A Precision- and Range-Independent Tool for Testing Floating-Point Arithmetic II: Conversions

BRIGITTE VERDONK, ANNIE CUYT, and DENNIS VERSCHAEREN
University of Antwerp

The IEEE 754 and 854 standards for floating-point arithmetic are essentially a specification of a programming environment, encompassing aspects from computer hardware, operating systems, and compilers to programming languages (see especially Section 8). Parts I and II of this paper together describe a tool to test floating-point implementations of arbitrary precision and exponent range (hardware as well as software) for compliance with the principles outlined in the IEEE standards. The tool consists of a driver program, together with a very large set of test vectors encoded in a precision-independent syntax. In Part I we have covered the testing of the basic operations $+$, $-$, $\times$, $/$ and of the square root and remainder functions. In Part II we describe the extension of the test tool to deal with conversions between floating-point formats, conversions between floating-point and integer formats, the rounding of floating-point numbers to integral values, and binary-decimal conversions. Conversions can now be tested from a floating-point format of arbitrary precision and exponent range to another arbitrary smaller (larger) floating-point format as well as to and from fixed hardware integer formats. Conversions between the bases 2 and 10 can be tested for a number of precisions ranging from single (24 bits), double (53 bits), long double or extended (64 bits) to quadruple (113 bits) precision and a proper multiprecision (240 bits) format. We conclude Part II with some applications of our test tool and report on the results of testing various floating-point implementations, meaning various language-compiler-hardware combinations as well as multiprecision libraries.

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*

General Terms: Verification

Additional Key Words and Phrases: Multiprecision, floating-point, arithmetic, IEEE floating-point standard, validation, conversion, decimal

Authors' address: Department of Mathematics and Computer Science, University of Antwerp, Universiteitsplein 1, Antwerp B-2610, Belgium; email: verdonk@uia.ua.ac.be.

1. INTRODUCTION

In Part I of this report we introduced a comprehensive precision- and range-independent tool to test how well a floating-point implementation with arbitrary precision and exponent range complies with the philosophy of the IEEE-754 [IEEE 1985] and IEEE-854 [IEEE 1987] standards. As pointed out in greater detail in Verdonk et al. [2001], the motivation for the development of such a tool is the ever increasing need in many applications for more precision than provided by the IEEE single and double formats. While implementations of floating-point arithmetic with larger precision become extensively available (be it in hardware or software), the tools for testing them systematically remain almost nonexistent.

In effect, the IEEE-754 and IEEE-854 standards are specifications of programming environments. These standards list, besides requirements for floating-point formats and specifications for rounding, also specifications for

(a) add, subtract, multiply, divide, square root, remainder, and compare operations,

(b) conversions between different floating-point formats,

(c) conversions between integer and floating-point formats,

(d) rounding of floating-point numbers to integral value,

(e) conversions between basic format floating-point numbers and decimal strings, and

(f) floating-point exceptions and their handling, including nonnumbers (NaNs).

IEEE-754 requires that each operation in (a) and each conversion in (b) through (d) shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the following four rounding modes: round to nearest, round to zero, round up, or round down. If the floating-point result is computed in this way, we shall call it exactly rounded.

The precision-independent tool we have developed is designed to test a floating-point system in its globality, in other words all operations (a), all conversions (b) through (e), as well as the handling of all floating-point exceptions (f). In Part I of the paper, we only presented those aspects of the test tool that check the operations add, subtract, multiply, divide, square root, and remainder, including the floating-point exceptions raised by those operations. In this part we describe the testing of all conversions listed in (b) through (e), including the involved exceptions.

The rest of the paper is structured as follows. In Section 2, we extend the precision-independent syntax proposed in Part I to accommodate the testing of floating-point conversions. Sections 3 through 6 discuss each conversion operator in detail and explain which aspects of the conversion are tested by the test vectors. In Section 7, we discuss the driver program of our test tool. In Section 8 we comment on programming language issues which pop up when testing conversions on hardware platforms. The final section contains the results of applying our tool to test conversions on some popular computing platforms as well as in multiprecision implementations.

2. A PRECISION- AND RANGE-INDEPENDENT SYNTAX FOR TEST VECTORS

In this section we extend the precision-independent syntax for test vectors introduced in Part I, to deal with the testing of conversions. To avoid duplication, we refer the reader to Section 3.1 of Part I for the introductory notation. As described in Part I, a test vector consists of at most 9 fields: version number and operator, (optional) precision specification, rounding mode, first operand, second operand (which in the case of conversions is always zero or void), exceptions, result, and (optional) comment. For more information on these fields we refer to Verdonk et al. [2001]. Here we only describe the extension of the syntax that was necessary to test conversions.

2.1 The Extended Syntax

The following operators have been added to refer to the different conversions:

—c: copying from a smaller to a larger floating-point format

—r: rounding from a larger to a smaller floating-point format

—i: rounding of a floating-point value to an integral floating-point value

—ri, ru: rounding of a floating-point value to signed, respectively unsigned, 32-bit integer formats

—rI, rU: rounding of a floating-point value to signed, respectively unsigned, 64-bit integer formats

—ci, cu: conversion from a 32-bit signed, respectively unsigned, hardware integer to a floating-point number

—cI, cU: conversion from a 64-bit signed, respectively unsigned, hardware integer to a floating-point number

—d2b: decimal-to-binary conversion

—b2d: binary-to-decimal conversion

Test vectors for the operators ‘c’ and ‘r’ are discussed in Section 3, for the operator ‘i’ in Section 4, for the conversion between floating-point and integer formats (the operators ‘ri’, ‘ru’, ‘rI’, ‘rU’, ‘ci’, ‘cu’, ‘cI’, and ‘cU’) in Section 5 and for decimal-to-binary conversion in Section 6.

The precision independence of our test tool stems from the fact that in each test vector the floating-point operands and result are encoded using a format-independent syntax. For a specified precision and exponent range (which are command-line parameters for the test driver program), the actual test vector (operands, rounding mode, and result) is generated at runtime from the format-independent encoding. The syntax of this encoding is extensively described in Part I and summarized in Appendix A in BNF form. We review some of its features here by means of simple examples.

