Numerical Computation Guide

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-2464-10
May 2002, Revision A

Send comments about this document to: `docfeedback@sun.com`

Contents

Figures

Tables

Before You Begin

This manual describes the floating-point environment supported by software and hardware on SPARC™ and x86 platforms running the Solaris™ operating system. Although this manual discusses some general aspects of the SPARC™ and Intel architectures, it is primarily a reference manual designed to accompany Sun™ language products.

Certain aspects of the IEEE Standard for Binary Floating-Point Arithmetic are discussed in this manual. To learn about IEEE arithmetic, see the 18-page Standard. See Appendix F for a brief bibliography on IEEE arithmetic.

Who Should Use This Book

This manual is written for those who develop, maintain, and port mathematical and scientific applications or benchmarks. Before using this manual, you should be familiar with the programming language used (Fortran, C, etc.), `dbx` (the source-level debugger), and the operating system commands and concepts.

How This Book Is Organized

Chapter 1 introduces the floating-point environment.

Chapter 2 describes the IEEE arithmetic model, IEEE formats, and underflow.

Chapter 3 describes the mathematics libraries provided with the Sun Forte™ Developer compilers.

Chapter 4 describes exceptions and shows how to detect, locate, and handle them.

Appendix A contains example programs.

Appendix B describes the floating-point hardware options for SPARC workstations.

Appendix C lists x86 and SPARC compatibility issues related to the floating-point units used in Intel systems.

Appendix D is an edited reprint of a tutorial on floating-point arithmetic by David Goldberg.

Appendix E discusses standards compliance.

Appendix F includes a list of references and related documentation.

Glossary contains a definition of terms.

The examples in this manual are in C and Fortran, but the concepts apply to either compiler on a SPARC or Intel system.

Typographic Conventions

TABLE P-1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type **`rm`** *filename*.

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	O[*n*]	O4, O
{ }	Braces contain a set of choices for required option.	d{y|n}	dy
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	B{dynamic|static}	Bstatic
:	The colon, like the comma, is sometimes used to separate arguments.	R*dir*[:*dir*]	R/local/libs:/U/a
…	The ellipsis indicates omission in a series.	xinline=*f1*[,…*fn*]	xinline=alpha,dos

Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Accessing Forte Developer Development Tools and Man Pages

The Forte Developer product components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the Forte Developer compilers and tools, you must have the Forte Developer component directory in your PATH environment variable. To access the Forte Developer man pages, you must have the Forte Developer man page directory in your MANPATH environment variable.

For more information about the PATH variable, see the csh(1), sh(1), and ksh(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH and MANPATH variables to access this Forte Developer release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Forte Developer products are installed in the /opt directory. If your product software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

Accessing Forte Developer Compilers and Tools

Use the steps below to determine whether you need to change your PATH variable to access the Forte Developer compilers and tools.

▼ To Determine Whether You Need to Set Your PATH Environment Variable

1. **Display the current value of the PATH variable by typing the following at a command prompt:**

```
% echo $PATH
```

2. **Review the output for a string of paths that contain /opt/SUNWspro/bin/.**

   If you find the path, your PATH variable is already set to access Forte Developer development tools. If you do not find the path, set your PATH environment variable by following the instructions in the next section.

▼ To Set Your PATH Environment Variable to Enable Access to Forte Developer Compilers and Tools

1. **If you are using the C shell, edit your home .cshrc file. If you are using the Bourne shell or Korn shell, edit your home .profile file.**
2. **Add the following to your PATH environment variable.**

   /opt/SUNWspro/bin

Accessing Forte Developer Man Pages

Use the following steps to determine whether you need to change your MANPATH variable to access the Forte Developer man pages.

▼ To Determine Whether You Need to Set Your MANPATH Environment Variable

1. **Request the** dbx **man page by typing the following at a command prompt:**

```
% man dbx
```

2. **Review the output, if any.**

   If the dbx(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next section for setting your MANPATH environment variable.

▼ To Set Your MANPATH Environment Variable to Enable Access to Forte Developer Man Pages

1. **If you are using the C shell, edit your home** .cshrc **file. If you are using the Bourne shell or Korn shell, edit your home** .profile **file.**

2. **Add the following to your** MANPATH **environment variable.**

   /opt/SUNWspro/man

Accessing Forte Developer Documentation

You can access Forte Developer product documentation at the following locations:

- The product documentation is available from the documentation index installed with the product on your local system or network at `/opt/SUNWspro/docs/index.html`.

  If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`sm web site. The following titles are available through your installed product only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

Product Documentation in Accessible Formats

Forte Developer 7 product documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Release notes	Text file on the product CD at `/cdrom/devpro_v10n1_sparc/release_notes.txt`

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	**Document Title**	**Description**
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

CHAPTER 1

Introduction

Sun’s floating-point environment on SPARC and Intel systems enables you to develop robust, high-performance, portable numerical applications. The floating-point environment can also help investigate unusual behavior of numerical programs written by others. These systems implement the arithmetic model specified by IEEE Standard 754 for Binary Floating Point Arithmetic. This manual explains how to use the options and flexibility provided by the IEEE Standard on these systems.

Floating-Point Environment

The *floating-point environment* consists of data structures and operations made available to the applications programmer by hardware, system software, and software libraries that together implement IEEE Standard 754. IEEE Standard 754 makes it easier to write numerical applications. It is a solid, well-thought-out basis for computer arithmetic that advances the art of numerical programming.

For example, the hardware provides storage formats corresponding to the IEEE data formats, operations on data in such formats, control over the rounding of results produced by these operations, status flags indicating the occurrence of IEEE numeric exceptions, and the IEEE-prescribed result when such an exception occurs in the absence of a user-defined handler for it. System software supports IEEE exception handling. The software libraries, including the math libraries, `libm` and `libsunmath`, implement functions such as `exp(x)` and `sin(x)` in a way that follows the spirit of IEEE Standard 754 with respect to the raising of exceptions. (When a floating-point arithmetic operation has no well-defined result, the system communicates this fact to the user by *raising an exception*.) The math libraries also provide function calls that handle special IEEE values like `Inf` (infinity) or `NaN` (Not a Number).

The three constituents of the floating-point environment interact in subtle ways, and those interactions are generally invisible to the applications programmer. The programmer sees only the computational mechanisms prescribed or recommended by the IEEE standard. In general, this manual guides programmers to make full and efficient use of the IEEE mechanisms so that they can write application software effectively.

Many questions about floating-point arithmetic concern elementary operations on numbers. For example,

- What is the result of an operation when the infinitely precise result is not representable in the computer system?
- Are elementary operations like multiplication and addition commutative?

Another class of questions is connected to exceptions and exception handling. For example, what happens when you:

- Multiply two very large numbers?
- Divide by zero?
- Attempt to compute the square root of a negative number?

In some other arithmetics, the first class of questions might not have the expected answers, or the exceptional cases in the second class are treated the same: the program aborts on the spot; in some very old machines, the computation proceeds, but with garbage.

The IEEE Standard 754 ensures that operations yield the mathematically expected results with the expected properties. It also ensures that exceptional cases yield specified results, unless the user specifically makes other choices.

In this manual, there are references to terms like `NaN` or *subnormal number*. The Glossary defines terms related to floating-point arithmetic.

CHAPTER 2

IEEE Arithmetic

This chapter discusses the arithmetic model specified by the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (“the IEEE standard” or “IEEE 754” for short). All SPARC and x86 processors use IEEE arithmetic. All Sun compiler products support the features of IEEE arithmetic.

IEEE Arithmetic Model

This section describes the IEEE 754 specification.

What Is IEEE Arithmetic?

IEEE 754 specifies:

- Two basic floating-point formats: *single* and *double*.

  The IEEE single format has a significand precision of 24 bits and occupies 32 bits overall. The IEEE double format has a significand precision of 53 bits and occupies 64 bits overall.

- Two classes of extended floating-point formats: *single extended* and *double extended*.

  The standard does not prescribe the exact precision and size of these formats, but it does specify the minimum precision and size. For example, an IEEE double extended format must have a significand precision of at least 64 bits and occupy at least 79 bits overall.

- Accuracy requirements on floating-point operations: *add, subtract, multiply, divide, square root, remainder, round numbers in floating-point format to integer values, convert between different floating-point formats, convert between floating-point and integer formats, and compare.*

The remainder and compare operations must be exact. Each of the other operations must deliver to its destination the exact result, unless there is no such result or that result does not fit in the destination's format. In the latter case, the operation must minimally modify the exact result according to the rules of prescribed rounding modes, presented below, and deliver the result so modified to the operation's destination.

- Accuracy, monotonicity and identity requirements for conversions between decimal strings and binary floating-point numbers in either of the basic floating-point formats.

  For operands lying within specified ranges, these conversions must produce exact results, if possible, or minimally modify such exact results in accordance with the rules of the prescribed rounding modes. For operands not lying within the specified ranges, these conversions must produce results that differ from the exact result by no more than a specified tolerance that depends on the rounding mode.

- Five types of IEEE floating-point exceptions, and the conditions for indicating to the user the occurrence of exceptions of these types.

  The five types of floating-point exceptions are *invalid operation, division by zero, overflow, underflow,* and *inexact.*

- Four rounding directions: *toward the nearest representable value*, with "even" values preferred whenever there are two nearest representable values; *toward negative infinity* (down); *toward positive infinity* (up); and *toward 0* (chop).
- Rounding precision; for example, if a system delivers results in double extended format, the user should be able to specify that such results are to be rounded to the precision of either the single or double format.

The IEEE standard also recommends support for user handling of exceptions.

The features required by the IEEE standard make it possible to support interval arithmetic, the retrospective diagnosis of anomalies, efficient implementations of standard elementary functions like `exp` and `cos`, multiple precision arithmetic, and many other tools that are useful in numerical computation.

IEEE 754 floating-point arithmetic offers users greater control over computation than does any other kind of floating-point arithmetic. The IEEE standard simplifies the task of writing numerically sophisticated, portable programs not only by imposing rigorous requirements on conforming implementations, but also by allowing such implementations to provide refinements and enhancements to the standard itself.

