IEEE Std 754-1985, Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 12 August 1985. Reprinted in ACM SIGPLAN Notices 22(2):9-25, June 1987. Also known as IEC 559:1989, Binary floating-point arithmetic for microprocessor systems.

[IEEE 1987] IEEE Task P854. *ANSI/IEEE Std 854-1987 Standard for Radix-independent Floating-Point Arithmetic.* IEEE, New York, 5 October 1987.

[ISO/IEC 1990] ISO/IEC. *Programming languages – C*, 15 December 1990. Reference Number: ISO/IEC 9899:1990 (E).

[ISO/IEC 1991] ISO/IEC. *Information technology – Programming – Fortran*, 1 July 1991. Reference Number: ISO/IEC 1539:1991 (E).

[ISO/IEC 1996] ISO/IEC JTC1/SC22/WG5. Information technology – Programming – Fortran. Technical Report for Floating-Point Exception Handling N1225 (draft), ISO/IEC, 15 August 1996.

[ISO/IEC 1997] ISO/IEC JTC1/SC22/WG5. Portable Floating-Point Exception Handling. Technical Report (22.02.01.02), ISO/IEC, 1997. Announced for 1997.

[Kernighan & Ritchie 1988] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, USA, second edition, 1988.

[Metcalf & Reid 1996] Michael Metcalf and John Reid. *Fortran 90/95 Explained.* Oxford University Press, Oxford, UK, 1996.

[Reid 1996] John Reid. Draft Technical Report for Floating-Point Exception Handling. ISO/IEC JTC1/SC22/WG5 N1195, ISO/IEC, 2 July 1996.

[Snelgrove 1989] Steve Snelgrove. IEEE floating-point routines. Private Communication, 1989.

[Stroustrup 1991] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, second edition, 1991. Copyright AT&T Bell Telephone Laboratories Inc., Murray Hill, New Jersey, USA.

[Thomas 1996] Jim Thomas. C9X Floating-Point Proposal. Technical report, ANSI/ISO, 26 February 1996. ANSI doc. no. X3J11/96-010, ISO doc. no. WG14/N546.

[Tydeman 1996] Fred Tydeman. Decimal to binary conversion for IEEE 754/854 double format. Posting to the reliable_computing@interval.usl.edu mailing list on 25 February 1996. E-mail: tydeman@tybor.com.