The encoding of a floating-point number consists of an optional sign, a mandatory root number and zero or more modifier suffixes. Consider for example the elements $f_1 = 2^{L-1} = 2^{-B}$ and $f_2 = 1 + 2^{-t+2}$ of the floating-point set $\mathbb{F}(2, t, L, U)$ with bias $B = U = -L + 1$. The encodings of these numbers in our precision-independent syntax are `1mB` and `1i(t-2)1`. In both cases, the root number is 1. The modifier in f_1 is `mB`, where `mB` stands for "exponent minus `B`," meaning that the root value is multiplied with 2^{-B}. Similarly, the modifier `pB` stands for "exponent plus `B`." The modifier in f_2 is `i(t-2)1` to increment the root number by 1 at bit position $t - 2$ (where the bit position varies between 0 and $t - 1$).

What needs to be remarked is that some encodings refer to the precision and the bias of the floating-point set under consideration. In test vectors for basic operations, both the operands and the result are elements of the same floating-point set, and the notations `t` and `B` are used to refer to the precision and bias of that set. In test vectors for conversions, often two different floating-point sets are involved. To distinguish between the precision and bias of the source floating-point set and the precision and bias of the destination floating-point set, we use `t` and `B` for the former and `u` and `C` (the next characters in the alphabet) for the latter. This is illustrated in the next test vectors:

```
Ar  =>  1i(u)3mC  0  xv  0i(1)1i(u-1)1
Ar  0<  1i(u)3mC  0  xu  0i(1)1
```

These vectors test rounding between two floating-point sets $\mathbb{F}(2, t, L_1, U_1)$ and $\mathbb{F}(2, u, L_2, U_2) \subseteq \mathbb{F}(2, t, L_1, U_1)$ with respective biases $B = -L_1 + 1$ and $C = -L_2 + 1$. The argument of the rounding is the floating-point number

$$(1 + 2^{-u+1} + 2^{-u}) \times 2^{L_2-1}.$$

The results of the conversion are the respective denormal numbers

$$(2^{-1} + 2^{-u+1}) \times 2^{L_2}$$

and

$$2^{-1} \times 2^{L_2},$$

depending on the rounding mode. This conversion should raise the inexact (`x`) and underflow exceptions. Following Coonen [1984], we use one of three characters to denote the underflow exception in a test vector (`u`, `v`, and `w`), corresponding to the three different definitions of underflow permitted by the IEEE standard. For a detailed description of these underflow detection mechanisms, we refer the reader to Cuyt et al. [2000b] and the references therein.

Our format-independent syntax only applies to operands and results that belong to floating-point sets. For operands or results that belong to the different sets of hardware integers, the integer operand/result in the test vector is represented in hexadecimal notation. For example, the next vector tests the conversion of an unsigned 32-bit integer to floating-point format:

```
3cu   ALL   0x00010001   0   OK   1i(16)1p16   = 65537
```

Note that this conversion is exact (no inexact exception raised) only if the precision of the destination floating-point format is at least 17, and the exponent range at least $[-15, 16]$. These conditions are not explicit in the test vector, because throughout the test tool it is assumed that the considered floating-point formats for operands and result are such that

$$24 \leq t \leq U + 1 = B + 1$$

$$U = -L + 1 = 2^{k-1} - 1 \qquad k \geq 8.$$

Subject to the above, the vectors in the test tool can be used to verify operations and conversions in any floating-point set $\mathbb{F}(2, t, L, U)$. The implicit requirement that $\beta = 2$ is not a restriction on their applicability. Indeed, in most multiprecision software packages the base β can be specified by the user, within certain bounds.

The encoding of operands and results in test vectors for decimal $\leftrightarrow$ binary conversion logically differs from the encodings described previously. We will discuss this and the corresponding test vectors extensively in Section 6.

2.2 The Complete Test Set

For conversions no test sets are available in Hough et al. [1988] while the vectors in Coonen [1984] refer specifically only to the hardware single, double, and quadruple formats (except for the rounding of floating-point numbers to integral values). Hence, for the conversion test set, the first job was to investigate which of these test vectors could be generalized for arbitrary floating-point sets. To this format-independent generalization, which consisted of about 850 vectors, we then added approximately 650 precision-independent vectors for the conversion operators ‘r’, ‘i’, ‘ri’, ‘ru’, ‘rI’, and ‘rU’, and 14,500 precision dependent vectors to test decimal $\leftrightarrow$ binary conversion (see Section 6 and Paxson and Kahan [1991]). The vectors are centered around the testing of

—the appropriate handling of special representations (signed zeroes, NaNs, etc.);

—the appropriate detection of exceptions such as overflow, underflow, and invalid (where relevant for the operation);

Table I. Floating-Point Format Conversion: Rounding Involving Underflow

	Operand	$\rho_{roundto\ x}$	$\sigma_{roundto\ x}$
Case 1	$(\pm, \sum_{i=0}^{t-1} x_i 2^{-i}, -C-k)$	x_{u-1-k}	$\sum_{i=u-k}^{t-1} x_i \neq 0$

—exact rounding and the corresponding detection of the inexact exception.

The complete set of test vectors is divided into files, one per operation or conversion to be tested. To actually test a floating-point implementation with precision t and exponent range $[L, U]$, one calls the driver program (called IeeeCC754), specifying the parameters of the floating-point format (the precision t, the number of bits k to store the exponent, and whether the leading bit is hidden) and a file of test vectors. For each vector in the file, the driver program translates the precision-independent test operands and result into the specified floating-point format. For the given operands, the implementation being tested then computes the result of the operation or conversion. The result and exceptions are compared with those in the test vector. The output of the driver program is a log-file listing any errors that have occurred.

For the complete functionality of the driver program and its many options, we refer to Section 7. In the following sections we first discuss the content of the test set for each conversion in more detail.

3. CONVERSIONS BETWEEN FLOATING-POINT FORMATS

The IEEE standards require that conversions between all supported floating-point formats be possible. If the conversion is to a narrower precision, the result shall be rounded exactly, subject to the current rounding mode. Conversion to a wider precision is exact.

We systematically tested all round and sticky bit combinations, including near halfway cases and almost exactly representable floating-point results. Such tests also serve to check the appropriate signaling of the inexact exception.