IEEE Formats

This section describes how floating-point data is stored in memory. It summarizes the precisions and ranges of the different IEEE storage formats.

Storage Formats

A floating-point format is a data structure specifying the fields that comprise a floating-point numeral, the layout of those fields, and their arithmetic interpretation. A floating-point *storage* format specifies how a floating-point format is stored in memory. The IEEE standard defines the formats, but it leaves to implementors the choice of storage formats.

Assembly language software sometimes relies on using the storage formats, but higher level languages usually deal only with the linguistic notions of floating-point data types. These types have different names in different high-level languages, and correspond to the IEEE formats as shown in TABLE 2-1.

TABLE 2-1 IEEE Formats and Language Types

IEEE Precision	C, C++	Fortran (SPARC only)
single	float	`REAL` *or* `REAL*4`
double	double	`DOUBLE PRECISION` *or* `REAL*8`
double extended	long double	`REAL*16`

IEEE 754 specifies exactly the single and double floating-point formats, and it defines a class of extended formats for each of these two basic formats. The `long double` and `REAL*16` types shown in TABLE 2-1 refer to one of the class of double extended formats defined by the IEEE standard.

The following sections describe in detail each of the storage formats used for the IEEE floating-point formats on SPARC and x86 platforms.

Single Format

The IEEE single format consists of three fields: a 23-bit fraction, `f`; an 8-bit biased exponent, `e`; and a 1-bit sign, `s`. These fields are stored contiguously in one 32-bit word, as shown in FIGURE 2-1. Bits 0:22 contain the 23-bit fraction, `f`, with bit 0 being the least significant bit of the fraction and bit 22 being the most significant; bits 23:30 contain the 8-bit biased exponent, `e`, with bit 23 being the least significant bit of the biased exponent and bit 30 being the most significant; and the highest-order bit 31 contains the sign bit, `s`.

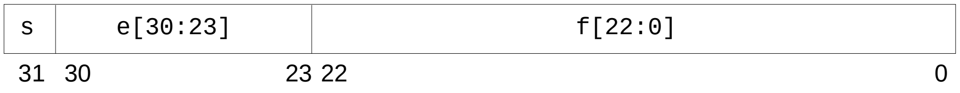


FIGURE 2-1 Single-Storage Format

TABLE 2-2 shows the correspondence between the values of the three constituent fields `s`, `e` and `f`, on the one hand, and the value represented by the single- format bit pattern on the other; *u* means *don't care*, that is, the value of the indicated field is irrelevant to the determination of the value of the particular bit patterns in single format.

TABLE 2-2 Values Represented by Bit Patterns in IEEE Single Format

Single-Format Bit Pattern	Value
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$ (normal numbers)
$e = 0;\ f \neq 0$ (at least one bit in `f` is nonzero)	$(-1)^s \times 2^{-126} \times 0.f$ (subnormal numbers)
$e = 0;\ f = 0$ (all bits in `f` are zero)	$(-1)^s \times 0.0$ (signed zero)
$s = 0;\ e = 255;\ f = 0$ (all bits in `f` are zero)	+INF (positive infinity)
$s = 1;\ e = 255;\ f = 0$ (all bits in `f` are zero)	–INF (negative infinity)
$s = u;\ e = 255;\ f \neq 0$ (at least one bit in `f` is nonzero)	NaN (Not-a-Number)

Notice that when `e` < 255, the value assigned to the single format bit pattern is formed by inserting the binary radix point immediately to the left of the fraction's most significant bit, and inserting an implicit bit immediately to the left of the binary point, thus representing in binary positional notation a mixed number (whole number plus fraction, wherein 0 ≤ Ò3fraction < 1).

The mixed number thus formed is called the *single-format significand*. The implicit bit is so named because its value is not explicitly given in the single- format bit pattern, but is implied by the value of the biased exponent field.

For the single format, the difference between a normal number and a subnormal number is that the leading bit of the significand (the bit to left of the binary point) of a normal number is 1, whereas the leading bit of the significand of a subnormal number is 0. Single-format subnormal numbers were called single-format denormalized numbers in IEEE Standard 754.

The 23-bit fraction combined with the implicit leading significand bit provides 24 bits of precision in single-format normal numbers.

Examples of important bit patterns in the single-storage format are shown in TABLE 2-3. The maximum positive normal number is the largest finite number representable in IEEE single format. The minimum positive subnormal number is the smallest positive number representable in IEEE single format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 2-3 Bit Patterns in Single-Storage Format and Their IEEE Values

Common Name	Bit Pattern (Hex)	Decimal Value
+0	`00000000`	0.0
–0	`80000000`	–0.0
1	`3f800000`	1.0
2	`40000000`	2.0
maximum normal number	`7f7fffff`	3.40282347e+38
minimum positive normal number	`00800000`	1.17549435e–38
maximum subnormal number	`007fffff`	1.17549421e–38
minimum positive subnormal number	`00000001`	1.40129846e–45
$+\infty$	`7f800000`	Infinity
$-\infty$	`ff800000`	–Infinity
Not-a-Number	`7fc00000`	NaN

A NaN (Not a Number) can be represented with any of the many bit patterns that satisfy the definition of a NaN. The hex value of the NaN shown in TABLE 2-3 is just one of the many bit patterns that can be used to represent a NaN.

Double Format

The IEEE double format consists of three fields: a 52-bit fraction, `f`; an 11-bit biased exponent, `e`; and a 1-bit sign, `s`. These fields are stored contiguously in two successively addressed 32-bit words, as shown in FIGURE 2-2.

In the SPARC architecture, the higher address 32-bit word contains the least significant 32 bits of the fraction, while in the x86 architecture the lower address 32-bit word contains the least significant 32 bits of the fraction.

If we denote `f`[31:0] the least significant 32 bits of the fraction, then bit 0 is the least significant bit of the entire fraction and bit 31 is the most significant of the 32 least significant fraction bits.

In the other 32-bit word, bits 0:19 contain the 20 most significant bits of the fraction, `f`[51:32], with bit 0 being the least significant of these 20 most significant fraction bits, and bit 19 being the most significant bit of the entire fraction; bits 20:30 contain the 11-bit biased exponent, `e`, with bit 20 being the least significant bit of the biased exponent and bit 30 being the most significant; and the highest-order bit 31 contains the sign bit, `s`.

FIGURE 2-2 numbers the bits as though the two contiguous 32-bit words were one 64-bit word in which bits 0:51 store the 52-bit fraction, `f`; bits 52:62 store the 11-bit biased exponent, `e`; and bit 63 stores the sign bit, `s`.

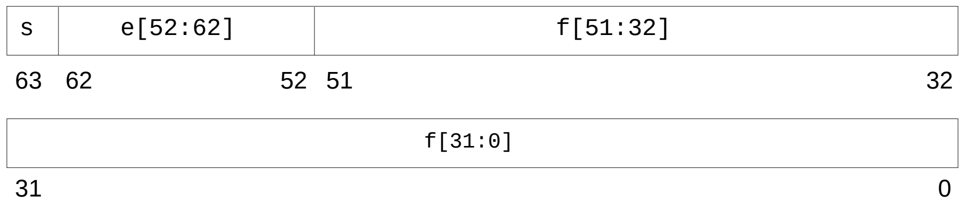


FIGURE 2-2 Double-Storage Format

The values of the bit patterns in these three fields determine the value represented by the overall bit pattern.

TABLE 2-4 shows the correspondence between the values of the bits in the three constituent fields, on the one hand, and the value represented by the double-format bit pattern on the other; *u* means *don't care*, because the value of the indicated field is irrelevant to the determination of value for the particular bit pattern in double format.

TABLE 2-4 Values Represented by Bit Patterns in IEEE Double Format

Double-Format Bit Pattern	Value
$0 < \texttt{e} < 2047$	$(-1)^s \times 2^{e-1023}$ x $1.\texttt{f}$ (normal numbers)
$\texttt{e} = 0$; $\texttt{f} \neq 0$ (at least one bit in `f` is nonzero)	$(-1)^s \times 2^{-1022}$ x $0.\texttt{f}$ (subnormal numbers)
$\texttt{e} = 0$; $\texttt{f} = 0$ (all bits in `f` are zero)	$(-1)^s \times 0.0$ (signed zero)
$\texttt{s} = 0$; $\texttt{e} = 2047$; $\texttt{f} = 0$ (all bits in `f` are zero)	+INF (positive infinity)
$\texttt{s} = 1$; $\texttt{e} = 2047$; $\texttt{f} = 0$ (all bits in `f` are zero)	–INF (negative infinity)
$\texttt{s} = \texttt{u}$; $\texttt{e} = 2047$; $\texttt{f} \neq 0$ (at least one bit in `f` is nonzero)	NaN (Not-a-Number)

Notice that when $\texttt{e} < 2047$, the value assigned to the double-format bit pattern is formed by inserting the binary radix point immediately to the left of the fraction's most significant bit, and inserting an implicit bit immediately to the left of the binary point. The number thus formed is called the *significand*. The implicit bit is so named because its value is not explicitly given in the double-format bit pattern, but is implied by the value of the biased exponent field.

For the double format, the difference between a normal number and a subnormal number is that the leading bit of the significand (the bit to the left of the binary point) of a normal number is 1, whereas the leading bit of the significand of a subnormal number is 0. Double-format subnormal numbers were called double-format denormalized numbers in IEEE Standard 754.

The 52-bit fraction combined with the implicit leading significand bit provides 53 bits of precision in double-format normal numbers.

Examples of important bit patterns in the double-storage format are shown in TABLE 2-5. The bit patterns in the second column appear as two 8-digit hexadecimal numbers. For the SPARC architecture, the left one is the value of the lower addressed 32-bit word, and the right one is the value of the higher addressed 32-bit word, while for the x86 architecture, the left one is the higher addressed word, and the right one is the lower addressed word. The maximum positive normal number is the largest finite number representable in the IEEE double format. The minimum positive subnormal number is the smallest positive number representable in IEEE double format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 2-5 Bit Patterns in Double-Storage Format and Their IEEE Values

Common Name	Bit Pattern (Hex)	Decimal Value
+ 0	`00000000 00000000`	0.0
– 0	`80000000 00000000`	–0.0
1	`3ff00000 00000000`	1.0
2	`40000000 00000000`	2.0
max normal number	`7fefffff ffffffff`	1.7976931348623157e+308
min positive normal number	`00100000 00000000`	2.2250738585072014e–308
max subnormal number	`000fffff ffffffff`	2.2250738585072009e–308
min positive subnormal number	`00000000 00000001`	4.9406564584124654e–324
$+\infty$	`7ff00000 00000000`	Infinity
$-\infty$	`fff00000 00000000`	–Infinity
Not-a-Number	`7ff80000 00000000`	NaN

A NaN (Not a Number) can be represented by any of the many bit patterns that satisfy the definition of NaN. The hex value of the NaN shown in TABLE 2-5 is just one of the many bit patterns that can be used to represent a NaN.

Double-Extended Format (SPARC)

The SPARC floating-point environment's quadruple-precision format conforms to the IEEE definition of double-extended format. The quadruple-precision format occupies four 32-bit words and consists of three fields: a 112-bit fraction, f; a 15-bit biased exponent, e; and a 1-bit sign, s. These are stored contiguously as shown in FIGURE 2-3.

The highest addressed 32-bit word contains the least significant 32-bits of the fraction, denoted f[31:0]. The next two 32-bit words contain f[63:32] and f[95:64], respectively. Bits 0:15 of the next word contain the 16 most significant bits of the fraction, f[111:96], with bit 0 being the least significant of these 16 bits, and bit 15 being the most significant bit of the entire fraction. Bits 16:30 contain the 15-bit biased exponent, e, with bit 16 being the least significant bit of the biased exponent and bit 30 being the most significant; and bit 31 contains the sign bit, s.

FIGURE 2-3 numbers the bits as though the four contiguous 32-bit words were one 128-bit word in which bits 0:111 store the fraction, f; bits 112:126 store the 15-bit biased exponent, e; and bit 127 stores the sign bit, s.

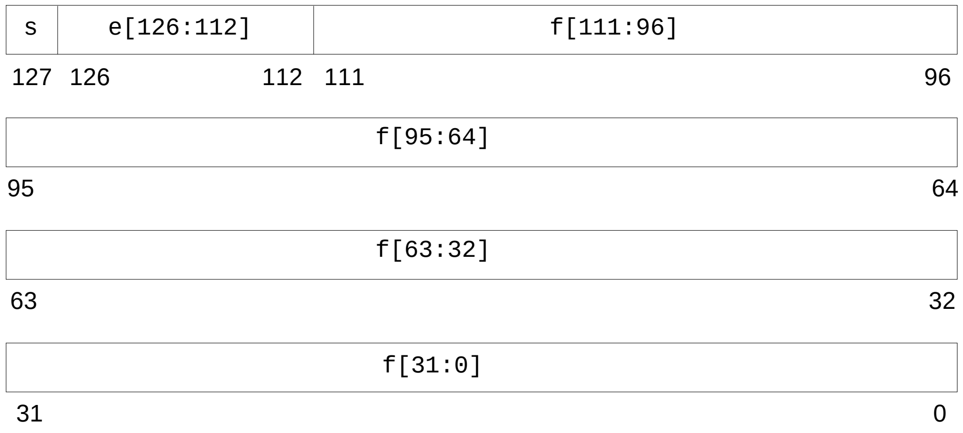


FIGURE 2-3 Double-Extended Format (SPARC)

The values of the bit patterns in the three fields f, e, and s, determine the value represented by the overall bit pattern.

TABLE 2-6 shows the correspondence between the values of the three constituent fields and the value represented by the bit pattern in quadruple-precision format. u means don't care, because the value of the indicated field is irrelevant to the determination of values for the particular bit patterns.

TABLE 2-6 Values Represented by Bit Patterns (SPARC)

Double-Extended Bit Pattern (SPARC)	Value
$0 < e < 32767$	$(-1)^s \times 2^{e-16383} \times 1.f$ (normal numbers)
$e = 0, f \neq 0$ (at least one bit in f is nonzero)	$(-1)^s \times 2^{-16382} \times 0.f$ (subnormal numbers)
$e = 0, f = 0$ (all bits in f are zero)	$(-1)^s \times 0.0$ (signed zero)

TABLE 2-6 Values Represented by Bit Patterns (SPARC) *(Continued)*

Double-Extended Bit Pattern (SPARC)	Value
`s` = 0, `e` = 32767, `f` = 0 (all bits in `f` are zero)	+INF (positive infinity)
`s` = 1, `e` = 32767; `f` = 0 (all bits in `f` are zero)	-INF (negative infinity)
`s` = u, `e` = 32767, `f` ≠ 0 (at least one bit in `f` is nonzero)	NaN (Not-a-Number)

Examples of important bit patterns in the quadruple-precision double-extended storage format are shown in TABLE 2-7. The bit patterns in the second column appear as four 8-digit hexadecimal numbers. The left-most number is the value of the lowest addressed 32-bit word, and the right-most number is the value of the highest addressed 32-bit word. The maximum positive normal number is the largest finite number representable in the quadruple precision format. The minimum positive subnormal number is the smallest positive number representable in the quadruple precision format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 2-7 Bit Patterns in Double-Extended Format (SPARC)

Common Name	Bit Pattern (SPARC)	Decimal Value
+0	`00000000 00000000 00000000 00000000`	0.0
–0	`80000000 00000000 00000000 00000000`	–0.0
1	`3fff0000 00000000 00000000 00000000`	1.0
2	`40000000 00000000 00000000 00000000`	2.0
max normal	`7ffeffff ffffffff ffffffff ffffffff`	1.1897314953572317650857593266280070e+4932
min normal	`00010000 00000000 00000000 00000000`	3.3621031431120935062626778173217526e–4932
max subnormal	`0000ffff ffffffff ffffffff ffffffff`	3.3621031431120935062626778173217520e–4932
min pos subnormal	`00000000 00000000 00000000 00000001`	6.4751751194380251109244389582276466e–4966

TABLE 2-7 Bit Patterns in Double-Extended Format (SPARC) *(Continued)*

Common Name	Bit Pattern (SPARC)	Decimal Value
$+\infty$	`7fff0000 00000000 00000000 00000000`	$+\infty$
$-\infty$	`ffff0000 00000000 00000000 00000000`	$-\infty$
Not-a-Number	`7fff8000 00000000 00000000 00000000`	NaN

The hex value of the NaN shown in TABLE 2-7 is just one of the many bit patterns that can be used to represent NaNs.

Double-Extended Format (x86)

This floating-point environment's double-extended format conforms to the IEEE definition of double-extended formats. It consists of four fields: a 63-bit fraction, `f`; a 1-bit explicit leading significand bit, `j`; a 15-bit biased exponent, `e`; and a 1-bit sign, `s`.

In the family of x86 architectures, these fields are stored contiguously in ten successively addressed 8-bit bytes. However, the UNIX System V Application Binary Interface Intel 386 Processor Supplement (Intel ABI) requires that double-extended parameters and results occupy three consecutively addressed 32-bit words in the stack, with the most significant 16 bits of the highest addressed word being unused, as shown in FIGURE 2-4.

The lowest addressed 32-bit word contains the least significant 32 bits of the fraction, `f`[31:0], with bit 0 being the least significant bit of the entire fraction and bit 31 being the most significant of the 32 least significant fraction bits. In the middle addressed 32-bit word, bits 0:30 contain the 31 most significant bits of the fraction, `f`[62:32], with bit 0 being the least significant of these 31 most significant fraction bits, and bit 30 being the most significant bit of the entire fraction; bit 31 of this middle addressed 32-bit word contains the explicit leading significand bit, `j`.

In the highest addressed 32-bit word, bits 0:14 contain the 15-bit biased exponent, `e`, with bit 0 being the least significant bit of the biased exponent and bit 14 being the most significant; and bit 15 contains the sign bit, `s`. Although the highest order 16 bits of this highest addressed 32-bit word are unused by the family of x86 architectures, their presence is essential for conformity to the Intel ABI, as indicated above.

FIGURE 2-4 numbers the bits as though the three contiguous 32-bit words were one 96-bit word in which bits 0:62 store the 63-bit fraction, `f`; bit 63 stores the explicit leading significand bit, `j`; bits 64:78 store the 15-bit biased exponent, `e`; and bit 79 stores the sign bit, `s`.

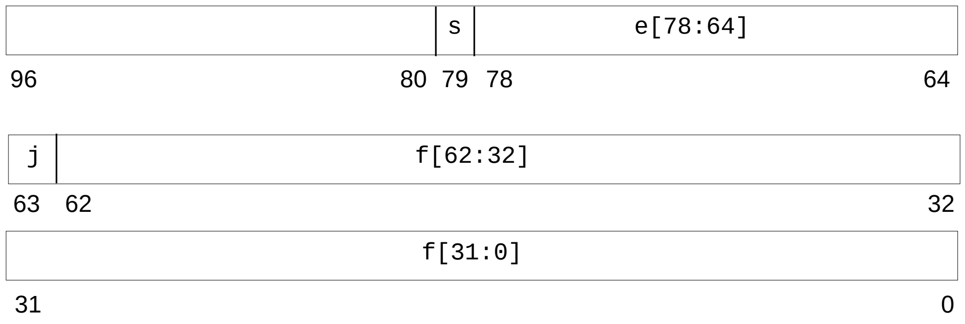


FIGURE 2-4 Double-Extended Format (x86)

The values of the bit patterns in the four fields `f`, `j`, `e` and `s`, determine the value represented by the overall bit pattern.

TABLE 2-8 shows the correspondence between the counting number values of the four constituent field and the value represented by the bit pattern. *u* means *don't care*, because the value of the indicated field is irrelevant to the determination of value for the particular bit patterns.