In Coonen [1984] some vectors are included where the conversion from double to single, from quadruple to single, and from quadruple to double induces overflow, but no cases to test the proper signaling of the underflow exception were present. To make the overflow cases precision-independent and to construct underflow cases, the literals C and u, which respectively refer to the bias and the precision of the destination floating-point set, are essential. The following example vectors from Table I with operator 'r' illustrate this:

```
Ar   =>  1pCp1        0   xo   H
Ar   <0  1pCp1        0   xo   Hd1
Ar   <0  1mCi(u+1)7   0   xu   0i(1)1
Ar   =>  1mCi(u+1)7   0   xv   0i(1)1i(u-1)1
```

In the first two vectors, the operand of the conversion is 2^{C+1}. When converted to $\mathbb{F}(2, u, -C + 1, C)$, the result is positive infinity in round up and round to nearest (denoted by the literal H for "huge") and the largest representable number in $\mathbb{F}(2, u, -C + 1, C)$ in the other rounding modes ("huge decrement 1"). In both cases, the inexact and overflow exceptions should be signaled. The last two vectors are underflow cases. In round down and round to zero, underflow is due to denormalization loss (called u-underflow in Coonen [1984] and Cuyt et al. [2000b]), while in round to nearest and round up underflow is due to tininess after rounding and inexactness, but without denormalization loss (called v-underflow in Coonen [1984] and Cuyt et al. [2000b]).

These vectors and the other vectors with operator 'r' in the test set are designed to test conversion from an arbitrary floating-point set $\mathbb{F}(2, t, -B + 1, B)$ to a smaller destination format $\mathbb{F}(2, u, -C + 1, C)$, i.e., for $t \geq u$ and $B \geq C$. Note, however, that in the last two test vectors given above, the floating-point operand

$$(1 + 2^{-u+1}+2^{-u}+2^{-u-1}) \times 2^{-C}$$

is an element of $\mathbb{F}(2, t, -B + 1, B)$ only when $t \geq u + 2$ and $B > C$ or $t \geq u + 3$ and $B = C$. For other test vectors, other conditions may apply. Rather than making these conditions explicit in the test vectors, they are checked by the driver program during translation of the operand to its binary representation. Test vectors where the operand is not an element of the source floating-point set for the given values of t, u, B, and C are skipped. The ignored vectors are logged for reference, and the tester is informed of the percentage of vectors that could not be executed.

4. ROUNDING FLOATING-POINT NUMBERS TO INTEGRAL VALUES

It is clear that only the inexact exception can be raised by the rounding of a floating-point number to its integral value, at least if we assume, as we do in our test tool, that the precision t and bias B of the floating-point set satisfy the natural condition $t - 1 \leq B$ (thus avoiding overflow).

With only exact rounding to test, we limited ourselves to creating only a few extra vectors with conversion operator `i`, including exact halfway cases, and incorporated these with the corresponding precision-independent encodings of vectors in Coonen [1984].

5. CONVERSION BETWEEN FLOATING-POINT AND INTEGER FORMATS

Because the IEEE standards are not specific about the integer formats, it seemed most obvious to restrict ourselves to two most frequently available hardware formats: 32 bit and 64 bit. This is also the choice made in Coonen [1984]. In line with most current hardware platforms, our test tool assumes 2's complement representation of signed integers, for which the range of representable numbers is $[-2^{31}, 2^{31}-1]$, respectively $[-2^{63}, 2^{63}-1]$.

Vectors to test conversions between floating-point and integer formats are, more often than is the case for the other conversions, precision dependent. Indeed, copying from unsigned 32-bit integers to any floating-point format with precision $t \geq 32$ is always exact, while this is clearly not necessarily the case when $t < 32$. This precision dependence is handled by the precision specification field in the test vectors. When the precision specification field is blank, as we have seen so far, the test vector is applicable to arbitrary floating-point formats. When not blank, it specifies the precision to which the test vector applies. The next vectors illustrate this. They test the conversion from unsigned 32-bit integers to IEEE single (precision specification 's'), respectively IEEE double (precision specification 'd'):

```
3cu  s  <0=  0x01000001  0  x   1p24
3cu  s   >   0x01000001  0  x   1i(23)1p24
3cu  d  ALL  0x01000001  0  OK  1i(24)1p24
```

While the latter is in fact applicable to all floating-point formats with precision $t \geq 32$, our test tool does not allow ranges of formats to be specified in the precision specification field. When a test vector is applicable only in a given range of floating-point formats, it is included in the test set for one or more specific formats within that range. The following floating-point precision specifications can be encountered in the context of testing conversions:

—s: single precision and exponent range ($t = 24$ and $[L, U] = [-126, 127]$)

—d: double precision and exponent range ($t = 53$ and $[L, U] = [-1022, 1023]$)

—l: long double precision and exponent range ($t = 64$ and $[L, U] = [-16382, 16383]$)

—q: quadruple precision and exponent range ($t = 113$ and $[L, U] = [-16382, 16383]$)

—m: multiprecision ($t = 240$ and $[L, U] = [-16382, 16383]$)

According to IEEE [1985; 1987], the conversion between floating-point and integer formats shall, like all other operations and conversions, be dependent on the rounding mode. As we shall see in Section 8, up until the recently published C standard [ANSI-ISO-IEC 1999], these specifications were not compatible with the specifications for floating-point ↔ hardware integer conversion in C/C++, nor are they fully compatible with the Fortran 95 standard [ANSI-ISO-IEC 1997]. Hence, IEEE-compliant conversions from floating-point to integer formats in all rounding modes are still often unavailable at the programming language level. It should be observed that, probably for this reason, the vectors in Coonen [1984] which test the conversion of floating-point numbers to integer formats identify all rounding modes with round to zero. The issue of supporting the conversion of floating-point numbers to integer values in all rounding modes has recently come under discussion by the IEEE 754R Revision Group.[1]

In our set, vectors are included which test exact rounding of all conversions between floating-point and integer formats in all rounding modes:

```
Ari   >=  3i(t-1)1m1   0   x   0x00000002
Ari   0<  3i(t-1)1m1   0   x   0x00000001
```

With respect to exceptions, certainly underflow cannot arise here. It is also easy to see that overflow cannot arise during the conversion of 32- or 64-bit integers to floating-point formats, since one of the assumptions of our test tool is that the floating-point format under consideration is at least as large as IEEE single precision. Hence the exponent range is at least $[-126, 127]$. As for the conversion in the other direction, it may be the case that the result of rounding a floating-point number to integer is too large to be represented in the integer hardware format. In this case, the IEEE standards specify that the invalid exception shall be signaled. To test whether this is done appropriately, we have generalized format-specific vectors included in Coonen [1984] where possible, and added a number of borderline cases. These are cases where the floating-point operand is just within, on the border of, or just outside the range of integers representable in the respective hardware formats:

```
3ri        ALL   1p31         0   i   ?0x7fffffff
Ari   d    ALL   1d(30)1p31   0   OK  0x7fffffff
Ari   l    ALL   1d(30)1p31   0   OK  0x7fffffff
Ari   q    ALL   1d(30)1p31   0   OK  0x7fffffff
Ari   m    ALL   1d(30)1p31   0   OK  0x7fffffff
Ari   s    ALL   1d(23)1p31   0   OK  0x7fffff80
```

The operand 2^{31} in the first test vector is just outside the range of representable 32-bit integers. The result of this conversion is not checked upon by the test driver program (indicated by the ? in the result), as it is left unspecified by the standard. However, it should raise the invalid exception. In the next four test vectors, a precision specification is needed, since the conversion of the operand $2^{31}-1$ only applies for precisions $t \geq 31$, of which 'd', 'l', 'q', and 'm' are instantiations. When $t < 31$, the largest representable signed integer cannot be represented exactly as a floating-point number.

6. DECIMAL-TO-BINARY AND BINARY-TO-DECIMAL CONVERSION

Decimal-to-binary conversions and vice versa stand out from the other conversions in IEEE-754 and IEEE-854 in that these conversions need not be correctly rounded for all ranges of operands. Full details can be found in IEEE [1985; 1987]. However, since the publication of the IEEE floating-point standards, algorithms for correctly rounded conversion between decimal

[1]IEEE 754R Revision Group. Notes from 754R meeting, January 8, 2001. Available at http://www.validlab.com/754R/01-01-08.html.

strings and binary floating-point formats have become available which incur little time penalty in common cases [Gay 1990].

Our test tool has taken this evolution into account. With the option `-ieee`, the driver program considers only those vectors in our conversion test set with operands lying in the range within which the IEEE standards require exact rounding. If the option `-ieee` is omitted, the whole decimal $\leftrightarrow$ binary test set is executed. Rather than testing on the loosened rounding requirements outside the range specified in the IEEE standards, our vectors then test on exact rounding for all operands. When the implementation being tested returns a result which is not exactly rounded, the driver program issues a warning rather than an error in case the operand is outside the IEEE range.

Whereas no test set for decimal $\leftrightarrow$ binary conversion is included in Coonen [1984], tables of decimal numbers are available [Tydeman 1996] which require a lot more than 53 bits to achieve correct rounding to double-precision binary representation. These tables correspond with results obtained in Paxson and Kahan [1991]. There, the authors describe algorithms to find numbers in an input base (2 or 10) which, in the output base, lie extremely close to representable numbers (for directed roundings), or exactly half-way between adjacent representable numbers (for round to nearest).

Using these algorithms, two groups of test vectors were created: one for the operator 'd2b' containing decimal strings of n decimal digits (where n can vary between 1 and 72 depending on the destination precision) and another one for the operator 'b2d' grouping binary floating-point operands of various precisions t. Both groups of test vectors also contain some cases where the decimal $\leftrightarrow$ binary conversion is exact or induces either overflow or underflow.

Some clarifications are needed regarding the syntax of the test vectors for 'd2b':

```
    Ad2b   sieee   UN   +429 E-10   x   +b8410c_10000000001& E-25
```

This example tests the conversion of the decimal number 429×10^{-10} to its binary representation in single precision, obtained by exactly rounding the binary floating-point equivalent of 429×10^{-10}:

$$1.011\ 1000\ 0100\ 0001\ 0000\ 1100\ 10000000001 \ldots \times 2^{-25}.$$

From the pattern following the $24th$ bit, a pattern which is preceded by an underscore in the test vector, it is clear that the binary representation is almost half-way between two single-precision floats, making 429×10^{-10} a decimal number which is relatively hard to round to IEEE single precision. Following the version number `A` and the operator `d2b`, the precision specification is `sieee`. This signifies that the operand is not just single precision but moreover within the range for which IEEE requires exact rounding. Similarly, f`ieee` indicates an operand in floating-point format f, whose value is in the range within which IEEE requires that the conversion of decimal string to binary representation be exactly rounded. Here, f is one of the precision specifications 's', 'd', 'l', 'q', or 'm'. Implementations must only pass the $f$ieee test vectors to comply with the IEEE-754 standard. The field following the precision specification is the rounding mode and takes on the value UN, which indicates that the result of the conversion is not yet rounded in the test vector: the driver program of the test tool will generate a correctly rounded representation for all rounding modes.

The rest of the test vector consists of the decimal string operand, the possible exception flags (here x), and the result, which is no longer encoded in our precision-independent syntax. Rather, the sign is given, then the first t bits of the significand in hexadecimal notation (no bit hidden), then an underscore followed by a sufficient number of bits (in binary notation) in the significand to allow for correct rounding (in this example 11 bits). Finally, the exponent of the result is given in decimal notation. When the precision t is not a multiple of 4, leading zero bits are added to obtain $\lceil t/4 \rceil \times 4$ bits, which are then written in hexadecimal notation. The following test vector for conversion of -9×10^{-47} to double precision further illustrates this:

```
Ad2b   d UN -9 E-47   x  -10711fed5b19a3_11111110& E-153.
```

Test vectors for binary-to-decimal conversion are very similar:

```
Ab2d   sieee   UN   -b5e621 E+44    x  -2_50000000& E+13
Ab2d   s       UN   ebac15 E+108    x  +597_499999999& E+30
```

The underscore in the decimal string is a signal to the test driver to round the result at that point in the string and compare the rounded result with the decimal string of length n generated by the floating-point implementation. Here n is the number of decimal digits preceding the underscore. The & indicates that decimal digits different from the one preceding the & follow in the exact decimal representation.

7. THE CONVERSION TEST DRIVER PROGRAM

For the basics on the test driver program we refer to Part I and to the Web page http://win-www.uia.ac.be/u/cant/ieeecc754.html. Here we only discuss the extensions made to the program to deal with the testing of conversions.

When testing conversions between floating-point formats, both a source and a destination floating-point set must be specified. For the former, the basic options apply, including `-s|-d|-l|-q|-m|{ -e <int> -t <int> [-h] }`. For the latter, the same options but preceded by the character d (destination format) should be used. For example, calling the driver program with the options `-q -ds` implies that the source floating-point format is quadruple and that the destination floating-point format is IEEE single precision.