TABLE 2-8 Values Represented by Bit Patterns (x86)

Double-Extended Bit Pattern (x86)	Value
`j` = 0, 0 < `e` <32767	Unsupported
`j` = 1, 0 < `e` < 32767	$(-1)^s \times 2^{e-16383} \times 1.$`f` (normal numbers)
`j` = 0, `e` = 0; `f` ≠ 0 (at least one bit in `f` is nonzero)	$(-1)^s \times 2^{-16382} \times 0.$`f` (subnormal numbers)
`j` = 1, `e` = 0	$(-1)^s \times 2^{-16382} \times 1.$`f` (pseudo-denormal numbers)
`j` = 0, `e` = 0, `f` = 0 (all bits in f are zero)	$(-1)^s \times 0.0$ (signed zero)
`j` = 1; `s` = 0; `e` = 32767; `f` = 0 (all bits in `f` are zero)	+INF (positive infinity)
`j` = 1; `s` = 1; `e` = 32767; `f` = 0 (all bits in `f` are zero)	–INF (negative infinity)
`j` = 1; `s` = *u*; `e` = 32767; `f` = .1*uuu* — *uu*	QNaN (quiet NaNs)
`j` = 1; `s` = *u*; `e` = 32767; `f` = .0*uuu* — *uu* ≠ 0 (at least one of the *u* in `f` is nonzero)	SNaN (signaling NaNs)

Notice that bit patterns in double-extended format do *not* have an implicit leading significand bit. The leading significand bit is given explicitly as a separate field, `j`, in the double-extended format. However, when `e` ≠ 0, any bit pattern with `j` = 0 is unsupported in the sense that using such a bit pattern as an operand in floating-point operations provokes an invalid operation exception.

The union of the disjoint fields `j` and `f` in the double extended format is called the *significand*. When `e` < 32767 and `j` = 1, or when `e` = 0 and `j` = 0, the significand is formed by inserting the binary radix point between the leading significand bit, `j`, and the fraction's most significant bit.

In the x86 double-extended format, a bit pattern whose leading significand bit `j` is 0 and whose biased exponent field `e` is also 0 represents a subnormal number, whereas a bit pattern whose leading significand bit `j` is 1 and whose biased exponent field `e` is nonzero represents a normal number. Because the leading significand bit is represented explicitly rather than being inferred from the value of the exponent, this format also admits bit patterns whose biased exponent is 0, like the subnormal numbers, but whose leading significand bit is 1. Each such bit pattern actually represents the same value as the corresponding bit pattern whose biased exponent field is 1, i.e., a normal number, so these bit patterns are called *pseudo-denormals*. (Subnormal numbers were called denormalized numbers in IEEE Standard 754.) Pseudo-denormals are merely an artifact of the x86 double-extended format's encoding; they are implicitly converted to the corresponding normal numbers when they appear as operands, and they are never generated as results.

Examples of important bit patterns in the double-extended storage format appear in TABLE 2-9. The bit patterns in the second column appear as one 4-digit hexadecimal counting number, which is the value of the 16 least significant bits of the highest addressed 32-bit word (recall that the most significant 16 bits of this highest addressed 32-bit word are unused, so their value is not shown), followed by two 8-digit hexadecimal counting numbers, of which the left one is the value of the middle addressed 32-bit word, and the right one is the value of the lowest addressed 32-bit word. The maximum positive normal number is the largest finite number representable in the x86 double-extended format. The minimum positive subnormal number is the smallest positive number representable in the double-extended format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 2-9 Bit Patterns in Double-Extended Format and Their Values (x86)

Common Name	Bit Pattern (x86)	Decimal Value
+0	`0000 00000000 00000000`	0.0
–0	`8000 00000000 00000000`	–0.0
1	`3fff 80000000 00000000`	1.0
2	`4000 80000000 00000000`	2.0
max normal	`7ffe ffffffff ffffffff`	1.18973149535723176505e+4932
min positive normal	`0001 80000000 00000000`	3.36210314311209350626e–4932
max subnormal	`0000 7fffffff ffffffff`	3.36210314311209350608e–4932
min positive subnormal	`0000 00000000 00000001`	3.64519953188247460253e–4951
$+\infty$	`7fff 80000000 00000000`	$+\infty$
$-\infty$	`ffff 80000000 00000000`	$-\infty$
quiet NaN with greatest fraction	`7fff ffffffff ffffffff`	QNaN
quiet NaN with least fraction	`7fff c0000000 00000000`	QNaN
signaling NaN with greatest fraction	`7fff bfffffff ffffffff`	SNaN
signaling NaN with least fraction	`7fff 80000000 00000001`	SNaN

A NaN (Not a Number) can be represented by any of the many bit patterns that satisfy the definition of NaN. The hex values of the NaNs shown in TABLE 2-9 illustrate that the leading (most significant) bit of the fraction field determines whether a NaN is quiet (leading fraction bit = 1) or signaling (leading fraction bit = 0).

Ranges and Precisions in Decimal Representation

This section covers the notions of range and precision for a given storage format. It includes the ranges and precisions corresponding to the IEEE single and double formats and to the implementations of IEEE double-extended format on SPARC and x86 architectures. For concreteness, in defining the notions of range and precision we refer to the IEEE single format.

The IEEE standard specifies that 32 bits be used to represent a floating point number in single format. Because there are only finitely many combinations of 32 zeroes and ones, only finitely many numbers can be represented by 32 bits.

One natural question is:

What are the decimal representations of the largest and smallest positive numbers that can be represented in this particular format?

Rephrase the question and introduce the notion of range:

What is the range, in decimal notation, of numbers that can be represented by the IEEE single format?

Taking into account the precise definition of IEEE single format, one can prove that the range of floating-point numbers that can be represented in IEEE single format (if restricted to positive normalized numbers) is as follows:

$1.175... \times (10^{-38})$ to $3.402... \times (10^{+38})$

A second question refers to the precision (not to be confused with the accuracy or the number of significant digits) of the numbers represented in a given format. These notions are explained by looking at some pictures and examples.

The IEEE standard for binary floating-point arithmetic specifies the set of numerical values representable in the single format. Remember that this set of numerical values is described as a set of binary floating-point numbers. The significand of the IEEE single format has 23 bits, which together with the implicit leading bit, yield 24 digits (bits) of (binary) precision.

One obtains a different set of numerical values by marking the numbers:

$x = (x_1.x_2\ x_3...x_q) \times (10^n)$

(representable by q decimal digits in the significand) on the number line.

FIGURE 2-5 exemplifies this situation:

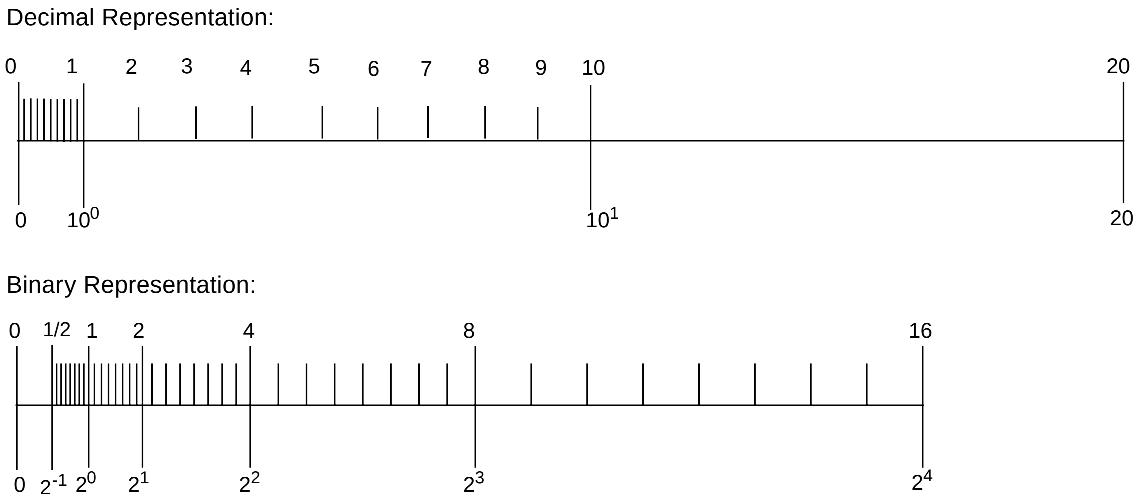


FIGURE 2-5 Comparison of a Set of Numbers Defined by Digital and Binary Representation

Notice that the two sets are different. Therefore, estimating the number of significant decimal digits corresponding to 24 significant binary digits, requires reformulating the problem.

Reformulate the problem in terms of converting floating-point numbers between binary representations (the internal format used by the computer) and the decimal format (the format users are usually interested in). In fact, you may want to convert from decimal to binary and back to decimal, as well as convert from binary to decimal and back to binary.

It is important to notice that because the sets of numbers are different, conversions are in general inexact. If done correctly, converting a number from one set to a number in the other set results in choosing one of the two neighboring numbers from the second set (which one specifically is a question related to rounding).

Consider some examples. Suppose one is trying to represent a number with the following decimal representation in IEEE single format:

x = x1.x2 x3... $\times 10^n$

Because there are only finitely many real numbers that can be represented exactly in IEEE single format, and not all numbers of the above form are among them, in general it will be impossible to represent such numbers exactly. For example, let

$y = 838861.2$, $z = 1.3$

and run the following Fortran program:

```
    REAL Y, Z
    Y = 838861.2
    Z = 1.3
    WRITE(*,40) Y
40  FORMAT("y: ",1PE18.11)
    WRITE(*,50) Z
50  FORMAT("z: ",1PE18.11)
```

The output from this program should be similar to:

```
y:  8.38861187500E+05
z:  1.29999995232E+00
```

The difference between the value 8.388612×10^5 assigned to y and the value printed out is 0.000000125, which is seven decimal orders of magnitude smaller than y. The accuracy of representing y in IEEE single format is about 6 to 7 significant digits, or that y has about *six significant* digits if it is to be represented in IEEE single format.