The options which influence the actual testing phase now also include:

`-ieee`	test decimal $\leftrightarrow$ binary conversion only within range where IEEE requires exact rounding; this option can be abbreviated to `-i`

To illustrate the applicability of our tool, we have applied it on one hand to several hardware floating-point implementations, including the Intel Pentium processors and SUN Sparc workstations, and on the other hand to the multiprecision software library FMLIB [Smith 1991] and our own IEEE-compliant multiprecision floating-point implementation MpIeee [Cuyt et al. 2000a]. We discuss the results of the conversion tests in Section 9. Before we can do so, however, it is necessary to comment on how programming language standards and compilers provide support for (IEEE) conversions.

8. LANGUAGE SUPPORT FOR IEEE FLOATING-POINT CONVERSIONS

While our test tool is particularly suited to test multiprecision floating-point implementations, which are mostly available in software, it can certainly also be applied to floating-point environments on hardware platforms. When testing the latter, support for floating-point conversions in compilers and programming language standards is an important issue which will be the topic of this section.

It is well-known that not all conversions required by IEEE are available in hardware. Hence they are compiler-specific and/or programming-language-specific. Furthermore, those conversion functions which are available in hardware and are IEEE-compliant, may be inaccessible to the programmer due to programming language specifications.

In fact, before dealing with conversions, it should be noted that a similar problem already arises for the remainder function. On most popular computing platforms, an IEEE-compliant remainder function is available in hardware. However, this hardware function is not available to the user through the standard C function `fmod`, because the specifications for `fmod` in ANSI-ISO-IEC [1999] are not compatible with the IEEE-754 specifications for the remainder operation. To overcome this, the recently published C standard [ANSI-ISO-IEC 1999] has added to its requirements the `remainderf`, `remainder`, and `remainderl` functions (for float, double, and long double arguments) to provide an IEEE-compliant remainder function in C/C++. Even before publication of the new C standard, several compiler builders already supplied the above functions. If not supplied, one was forced to call the corresponding assembler routine to directly address the hardware.

In the next few paragraphs, we discuss language support for IEEE floating-point conversions in different compiler/programming language/hardware combinations.

8.1 Fortran and SUN Ultra Sparc-II

In Table II, we consider the programming environment consisting of SUN Ultra Sparc-II hardware equipped with the Forte Developer 6 update 1

Table II. Support for IEEE Conversions in Fortran 2000 and in the Forte 6.1 f95 Compiler on SUN Ultra Sparc-II

	real(4)				real(8)				real(16)			
Fortran SUN Ultra Sparc-II	Language Standard	RM	f95	ASM single	Language Standard	RM	f95	ASM double	Language Standard	RM	f95	ASM long
format	=, real(_, 4)	?		fdtos	=, real(_, 8)	?		fstod	=, real(_, 16)	?		fstoq
conversion	”	ALL	✓	fqtos	”	*ALL*	✓	fqtod	”	*ALL*	✓	fdtoq
to integral value	aint(_, 4)	0	✓		aint(_, 8)	0	✓		aint(_, 16)	0	✓	
	anint(_, 4)	=[1]	✓		anint(_, 8)	=[1]	✓		anint(_, 16)	=[1]	✓	
	ieee_rint	ALL			ieee_rint	ALL			ieee_rint	ALL		
		ALL	*r_rint*			*ALL*	*d_rint*					
to 32-bit integer	nint(_, 4)	=[1]	✓		nint(_, 4)	=[1]	✓		nint(_, 4)	=[1]	✓	
	=, int(_, 4)	0	✓	fstoi	=, int(_, 4)	0	✓	fdtoi	=, int(_, 4)	0	✓	fqtoi
	ceiling	>	✓		ceiling	>	✓		ceiling	>	✓	
	floor	<	✓		floor	<	✓		floor	<	✓	
from 32-bit integer	=, real(_, 4)	?		fitos	=, real(_, 8)	?		fitod	=, real(_, 16)	?		fitoq
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	
to 64-bit integer	nint(_, 8)	=[1]	✓		nint(_, 8)	=[1]	✓		nint(_, 8)	=[1]	✓	
	=, int(_, 8)	0	✓		=, int(_, 8)	0	✓		=, int(_, 8)	0	✓	
	ceiling(_,8)	>	✓		ceiling(_,8)	>	✓		ceiling(_,8)	>	✓	
	floor(_,8)	<	✓		floor(_,8)	<	✓		floor(_,8)	<	✓	
from 64-bit integer	=, real(_, 4)	?			=, real(_, 8)	?			=, real(_, 16)	?		
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	
binary to decimal	print, write	?			print, write	?			print,write	?		
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	
decimal to binary	read	?			read	?			read	?		
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	

[1] Round to nearest except for exact half-way cases which are rounded away from zero.

Fortran 95 compiler (which we shall abbreviate as f95). The f95 compiler is fully compliant with the Fortran 95 standard [ANSI-ISO-IEC 1997], which describes but does not require modules to support IEEE-754 arithmetic. To make our overview more time resilient and to illustrate the drive at SUN to include IEEE-supporting functionality, we shall in the sequel refer to the upcoming Fortran 2000 standard [Committee J3 2000] when discussing support for IEEE conversions in the Fortran programming language.

SUN Sparc is a single/double-based architecture, providing single- and double-precision hardware as well as quadruple-precision floating-point arithmetic in software [Sun Microsystems 1997]. These three formats can be addressed in Fortran 95 by the `real(kind(0.0))`, `real(kind(0.D0))`, and `real(kind(0.Q0))` type declarations respectively. Alternatively, one can use the `real(4)`, `real(8)`, and `real(16)` declarations.

Table II lists, for each conversion required by IEEE and for each floating-point format, four features:

—language support for that conversion as described by the upcoming Fortran 2000 language standard [Committee J3 2000];

—in which rounding modes (RM) the conversion is callable from Fortran according to the above draft standard ('0' for round to zero, '<' round down, '>' round up, '=' round to nearest, 'ALL' all of the above, and '?' processor-dependent rounding);

—support of the above in the SUN Fortran 95 compiler;

—what, if any, the corresponding assembler (ASM) instruction is.

All entries in the table are in Roman font, except when the functionality is specific for the SUN Fortran 95 compiler and different from the specifications in the draft standard. For such entries, a slanted font is used.

As can be seen from Table II, the upcoming Fortran standard is either unspecific about rounding during conversion, or requires language support for the conversion in all rounding modes, as required by the IEEE standards. However, it should be noted that the Fortran specifications for rounding to the nearest integer (the functions `nint(_, _)` and `anint(_, _)`) differ from the IEEE specifications when the argument is an exact half-way case. The latter requires rounding to the nearest even integer while the former requires rounding away from zero.

Fortunately, where the Fortran standards are unspecific about rounding, the conversion in the f95 compiler is executed subject to the rounding mode set by the programmer, with "round to nearest" as the default rounding mode. To set the rounding mode, the upcoming Fortran standard [Committee J3 2000] prescribes the function `ieee_set_rounding_mode`. In the f95 compiler, the rounding mode can be set with the function `ieee_flags`, which can also be used to test the status of and to clear exception flags. Functions to round a single- or double-precision argument to its integral value in the rounding mode specified by the programmer are only available as part of the SUN `sunmath` library. These functions, called `r_rint` and `d_rint`, are the precursor of the function `ieee_rint` in Committee J3 [2000].

8.2 C/C++ and SUN Ultra Sparc-II

In Table III we consider the programming environment consisting of the same SUN Sparc hardware, now in combination with the Forte Developer 6 Update 1 C++ compiler (CC) and the GNU C++ compiler v2.95.1 (g++). For each conversion we list the same features as in Table II, but now for the C/C++ language standards [ANSI-ISO-IEC 1998; 1999]. Here again we use slanted font to indicate compiler-specific features.

Where the recent C standard is specific about rounding, it requires language support for the conversion in all rounding modes, as can be seen from Table III. This too, is a major and welcome addition to the previous C standard and a result of the efforts of the Numerical C Extension Group [Thomas 1995]. It should be specified, however, that the functions `lrint` and `llrint` and their single- and extended-precision equivalents convert to signed integers only. For conversion to unsigned integers, the required round-to-zero mode of C/C++ is the only rounding mode available.

When the C/C++ standard is not specific about rounding, the conversion using SUN CC is again executed subject to the rounding mode specified by the programmer. The same holds for the g++ compiler, except for decimal ↔ binary conversion which is only supported in round to nearest. Changing the rounding mode in g++ can be done with the function `fpsetround`, and in CC also with the function `fesetround`. Both functions have the same functionality, but the name of the latter is compliant with ANSI-ISO-IEC [1999].

A number of conversion functions, namely `r_rint_`, `rintl`, `irint`, `irintf`, and `irintl`, are only available as part of the SUN `sunmath` library and need to be declared as `extern C`. The latter three provide the functionality of the functions `lrint`, `lrintf`, and `lrintl` required by ANSI-ISO-IEC [1999]. Analogous functions to convert a floating-point number to a 64-bit integer in the rounding mode specified by the programmer are not yet supported in SUN CC and g++.

8.3 C/C++ and Intel x87

In Table IV we consider the programming environment consisting of the Intel x87 processor family equipped with the GNU C++ compiler v2.95.2 (g++). Intel is an example of extended-based hardware, meaning that all computations are performed in extended precision. To mimic pure single precision and double precision in the absence of overflow and underflow, one has to set the rounding precision (called precision control) to 24, respectively 53 bits. The single, double, and extended formats can be addressed in C++ by the float, double, and long-double type declarations respectively.

The last column in Table IV shows that on the Intel processor each conversion, except decimal ↔ binary, is available in extended-precision hardware. Furthermore, it can be observed from Table IV that the g++ compiler for Linux already provides all IEEE-compliant conversion functions required by the new C standard. Some of these functions, however, need to be declared as `extern C`.

Table III. Support for IEEE Conversions in C++ and in the GNU g++ and SUN CC Compilers on SUN Ultra Sparc-II

	Float				Double				Long Double			
C++ SUN Ultra Sparc-II	Language Standard	RM	g++ CC	ASM single	Language Standard	RM	g++ CC	ASM double	Language Standard	RM	g++ CC	ASM long
format conversion	=	?		fdtos	=	?		fstod	=	?		fstoq
	”	*ALL*	✓	fqtos	”	*ALL*	✓	fqtod	”	*ALL*	✓	fdtoq
to integral value	ceilf	>	✓		ceil	>	✓		ceill	>	✓	
	floorf	<	✓		floor	<	✓		floorl	<	✓	
	truncf	0			trunc	0			truncl	0		
	rintf	ALL	✓[1]		rint	ALL	✓		rintl	ALL	✓	
to 32-bit integer	=	0	✓	fstoi	=	0	✓	fdtoi	=	0	✓	fqtoi
	lrintf	ALL			lrint	ALL			lrintl	ALL		
		ALL	*irintf*			*ALL*	*irint*			*ALL*	*irintl*	
from 32-bit integer	=	?		fitos	=	?		fitod	=	?		fitoq
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	
to 64-bit integer	=	0	✓		=	0	✓		=	0	✓	
	llrintf	ALL			llrint	ALL			llrintl	ALL		
from 64-bit integer	=	?			=	?			=	?		
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	
binary to decimal	$\ll$	?			$\ll$	?			$\ll$	?		
	”	*ALL*	✓[2]		”	*ALL*	✓[2]		”	*ALL*	✓[2]	
decimal to binary	$\gg$	?			$\gg$	?			$\gg$	?		
	”	*ALL*	✓[2]		”	*ALL*	✓[2]		”	*ALL*	✓[2]	

[1] `r_rint_` instead of `rintf` in g++
[2] *ALL* only in SUN CC, round to nearest in g++

Table IV. Support for IEEE Conversions in C++ and in the GNU g++ Compiler on Intel x87

	Float				Double				Long Double			
C++ Intel x87	Language Standard	RM	g++	ASM single	Language Standard	RM	g++	ASM double	Language Standard	RM	g++	ASM long
format conversion	=	?			=	?			=	?		fld
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	fst
to integral value	ceilf	$>$	✓		ceil	$>$	✓		ceill	$>$	✓	frndint
	floorf	$<$	✓		floor	$<$	✓		floorl	$<$	✓	
	truncf	0	✓		trunc	0	✓		truncl	0	✓	
	rintf	ALL	✓		rint	ALL	✓		rintl	ALL	✓	
to 32-bit integer	=	0	✓		=	0	✓		=	0	✓	fist
	lrintf	ALL	✓		lrint	ALL	✓		lrintl	ALL	✓	
from 32-bit integer	=	?			=	?			=	?		fild
	”	*ALL*	✓		”	*ALL*	✓		”	*ALL*	✓	
binary to decimal	$\ll$	?			$\ll$	?			$\ll$	?		
	”	=	✓		”	=	✓		”	=	✓	
decimal to binary	$\gg$	?			$\gg$	?			$\gg$	?		
	”	=	✓		”	=	✓		”	=	✓	

9. APPLICATIONS

In line with the discussion in the previous section, we have tested three hardware/programming language/compiler programming environments. All log files of these tests are available at `http://win-www.uia.ac.be/u/cant/ieeecc754.html`. We summarize the results below, but do not report here on functionality which is missing in the implementation due to specifications in the programming language standard or due to lack of support in the compiler itself, as these issues have been thoroughly discussed in the previous section. In running the test set for each implementation, we therefore skipped those test files corresponding to conversion operators which are not supported in the floating-point environment being tested, or called `IeeeCC754` with appropriate options to test a particular conversion only in specific rounding modes.