Similarly, the difference between the value 1.3 assigned to *z* and the value printed out is 0.00000004768, which is eight decimal orders of magnitude smaller than *z*. The accuracy of representing *z* in IEEE single format is about 7 to 8 significant digits, or that *z* has about seven *significant digits* if it is to be represented in IEEE single format.

Now formulate the question:

Assume you convert a decimal floating point number *a* to its IEEE single format binary representation *b*, and then translate *b* back to a decimal number *c*; how many orders of magnitude are between *a* and *a* - *c*?

Rephrase the question:

What is the number of *significant decimal digits* of *a* in the IEEE single format representation, or how many decimal digits are to be trusted as accurate when one represents *x* in IEEE single format?

The number of significant decimal digits is always between 6 and 9, that is, at least 6 digits, but not more than 9 digits are accurate (with the exception of cases when the conversions are exact, when infinitely many digits could be accurate).

Conversely, if you convert a binary number in IEEE single format to a decimal number, and then convert it back to binary, generally, you need to use at least 9 decimal digits to ensure that after these two conversions you obtain the number you started from.

The complete picture is given in TABLE 2-10:

TABLE 2-10 Range and Precision of Storage Formats

Format	Significant Digits (Binary)	Smallest Positive Normal Number	Largest Positive Number	Significant Digits (Decimal)
single	24	1.175... 10^{-38}	3.402... 10^{+38}	6-9
double	53	2.225... 10^{-308}	1.797... 10^{+308}	15-17
double extended (SPARC)	113	3.362... 10^{-4932}	1.189... 10^{+4932}	33-36
double extended (x86)	64	3.362... 10^{-4932}	1.189... 10^{+4932}	18-21

Base Conversion in the Solaris Environment

Base conversion is used by I/O routines, like `printf` and `scanf` in C, and `read`, `write`, and `print` in Fortran. For these functions you need conversions between numbers representations in bases 2 and 10:

- Base conversion from base 10 to base 2 occurs when reading in a number in conventional decimal notation and storing it in internal binary format.
- Base conversion from base 2 to base 10 occurs when printing an internal binary value as an ASCII string of decimal digits.

In the Solaris environment, the fundamental routines for base conversion in all languages are contained in the standard C library, `libc`. These routines use table-driven algorithms that yield correctly-rounded conversion between any input and output formats. In addition to their accuracy, table-driven algorithms reduce the worst-case times for correctly-rounded base conversion.

The IEEE standard requires correct rounding for typical numbers whose magnitudes range from 10^{-44} to 10^{+44} but permits slightly incorrect rounding for larger exponents. (See section 5.6 of IEEE Standard 754.) The `libc` table-driven algorithms round correctly throughout the entire range of single, double, and double extended formats.

In C, conversions between decimal strings and binary floating point values are always rounded correctly in accordance with IEEE 754: the converted result is the number representable in the result's format that is nearest to the original value in the direction specified by the current rounding mode. When the rounding mode is round-to-nearest and the original value lies exactly halfway between two representable numbers in the result format, the converted result is the one whose least significant digit is even. These rules apply to conversions of constants in source code performed by the compiler as well as to conversions of data performed by the program using standard library routines.

In Fortran, conversions between decimal strings and binary floating point values are rounded correctly following the same rules as C by default. For I/O conversions, the “round-ties-to-even” rule in round-to-nearest mode can be overridden, either by using the `ROUNDING=` specifier in the program or by compiling with the `-iorounding` flag. See the *Fortran User’s Guide* and the `f95`(1) man page for more information.

See Appendix F for references on base conversion. Particularly good references are Coonen’s thesis and Sterbenz’s book.

Underflow

Underflow occurs, roughly speaking, when the result of an arithmetic operation is so small that it cannot be stored in its intended destination format without suffering a rounding error that is larger than usual.

Underflow Thresholds

TABLE 2-11 shows the underflow thresholds for single, double, and double-extended precision.

TABLE 2-11 Underflow Thresholds

Destination Precision	Underflow Threshold	
single	smallest normal number	1.17549435e–38
	largest subnormal number	1.17549421e–38
double	smallest normal number	2.2250738585072014e–308
	largest subnormal number	2.2250738585072009e–308
double-extended (SPARC)	smallest normal number	3.3621031431120935062626778173217526e–4932
	largest subnormal number	3.3621031431120935062626778173217520e–4932
double-extended (x86)	smallest normal number	3.36210314311209350626e–4932
	largest subnormal number	3.36210314311209350590e–4932

The positive subnormal numbers are those numbers between the smallest normal number and zero. Subtracting two (positive) tiny numbers that are near the smallest normal number might produce a subnormal number. Or, dividing the smallest positive normal number by two produces a subnormal result.

The presence of subnormal numbers provides greater precision to floating-point calculations that involve small numbers, although the subnormal numbers themselves have fewer bits of precision than normal numbers. Producing subnormal numbers (rather than returning the answer zero) when the mathematically correct result has magnitude less than the smallest positive normal number is known as gradual underflow.

There are several other ways to deal with such *underflow* results. One way, common in the past, was to flush those results to zero. This method is known as *Store 0* and was the default on most mainframes before the advent of the IEEE Standard.

The mathematicians and computer designers who drafted IEEE Standard 754 considered several alternatives while balancing the desire for a mathematically robust solution with the need to create a standard that could be implemented efficiently.

How Does IEEE Arithmetic Treat Underflow?

IEEE Standard 754 chooses gradual underflow as the preferred method for dealing with underflow results. This method amounts to defining two representations for stored values, normal and subnormal.

Recall that the IEEE format for a normal floating-point number is:

$$(-1)^s \times (2^{(e - bias)}) \times 1.f$$

where s is the sign bit, e is the biased exponent, and f is the fraction. Only s, e, and f need to be stored to fully specify the number. Because the implicit leading bit of the significand is defined to be 1 for normal numbers, it need not be stored.

The smallest positive normal number that can be stored, then, has the negative exponent of greatest magnitude and a fraction of all zeros. Even smaller numbers can be accommodated by considering the leading bit to be zero rather than one. In the double-precision format, this effectively extends the minimum exponent from 10^{-308} to 10^{-324}, because the fraction part is 52 bits long (roughly 16 decimal digits.) These are the *subnormal* numbers; returning a subnormal number (rather than flushing an underflowed result to zero) is *gradual underflow*.

Clearly, the smaller a subnormal number, the fewer nonzero bits in its fraction; computations producing subnormal results do not enjoy the same bounds on relative roundoff error as computations on normal operands. However, the key fact about gradual underflow is that its use implies:

- Underflowed results need never suffer a loss of accuracy any greater than that which results from ordinary roundoff error.
- Addition, subtraction, comparison, and remainder are always exact when the result is very small.

Recall that the IEEE format for a subnormal floating-point number is:

$$(-1)^s \times (2^{(-bias + 1)}) \times 0.f$$

where s is the sign bit, the biased exponent e is zero, and f is the fraction. Note that the implicit power-of-two bias is one greater than the bias in the normal format, and the implicit leading bit of the fraction is zero.

Gradual underflow allows you to extend the lower range of representable numbers. It is not *smallness* that renders a value questionable, but its associated error. Algorithms exploiting subnormal numbers have smaller error bounds than other systems. The next section provides some mathematical justification for gradual underflow.

Why Gradual Underflow?

The purpose of subnormal numbers is not to avoid underflow/overflow entirely, as some other arithmetic models do. Rather, subnormal numbers eliminate underflow as a cause for concern for a variety of computations (typically, multiply followed by add). For a more detailed discussion, see *Underflow and the Reliability of Numerical Software* by James Demmel and *Combatting the Effects of Underflow and Overflow in Determining Real Roots of Polynomials* by S. Linnainmaa.

The presence of subnormal numbers in the arithmetic means that untrapped underflow (which implies loss of accuracy) cannot occur on addition or subtraction. If x and y are within a factor of two, then $x - y$ is error-free. This is critical to a number of algorithms that effectively increase the working precision at critical places in algorithms.

In addition, gradual underflow means that errors due to underflow are no worse than usual roundoff error. This is a much stronger statement than can be made about any other method of handling underflow, and this fact is one of the best justifications for gradual underflow.

Error Properties of Gradual Underflow

Most of the time, floating-point results are rounded:

$$\textit{computed result} = \textit{true result} + \textit{roundoff}$$

How large can the roundoff be? One convenient measure of its size is called a *unit in the last place*, abbreviated *ulp*. The least significant bit of the fraction of a floating-point number in its standard representation is its *last place*. The value represented by this bit (e.g., the absolute difference between the two numbers whose representations are identical except for this bit) is a *unit in the last place* of that number. If the computed result is obtained by rounding the true result to the nearest representable number, then clearly the roundoff error is no larger than half a unit in the last place of the computed result. In other words, in IEEE arithmetic with rounding mode to nearest,

$$0 \le |\textit{roundoff}| \le 1/2 \text{ ulp}$$

of the computed result.

Note that an ulp is a relative quantity. An ulp of a very large number is itself very large, while an ulp of a tiny number is itself tiny. This relationship can be made explicit by expressing an ulp as a function: *ulp(x)* denotes a unit in the last place of the floating-point number x.

Moreover, an ulp of a floating-point number depends on the precision to which that number is represented. For example, TABLE 2-12 shows the values of ulp(1) in each of the four floating-point formats described above:

TABLE 2-12 ulp(1) in Four Different Precisions

Precision	**Value**
single	ulp(1) = 2^-23 ~ 1.192093e-07
double	ulp(1) = 2^-52 ~ 2.220446e-16
double extended (x86)	ulp(1) = 2^-63 ~ 1.084202e-19
quadruple (SPARC)	ulp(1) = 2^-112 ~ 1.925930e-34

Recall that only a finite set of numbers can be exactly represented in any computer arithmetic. As the magnitudes of numbers get smaller and approach zero, the gap between neighboring representable numbers narrows. Conversely, as the magnitude of numbers gets larger, the gap between neighboring representable numbers widens.