A final remark is in order before listing our test results. To allow a clear interpretation of the test results, we have compiled the driver program without any compiler optimization options, as it turns out that such options may influence the result of computations (in sometimes unexpected ways).

9.1 SUN f95 on SUN Ultra-Sparc II

Our test tool reported (almost only exception signaling) errors in the Forte Developer 6 update 1 Fortran 95 compiler for the following conversions.

—Conversion to 32-bit integer (operator ‘ri’):

In cases where overflow precludes a faithful representation in the integer format, both the invalid and the inexact exception are signaled in round-up and round-to-zero mode for double- and quadruple-precision arguments. The latter exception should not be raised.

—Conversion to 64-bit integer (operator ‘rI’):

This conversion returned several erroneous results in previous versions of the SUN Fortran compiler. In the current version all results are correct, but the test tool reports some errors with respect to exception signaling. For inexact conversions, the inexact exception is not raised in round-to-zero mode. Furthermore, when overflow precludes a faithful representation in the destination format, the invalid exception is not raised (in single and double precision) in round to nearest, while both the inexact and invalid exceptions are raised (in quadruple precision) in round up and down.

—Binary-to-decimal conversion (operator ‘b2d’):

Conversion is supported in all rounding modes, and no errors were reported by the test set except for a negative zero argument in single, double, and quadruple precision for which the decimal string +0E0 is returned rather than -0E0, in all rounding modes.

9.2 SUN CC on SUN Ultra-Sparc II

All conversions return exact results, and the only reported errors are the following.

—Conversion to 64-bit signed and unsigned integer (operators ‘rI’ and ‘rU’): No signaling of the inexact exception in round-to-zero mode, for all precisions.

—Conversion to 32–bit unsigned integer (operator ‘ru’):
This conversion incorrectly raises the inexact exception together with the invalid exception (in single precision and double precision) in round-to-zero mode.

The result of copying integer zero to floating-point in round-down mode differs depending on the source integer format and the destination floating-point format, sometimes returning -0 and sometimes $+0$.

9.3 GNU g++ on SUN Ultra-Sparc II

In this programming environment, besides some exception signaling errors, erroneous results are returned in decimal $\leftrightarrow$ binary conversion.

—Conversion to 64-bit signed and unsigned integer (operators ‘rI’ and ‘rU’): In single and double precision and round to zero, the inexact exception is not signaled (same errors as in the SUN CC compiler). In quadruple precision and round to zero, the behavior is different from the SUN CC compiler. The inexact flag is erroneously set for exact and for invalid conversions. In two cases of inexact conversions, the underflow exception is signaled together with the inexact exception.

—Conversion to 32-bit unsigned integer (operator ‘ru’):
This conversion incorrectly raises the inexact exception together with the invalid exception (in single and double precision) in round-to-zero mode.

—Binary-to-decimal conversion (operator ‘b2d’):
As indicated in Table III, the rounding mode does not affect the conversion from binary to decimal in the g++ compiler. In other words, the only supported rounding mode is the default round to nearest. In that rounding mode, errors were reported for the conversion of quadruple arguments to decimal representation. All errors concern almost exact halfway cases.

—Decimal-to-binary conversion (operator ‘d2b’):
Here, too, conversion is supported in round to nearest mode only. Our tool reported incorrect results in the conversion to single- and quadruple-precision representation for operands both within and outside the range where IEEE requires exact rounding, and exception-signaling errors for quadruple precision. For conversion to double-precision representation, all binary representations are exact, and only a few exception signaling errors were reported.

9.4 GNU g++ on Intel

In this programming environment, erroneous results again only occur in decimal $\leftrightarrow$ binary conversion, which are the only conversions not implemented in hardware on x87 platforms.

—Binary-to-decimal conversion (operator 'b2d'):
Conversion is supported in round to nearest mode only. For all vectors in the test set the correct decimal output is generated, however without raising the appropriate exceptions.

—Decimal-to-binary conversion (operator 'd2b'):
Here, too, conversion is supported in round to nearest mode only. Our tool reported errors in the conversion to single- and double-precision representation. For the latter, some of the erroneous results were reported for operands lying within the range where IEEE requires exact rounding. When converting to single precision, several errors occurred for vectors where the operand is only slightly larger (in magnitude) than the smallest normal single-precision float, while zero is returned instead. All other errors in conversion from decimal to single precision and double precision concern erroneous last bits in the binary representation. The conversion from decimal to extended precision was error free according to our test tool, except for two overflow cases where NaN is returned instead of infinity. Furthermore, exception signaling is neglected for all destination precisions (single, double, and extended).

—Conversion to 32-bit unsigned integer (operator 'ru'):
This conversion does not raise the invalid exception in cases where the result cannot be represented as an unsigned hardware integer.

9.5 Multiprecision Floating-Point Packages

Our test tool was also applied to two software libraries for multiprecision floating-point arithmetic: FMLIB [Smith 1991], a collection of Fortran routines, and MpIeee [Cuyt et al. 2000a], a C++ class library. Each implementation was tested with respect to its support for conversions, in line with the philosophy of the IEEE standards. The log files of these tests are available at `http://win-www.uia.ac.be/u/cant/ieeecc754.html`.

FMLIB V1.1 supports only two rounding modes: (almost) round to zero and (almost) round to nearest (see Section 10 in Part I). Also, neither denormal numbers nor special representations such as ± 0 are supported, and floating-point exceptions differ significantly from the IEEE requirements. Therefore, we applied our test tool with the command-line options `-r nz` to specify the rounding modes for the test, and `-n xiuoz tiny nan inf snz` to ignore incorrect signaling of exceptions and skip test vectors containing special representations (denormals, NaN, infinity, and signed zero). In the next release of FMLIB all four rounding modes will be supported.