For example, imagine you are using a binary arithmetic that has only 3 bits of precision. Then, between any two powers of 2, there are $2^3 = 8$ representable numbers, as shown in FIGURE 2-6.

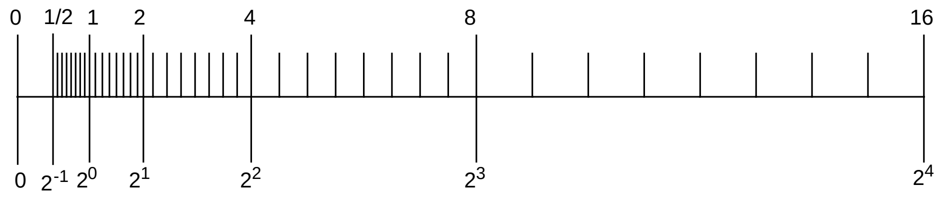


FIGURE 2-6 Number Line

The number line shows how the gap between numbers doubles from one exponent to the next.

In the IEEE single format, the difference in magnitude between the two smallest positive subnormal numbers is approximately 10^{-45}, whereas the difference in magnitude between the two largest finite numbers is approximately 10^{31}!

In TABLE 2-13, `nextafter(x,+∞)` denotes the next representable number after `x` as you move along the number line towards +∞.

TABLE 2-13 Gaps Between Representable Single-Format Floating-Point Numbers

x	nextafter(x, +∞)	Gap
0.0	1.4012985e–45	1.4012985e–45
1.1754944e–38	1.1754945e–38	1.4012985e–45
1.0	1.0000001	1.1920929e–07
2.0	2.0000002	2.3841858e–07
16.000000	16.000002	1.9073486e–06
128.00000	128.00002	1.5258789e–05
1.0000000e+20	1.0000001e+20	8.7960930e+12
9.9999997e+37	1.0000001e+38	1.0141205e+31

Any conventional set of representable floating-point numbers has the property that the worst effect of one inexact result is to introduce an error no worse than the distance to one of the representable neighbors of the computed result. When subnormal numbers are added to the representable set and gradual underflow is implemented, the worst effect of one inexact or *underflow*ed result is to introduce an error no greater than the distance to one of the representable neighbors of the computed result.

In particular, in the region between zero and the smallest *normal* number, the distance between any two neighboring numbers equals the distance between zero and the smallest *subnormal* number. The presence of subnormal numbers eliminates the possibility of introducing a roundoff error that is greater than the distance to the nearest representable number.

Because no calculation incurs roundoff error greater than the distance to any of the representable neighbors of the computed result, many important properties of a robust arithmetic environment hold, including these three:

- $x \neq y \Leftrightarrow x - y \neq 0$
- $(x - y) + y \approx x$, to within a rounding error in the larger of x and y
- $1/(1/x) \approx x$, when x is a normalized number, implying $1/x \neq 0$

An alternative underflow scheme is `Store 0`, which flushes underflow results to zero. `Store 0` violates the first and second properties whenever `x – y` underflows. Also, `Store 0` violates the third property whenever `1/x` underflows.

Let λ represent the smallest positive normalized number, which is also known as the underflow threshold. Then the error properties of gradual underflow and `Store 0` can be compared in terms of λ.

gradual underflow: $|\text{error}| < \frac{1}{2}$ *ulp* in λ

Store 0: $|\text{error}| \approx \lambda$

There is a significant difference between half a unit in the last place of λ, and λ itself.

Two Examples of Gradual Underflow Versus `Store 0`

The following are two well-known mathematical examples. The first example is code that computes an inner product.

```
sum = 0;
for (i = 0; i < n; i++) {
    sum = sum + a[i] * y[i];
}
return sum;
```

With gradual underflow, the result is as accurate as roundoff allows. In `Store 0`, a small but nonzero sum could be delivered that looks plausible but is wrong in nearly every digit. However, in fairness, it must be admitted that to avoid just these sorts of problems, clever programmers scale their calculations if they are able to anticipate where minuteness might degrade accuracy.

The second example, deriving a complex quotient, is not amenable to scaling:

$$a + i \cdot b = \frac{p + i \cdot q}{r + i \cdot s}, \textit{ assuming } |r/s| \le 1$$

$$= \frac{(p \cdot (r/s) + q) + i(q \cdot (r/s) - p)}{s + r \cdot (r/s)}$$

It can be shown that, despite roundoff, the computed complex result differs from the exact result by no more than what would have been the exact result if $p + i \cdot q$ and $r + i \cdot s$ each had been perturbed by no more than a few *ulps*. This error analysis holds in the face of underflows, except that when both a and b underflow, the error is bounded by a few *ulps* of $|a + i \cdot b|$. Neither conclusion is true when underflows are flushed to zero.

This algorithm for computing a complex quotient is robust, and amenable to error analysis, in the presence of gradual underflow. A similarly robust, easily analyzed, and efficient algorithm for computing the complex quotient in the face of `Store 0` *does not exist*. In `Store 0`, the burden of worrying about low-level, complicated details shifts from the implementor of the floating-point environment to its users.

The class of problems that succeed in the presence of gradual underflow, but fail with `Store 0`, is larger than the fans of `Store 0` may realize. Many frequently used numerical techniques fall in this class:

- Linear equation solving
- Polynomial equation solving
- Numerical integration
- Convergence acceleration
- Complex division

Does Underflow Matter?

Despite these examples, it can be argued that underflow rarely matters, and so, why bother? However, this argument turns upon itself.

In the absence of gradual underflow, user programs need to be sensitive to the implicit inaccuracy threshold. For example, in single precision, if underflow occurs in some parts of a calculation, and `Store 0` is used to replace underflowed results with `0`, then accuracy can be guaranteed only to around 10^{-31}, not 10^{-38}, the usual lower range for single-precision exponents.

This means that programmers need to implement their own method of detecting when they are approaching this inaccuracy threshold, or else abandon the quest for a robust, stable implementation of their algorithm.

Some algorithms can be scaled so that computations don't take place in the constricted area near zero. However, scaling the algorithm and detecting the inaccuracy threshold can be difficult and time-consuming for each numerical program.

CHAPTER **3**

The Math Libraries

This chapter describes the math libraries provided with the Solaris operating environment and Forte Developer compilers. Besides listing each of the libraries along with its contents, this chapter discusses some of the features supported by the math libraries provided with the Forte Developer compilers, including IEEE supporting functions, random number generators, and functions that convert data between IEEE and non-IEEE formats.

The contents of the `libm` and `libsunmath` libraries are also listed on the `Intro`(3M) man page.

Standard Math Library

The `libm` math library contains the functions required by the various standards to which the Solaris operating environment conforms. This library is bundled with the Solaris operating environment in two forms: `libm.a`, the static version, and `libm.so`, the shared version.

The default directories for a standard installation of `libm` are:

```
/usr/lib/libm.a
/usr/lib/libm.so
```

The default directories for standard installation of the header files for `libm` are:

```
/usr/include/floatingpoint.h
/usr/include/math.h
/usr/include/sys/ieeefp.h
```

TABLE 3-1 lists the functions in `libm`.

TABLE 3-1 Contents of `libm`

Type	Function Name
Algebraic functions	`cbrt`, `hypot`, `sqrt`
Elementary transcendental functions	`asin`, `acos`, `atan`, `atan2`, `asinh`, `acosh`, `atanh`, `exp`, `expm1`, `pow`, `log`, `log1p`, `log10`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`
Higher transcendental functions	`j0`, `j1`, `jn`, `y0`, `y1`, `yn`, `erf`, `erfc`, `gamma`, `lgamma`, `gamma_r`, `lgamma_r`
Integral rounding functions	`ceil`, `floor`, `rint`
IEEE standard recommended functions	`copysign`, `fmod`, `ilogb`, `nextafter`, `remainder`, `scalbn`, `fabs`
IEEE classification functions	`isnan`
Old style floating-point functions	`logb`, `scalb`, `significand`
Error handling routine (user-defined)	`matherr`

Note that the functions `gamma_r` and `lgamma_r` are reentrant versions of `gamma` and `lgamma`.

See the `ld`(1) and compiler manual pages for more information about dynamic and static linking and the options and environment variables that determine which shared objects are loaded when a program is run.

Additional Math Libraries

Sun Math Library

The library `libsunmath` is part of the libraries supplied with all Sun language products. The library `libsunmath` contains a set of functions that were incorporated in previous versions of `libm` from Sun.

The default directories for a standard installation of `libsunmath` are:

```
/opt/SUNWspro/prod/lib/libsunmath.a
/opt/SUNWspro/lib/libsunmath.so
```

The default directories for standard installation of the header files for `libsunmath` are:

```
/opt/SUNWspro/prod/include/cc/sunmath.h
/opt/SUNWspro/prod/include/floatingpoint.h
```

TABLE 3-2 lists the functions in `libsunmath`. For each mathematical function, the table gives only the name of the double precision version of the function as it would be called from a C program.

TABLE 3-2 Contents of `libsunmath`

Type	Function Name
Functions from TABLE 3-1	single and extended/quadruple precision available, except for `matherr`
Elementary transcendental functions	`exp2`, `exp10`, `log2`, `sincos`
Trigonometric functions in degrees	`asind`, `acosd`, `atand`, `atan2d`, `sind`, `cosd`, `sincosd`, `tand`
Trigonometric functions scaled in π	`asinpi`, `acospi`, `atanpi`, `atan2pi`, `sinpi`, `cospi`, `sincospi`, `tanpi`
Trigonometric functions with double precision π argument reduction	`asinp`, `acosp`, `atanp`, `sinp`, `cosp`, `sincosp`, `tanp`
Financial functions	`annuity`, `compound`
Integral rounding functions	`aint`, `anint`, `irint`, `nint`
IEEE standard recommended functions	`signbit`
IEEE classification functions	`fp_class`, `isinf`, `isnormal`, `issubnormal`, `iszero`
Functions that supply useful IEEE values	`min_subnormal`, `max_subnormal`, `min_normal`, `max_normal`, `infinity`, `signaling_nan`, `quiet_nan`

TABLE 3-2 Contents of `libsunmath` *(Continued)*

Type	Function Name
Additive random number generators	`i_addran_`, `i_addrans_`, `i_init_addrans_`, `i_get_addrans_`, `i_set_addrans_`, `r_addran_`, `r_addrans_`, `r_init_addrans_`, `r_get_addrans_`, `r_set_addrans_`, `d_addran_`, `d_addrans_`, `d_init_addrans_`, `d_get_addrans_`, `d_set_addrans_`, `u_addrans_`
Linear congruential random number generators	`i_lcran_`, `i_lcrans_`, `i_init_lcrans_`, `i_get_lcrans_`, `i_set_lcrans_`, `r_lcran_`, `r_lcrans_`, `d_lcran_`, `d_lcrans_`, `u_lcrans_`
Multiply-with-carry random number generators	`i_mwcran_`, `i_mwcrans_`, `i_init_mwcrans_`, `i_get_mwcrans_`, `i_set_mwcrans`, `i_lmwcran_`, `i_lmwcrans_`, `i_llmwcran_`, `i_llmwcrans_`, `u_mwcran_`, `u_mwcrans_`, `u_lmwcran_`, `u_lmwcrans`, `u_llmwcran_`, `u_llmwcrans_`, `r_mwcran_`, `r_mwcrans_`, `d_mwcran_`, `d_mwcrans_`, `smwcran_`
Random number shufflers	`i_shufrans_`, `r_shufrans_`, `d_shufrans_`, `u_shufrans_`
Data conversion	`convert_external`
Control rounding mode and floating-point exception flags	`ieee_flags`
Floating-point trap handling	`ieee_handler`,`sigfpe`
Show status	`ieee_retrospective`
Enable/disable nonstandard arithmetic	`standard_arithmetic`, `nonstandard_arithmetic`

Optimized Libraries

Optimized versions of some of the routines in `libm` are provided in the library `libmopt`. Optimized versions of some of the support routines in `libc` are provided in the library `libcopt`. Finally, on SPARC systems, alternate forms of some `libc` support routines are provided in `libcx`.

The default directories for a standard installation of `libmopt`, `libcopt`, and `libcx` are:

`/opt/SUNWspro/prod/lib/<`*arch*`>/libmopt.a`

`/opt/SUNWspro/prod/lib/<`*arch*`>/libcopt.a`

`/opt/SUNWspro/prod/lib/<`*arch*`>/libcx.a` (SPARC only)

`/opt/SUNWspro/prod/lib/<`*arch*`>/libcx.so.1` (SPARC only)

Here <*arch*> denotes one of the architecture-specific library directories. On SPARC, these directories include `v7`, `v8`, `v8a`, `v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, `and v9b`. On x86 platforms, the only directory provided is `f80387`.

For a complete description of `-xarch`, see the Fortran, C, or C++ user's guide.

The routines contained in `libcopt` are not intended to be called by the user directly. Instead, they replace support routines in `libc` that are used by the compiler.

The routines contained in `libmopt` replace corresponding routines in `libm`. The `libmopt` versions are generally noticeably faster. Note that unlike the `libm` versions, which can be configured to provide any of ANSI/POSIX, SVID, X/Open, or IEEE-style treatment of exceptional cases, the `libmopt` routines only support IEEE-style handling of these cases. (See Appendix E.)

To link with both `libmopt` and `libcopt` using `cc`, give the `-lmopt` and `-lcopt` options on the command line. (For best results, put `-lmopt` immediately before `-lm` and put `-lcopt` last.) To link with both libraries using any other compiler, specify the `-xlibmopt` flag anywhere on the command line.

SPARC: Library `libcx` contains faster versions of the 128-bit quadruple precision floating point arithmetic support routines. These routines are not intended to be called directly by the user; instead, they are called by the compiler. The C++ compiler links with `libcx` automatically, but the C compiler does not automatically link with `libcx`. To use `libcx` with C programs, link with `-lcx`.

A shared version of `libcx`, called `libcx.so.1`, is also provided. This version can be preloaded at run time by setting the environment variable `LD_PRELOAD` to the full path name of the `libcx.so.1` file. For best performance, use the appropriate version of `libcx.so.1` for your system's architecture. For example, on an UltraSPARC system, assuming the library is installed in the default location, set `LD_PRELOAD` as follows:

csh:

```
setenv LD_PRELOAD /opt/SUNWspro/lib/v8plus/libcx.so.1
```

sh:

```
LD_PRELOAD=/opt/SUNWspro/lib/v8plus/libcx.so.1
export LD_PRELOAD
```

Vector Math Library (SPARC only)

On SPARC platforms, the library `libmvec` provides routines that evaluate common mathematical functions for an entire vector of arguments.

The default directory for a standard installation of `libmvec` is:

/opt/SUNWspro/prod/lib/<*arch*>/libmvec.a

/opt/SUNWspro/prod/lib/<*arch*>/libmvec_mt.a

Here <*arch*> denotes one of the architecture-specific library directories. On SPARC, these directories include `v7`, `v8`, `v8a`, `v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, `and v9b`. On x86 platforms, the only directory provided is `f80387`.

TABLE 3-3 lists the functions in `libmvec`.

TABLE 3-3 Contents of `libmvec`

Type	Function Name
Algebraic functions	`vhypot_`, `vhypotf_`, `vc_abs_`, `vz_abs_`, `vsqrt_`, `vsqrtf_`, `vrsqrt_`, `vrsqrtf_`
Exponential and related functions	`vexp_`, `vexpf_`, `vlog_`, `vlogf_`, `vpow_`, `vpowf_`, `vc_exp_`, `vz_exp_`, `vc_log_`, `vz_log_`, `vc_pow_`, `vz_pow_`
Trigonometric functions	`vatan_`, `vatanf_`, `vatan2_`, `vatan2f_`, `vcos_`, `vcosf_`, `vsin_`, `vsinf_`, `vsincos_`, `vsincosf_`

Note that `libmvec_mt.a` provides parallel versions of the vector functions that rely on multiprocessor parallelization. To use `libmvec_mt.a`, you must link with `-xparallel`.

See the `libmvec`(3m) and `clibmvec`(3m) manual pages for more information.

`libm9x` Math Library

The `libm9x` math library contains some of the math and floating-point related functions specified in C99. In the Forte Developer compilers release, this library contains the `<fenv.h>` Floating-Point Environment functions as well as enhancements to support improved handling of floating-point exceptions.

The default directory for a standard installation of `libm9x` is:

```
/opt/SUNWspro/lib/libm9x.so
```

The default directories for standard installation of the header files for `libm9x` are:

```
/opt/SUNWspro/prod/include/cc/fenv.h
/opt/SUNWspro/prod/include/cc/fenv96.h
```

TABLE 3-4 lists the functions in `libm9x`. (The precision control functions `fegetprec` and `fesetprec` are available only on x86 platforms.)

TABLE 3-4 Contents of `libm9x`

Type	Function Name
C99 standard floating point environment functions	`feclearexcept`, `fegetenv`, `fegetexceptflag`, `fegetround`, `feholdexcept`, `feraiseexcept`, `fesetenv`, `fesetexceptflag`, `fesetround`, `fetestexcept`, `feupdateenv`
Precision control (x86)	`fegetprec`, `fesetprec`
Exception handling and retrospective diagnostics	`fex_get_handling`, `fex_get_log`, `fex_get_log_depth`, `fex_getexcepthandler`, `fex_log_entry`, `fex_merge_flags`, `fex_set_handling`, `fex_set_log`, `fex_set_log_depth`, `fex_setexcepthandler`

Note that `libm9x` is provided as a shared library only. The `cc` compiler does not automatically search for libraries in the shared library installation directory when linking. Therefore, to link with `libm9x` using `cc`, you must enable both the static linker and the run-time linker to locate the library. You can enable the static linker to locate `libm9x` in one of three ways:

- Specify `-L/opt/SUNWspro/lib` before `-lm9x` on the command line.
- Give the full path name `/opt/SUNWspro/lib/libm9x.so` on the command line.
- Add `/opt/SUNWspro/lib` to the list of directories specified by the environment variable `LD_LIBRARY_PATH`.

You can enable the run-time linker to locate `libm9x` in one of three ways:

- Specify `-R/opt/SUNWspro/lib` when linking.
- Add `/opt/SUNWspro/lib` to the list of directories specified by the environment variable `LD_RUN_PATH` when linking.
- Add `/opt/SUNWspro/lib` to the list of directories specified by the environment variable `LD_LIBRARY_PATH` at run time.

Note – Adding `/opt/SUNWspro/lib` to the environment variable `LD_LIBRARY_PATH` can cause a program linked with the Sun Performance Library to use a different version of that library than the one best suited for the system on which the program is run. To use both `libm9x` and the Sun Performance Library in a program linked with `cc`, do not add `/opt/SUNWspro/lib` to `LD_LIBRARY_PATH`. Instead, just specify `-xlic_lib=sunperf` before `-lm9x` on the command line.

All other Forte Developer compilers automatically search the shared library installation directory. To link with `libm9x` using any of these compilers, simply specify `-lm9x` on the command line. (While `libm9x` is primarily intended to be used with C/C++ programs, it is possible to use it with Fortran programs; see Appendix A for an example.)

Single, Double, and Long Double Precision

Most numerical functions are available in single, double, and long-double precision. Examples of calling different precision versions from different languages are shown in TABLE 3-5.