Our own MpIeee C++ class library achieves performance comparable to Smith [1991], while at the same time being fully compliant with the IEEE standards for floating-point arithmetic. In particular, all conversions are exactly rounded to their destination format. Table V lists, for both packages, all the conversion routines which are available and relevant with respect to the IEEE specifications.

Table V. Conversions in FMLIB and MpIeee

IEEE 754	FMLIB	MpIeee
format conversion	FMEQU(MA,MB,NA,NB)	=
to single	FMM2SP(MA,X)	=
to double	FMM2DP(MA,X)	=
from single	FMSP2M(X,MA)	=
from double	FMDP2M(X,MA)	=
to integral value	FMINT(MA,MB)	void rint()
	FMNINT(MA,MB)	
to 32 bit integer	FMM2I(MA,IVAL)	long toInt32()
		unsigned long toUint32()
from 32 bit integer	FMI2M(IVAL,MA)	=
		=
to 64 bit integer		long long toInt64()
		unsigned long long toUint64()
from 64 bit integer		=
		=
binary to decimal	FMOUT(MA,LINE,LB)	ToDecimal(char *decimal, unsigned int prec)
decimal to binary	FMINP(LINE,MA,LA,LB)	FromDecimal(char *decimal)

APPENDIX

A. SYNTAX DESCRIPTION OF EXTENDED COONEN VECTORS IN BACKUS-NAUR FORM (BNF)

```
<test-vector>::= <version><operation> <prec> <rounding> <fp>
                 {<fp>} <exceptions> <fp>
<version>::= <digit> | H | A
<operation>::= + | - | * | / | % | S | <conv>
<conv>::= r| c | i | d2b | b2d | <intconv>
<intconv>::= ci | ri | cu | ru | cI | rI | cU | rU
<prec>::= {e | o | s{ieee} | d{ieee} | l{ieee} | q{ieee} | m{ieee}}
<rounding>::= ALL | 0 | < | > | = | 0< | 0> | =< | => | =0> |
              =0< | UN
<exceptions>::= OK | x | xo | xu | xv | xw | i | z
<fp>::= {<sign>}<root>{<suffix>}* | <decimal> | <integer>
<integer>::= {?}0x{<hex>}+
<decimal>::= <sign>{<hex>}+{_{<digit>}+&} E<sign>{<digit>}+
<sign>::= + | -
<root>::= Q | H | T | {<digit>}+
<suffix>::= {p<literal> | m<literal>} {i<spec> | d<spec>}
            {u<digit>}
<spec>::= <digit> | (<pos>) <digit>
<pos>::= <literal>{+<digit>} | <literal>{-<digit>}
<literal>:: = <digit> | t | h | B | B<digit> | u | C
<hex>::= <digit> | a | b | c | d | e | f
<digit>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

ACKNOWLEDGMENTS

For the construction of the test tool we received valuable help from several people. We specifically thank Jerome Coonen. Our test tool is in parts based on and greatly benefited from his FPTEST program [Coonen 1984]. We also thank David Hough for his feedback on the UCBTEST package. Finally, we are indebted to Johan Bogo and Tim Gevers for their support on the development of the test driver.

The first and second authors are respectively Postdoctoral Fellow and Research Director of the Fund for Scientific Research—Flanders (Belgium) (FWO). The third author is supported by a grant from the Flemish Institute for the promotion of Scientific and Technological research in Industry (Belgium) (IWT).

REFERENCES

ANSI-ISO-IEC. 1997. Information technology—Programming languages—Fortran. Ref. No. ISO/IEC 1539-1.

ANSI-ISO-IEC. 1998. Information technology—Programming languages—C++. Ref. No. ANSI/ISO/IEC 14882-1998.

ANSI-ISO-IEC. 1999. Programming languages—C. Ref. No. ANSI/ISO/IEC 9899.

COMMITTEE J3. 2000. Fortran 2000—Draft standard. Ref. no. J3/01-007. http://www.j3-fortran.org.

COONEN, J. 1984. Contributions to a proposed standard for binary floating-point arithmetic. Ph.D. Dissertation. University of California at Berkeley, Berkeley, CA.

CUYT, A. ET AL. 2000a. The Arithmos project. Univ. of Antwerp, Antwerp, Belgium. http://win-www.uia.ac.be/u/cant/arithmos.

CUYT, A., KUTERNA, P., VERDONK, B., AND VERSCHAEREN, D. 2000b. Underflow revisited. Tech. Rep. Univ. of Antwerp, Antwerp, Belgium. Available via http://win-www.uia.ac.be/u/cant/publications.html. Submitted for publication.

GAY, D. 1990. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10. AT&T Bell Laboratories, Inc., Murray Hill, NJ.

HOUGH, D. ET AL. 1988. 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. *ANSI/IEEE standard for binary floating point arithmetic: Standard 754-1985*. IEEE Press, Piscataway, NJ. Reprinted in ACM SIGPLAN Not. 22, 2 (1987), pp. 9–25.

IEEE. 1987. *ANSI/IEEE standard for radix-independent floating-point arithmetic: Standard 854-1987*. IEEE Press, Piscataway, NJ.

PAXSON, V. AND KAHAN, W. 1991. A program for testing IEEE decimal-binary conversion. Tech. Rep. University of California at Berkeley, Berkeley, CA.

SMITH, D. M. 1991. Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw. 17*, 2 (June), 273–283.

SUN MICROSYSTEMS. 1997. The UltraSparc processor. Technology White Paper. Sun Microsystems, Inc., Mountain View, CA. http://www.sun.com.

THOMAS, J. 1995. Chapter 5: Floating-point C extensions. Tech. Rep. X3J11. Numerical C Extensions Group.

TYDEMAN, F. 1996. Decimal to binary conversion for IEEE 754/854 double format. Posting to the reliable_computing@interval.usl.edu mailing list (Feb. 25, 1996). Email: tydeman@tybor.com.

VERDONK, B., CUYT, A., AND VERSCHAEREN, D. 2001. A precision- and range-independent tool for testing floating-point artithmetic I: Basic operations, square root, and remainder. *ACM Trans. Math. Softw. 27*, 1 (Mar.). This issue.

Received: June 1999; revised: December 2000; accepted: February 2001