TABLE 3-5 Calling Single, Double, and Quadruple Functions

Language	Single	Double	Quadruple
C, C++	`#include <sunmath.h>`	`#include <math.h>`	`#include <sunmath.h>`
	`float x,y,z;`	`double x,y,z;`	`long double x,y,z;`
	`x = sinf(y);`	`x = sin(y);`	`x = sinl(y);`
	`x = fmodf(y,z);`	`x = fmod(y,z);`	`x = fmodl(y,z);`
	`x = max_normalf();`		`x = max_normall();`
	`x = r_addran_();`	`#include <sunmath.h>`	
		`double x,y,z;`	
		`x = max_normal();`	
		`x = d_addran_();`	
Fortran	`REAL x,y,z`	`REAL*8 x,y,z`	`REAL*16 x,y,z`
	`x = sin(y)`	`x = sin(y)`	`x = sin(y)`
	`x = r_fmod(y,z)`	`x = d_fmod(y,z)`	`x = q_fmod(y,z)`
	`x = r_max_normal()`	`x = d_max_normal()`	`x = q_max_normal()`
	`x = r_addran()`	`x = d_addran()`	

In C, names of single-precision functions are formed by appending f to the double-precision name, and names of quadruple-precision functions are formed by adding l. Because Fortran calling conventions differ, libsunmath provides r_..., d_..., and q_... versions for single, double, and quadruple precision functions, respectively. Fortran intrinsic functions can be called by the generic name for all three precisions.

Not all functions have q_... versions. Refer to math.h and sunmath.h for names and definitions of libm and libsunmath functions.

In Fortran programs, remember to declare r_... functions as real, d_... functions as double precision, and q_... functions as REAL*16. Otherwise, type mismatches might result.

Note – The x86 edition of C supports long double.

IEEE Support Functions

This section describes the IEEE recommended functions, the functions that supply useful values, ieee_flags, ieee_retrospective, and standard_arithmetic and nonstandard_arithmetic. Refer to Chapter 4 for more information on the functions ieee_flags and ieee_handler.

ieee_functions(3m) and ieee_sun(3m)

The functions described by ieee_functions(3m) and ieee_sun(3m) provide capabilities either required by the IEEE standard or recommended in its appendix. These are implemented as efficient bit mask operations.

TABLE 3-6 `ieee_functions`(3m)

Function	Description
`math.h`	Header file
`copysign(x,y)`	`x` with `y`'s sign bit
`fabs(x)`	Absolute value of `x`
`fmod(x, y)`	Remainder of `x` with respect to `y`
`ilogb(x)`	Base 2 unbiased exponent of `x` in integer format
`nextafter(x,y)`	Next representable number after `x`, in the direction `y`
`remainder(x,y)`	Remainder of `x` with respect to `y`
`scalbn(x,n)`	$x \times 2^n$

TABLE 3-7 `ieee_sun`(3m)

Function	Description
`sunmath.h`	Header file
`fp_class(x)`	Classification function
`isinf(x)`	Classification function
`isnormal(x)`	Classification function
`issubnormal(x)`	Classification function
`iszero(x)`	Classification function
`signbit(x)`	Classification function
`nonstandard_arithmetic(void)`	Toggle hardware
`standard_arithmetic(void)`	Toggle hardware
`ieee_retrospective(*f)`	

The `remainder(x,y)` is the operation specified in IEEE Standard 754-1985. The difference between `remainder(x,y)` and `fmod(x,y)` is that the sign of the result returned by `remainder(x,y)` might not agree with the sign of either `x` or `y`, whereas `fmod(x,y)` always returns a result whose sign agrees with `x`. Both functions return exact results and do not generate inexact exceptions.

TABLE 3-8 Calling `ieee_functions` From Fortran

IEEE Function	Single Precision	Double Precision	Quadruple Precision
`copysign(x,y)`	`t=r_copysign(x,y)`	`z=d_copysign(x,y)`	`z=q_copysign(x,y)`
`ilogb(x)`	`i=ir_ilogb(x)`	`i=id_ilogb(x)`	`i=iq_ilogb(x)`
`nextafter(x,y)`	`t=r_nextafter(x,y)`	`z=d_nextafter(x,y)`	`z=q_nextafter(x,y)`
`scalbn(x,n)`	`t=r_scalbn(x,n)`	`z=d_scalbn(x,n)`	`z=q_scalbn(x,n)`
`signbit(x)`	`i=ir_signbit(x)`	`i=id_signbit(x)`	`i=iq_signbit(x)`

TABLE 3-9 Calling `ieee_sun` From Fortran

IEEE Function	Single Precision	Double Precision	Quadruple Precision
`signbit(x)`	`i=ir_signbit(x)`	`i=id_signbit(x)`	`i=iq_signbit(x)`

Note – You must declare `d_`*function* as double precision and `q_`*function* as `REAL*16` in the Fortran program that uses them.

`ieee_values`(3m)

IEEE values like infinity, NaN, maximum and minimum positive floating-point numbers are provided by the functions described by the `ieee_values`(3m) man page. TABLE 3-10, TABLE 3-11, TABLE 3-12, and TABLE 3-12 show the decimal values and hexadecimal IEEE representations of the values provided by `ieee_values`(3m) functions.

TABLE 3-10 IEEE Values: Single Precision

IEEE value	Decimal value hexadecimal representation	C, C++ Fortran

max normal	`3.40282347e+38`
	`7f7fffff`
		`r = max_normalf();`
		`r = r_max_normal()`

min normal	`1.17549435e–38`
	`00800000`
		`r = min_normalf();`
		`r = r_min_normal()`

max subnormal	`1.17549421e–38`
	`007fffff`
		`r = max_subnormalf();`
		`r = r_max_subnormal()`

min subnormal	`1.40129846e–45`
	`00000001`
		`r = min_subnormalf();`
		`r = r_min_subnormal()`

∞	`Infinity`
	`7f800000`
		`r = infinityf();`
		`r = r_infinity()`

quiet NaN	`NaN`
	`7fffffff`
		`r = quiet_nanf(0);`
		`r = r_quiet_nan(0)`

signaling NaN	`NaN`
	`7f800001`
		`r = signaling_nanf(0);`
		`r = r_signaling_nan(0)`

TABLE 3-11 IEEE Values: Double Precision

IEEE value	Decimal Value hexadecimal representation	C, C++ Fortran

max normal	1.7976931348623157e+308
	`7fefffff ffffffff`
		`d = max_normal();`
		`d = d_max_normal()`

min normal	2.2250738585072014e–308
	`00100000 00000000`
		`d = min_normal();`
		`d = d_min_normal()`

max subnormal	2.2250738585072009e–308
	`000fffff ffffffff`
		`d = max_subnormal();`
		`d = d_max_subnormal()`

min subnormal	4.9406564584124654e–324
	`00000000 00000001`
		`d = min_subnormal();`
		`d = d_min_subnormal()`

∞	`Infinity`
	`7ff00000 00000000`
		`d = infinity();`
		`d = d_infinity()`

quiet NaN	`NaN`
	`7fffffff ffffffff`
		`d = quiet_nan(0);`
		`d = d_quiet_nan(0)`

signaling NaN	`NaN`
	`7ff00000 00000001`
		`d = signaling_nan(0);`
		`d = d_signaling_nan(0)`

TABLE 3-12 IEEE Values: Quadruple Precision (SPARC)

IEEE value	Decimal value hexadecimal representation	C, C++ Fortran

max normal	`1.18973149535723176508575932662800 70e+4932`
	`7ffeffff ffffffff ffffffff ffffffff`
		`q = max_normall();`
		`q = q_max_normal()`

min normal	`3.36210314311209350626267781732175 26e-4932`
	`00010000 00000000 00000000 00000000`
		`q = min_normall();`
		`q = q_min_normal()`

max subnormal	`3.36210314311209350626267781732175 20e-4932`
	`0000ffff ffffffff ffffffff ffffffff`
		`q = max_subnormall();`
		`q = q_max_subnormal()`

min subnormal	`6.47517511943802511092443895822764 66e-4966`
	`00000000 00000000 00000000 00000001`
		`q = min_subnormall();`
		`q = q_min_subnormal()`

∞	`Infinity`
	`7fff0000 00000000 00000000 00000000`
		`q = infinityl();`
		`q = q_infinity()`

quiet NaN	`NaN`
	`7fff8000 00000000 00000000 00000000`
		`q = quiet_nanl(0);`
		`q = q_quiet_nan(0)`

signaling NaN	`NaN`
	`7fff0000 00000000 00000000 00000001`
		`q = signaling_nanl(0);`
		`q = q_signaling_nan(0)`

TABLE 3-13 IEEE Values: Double Extended Precision (x86)

IEEE value	Decimal value hexadecimal representation (80 bits)	C, C++

max normal	`1.18973149535723176505e+4932`
	`7ffe ffffffff ffffffff`
		`x = max_normall();`

min positive normal	`3.36210314311209350626e-4932`
	`0001 80000000 00000000`
		`x = min_normall();`

max subnormal	`3.36210314311209350608e-4932`
	`0000 7fffffff ffffffff`
		`x = max_subnormall();`

min positive subnormal	`1.82259976594123730126e-4951`
	`0000 00000000 00000001`
		`x = min_subnormall();`

∞	`Infinity`
	`7fff 80000000 00000000`
		`x = infinityl();`

quiet NaN	`NaN`
	`7fff c0000000 00000000`
		`x = q`

signaling NaN	`NaN`
	`7fff 80000000 00000001`
		`x = signaling_nanl(0);`

ieee_flags(3m)

ieee_flags (3m) is the Sun interface to:

- Query or set rounding direction mode
- Query or set rounding precision mode
- Examine, clear, or set accrued exception flags

The syntax for a call to ieee_flags(3m) is:

```
i = ieee_flags(action, mode, in, out);
```

The ASCII strings that are the possible values for the parameters are shown in TABLE 3-14:

TABLE 3-14 Parameter Values for ieee_flags

Parameter	C or C++ Type	All Possible Values
action	char *	get, set, clear, clearall
mode	char *	direction, precision, exception
in	char *	nearest, tozero, negative, positive, extended, double, single, inexact, division, underflow, overflow, invalid, all, common
out	char **	nearest, tozero, negative, positive, extended, double, single, inexact, division, underflow, overflow, invalid, all, common

The ieee_flags(3m) man page describes the parameters in complete detail.

Some of the arithmetic features that can be modified by using ieee_flags are covered in the following paragraphs. Chapter 4 contains more information on ieee_flags and IEEE exception flags.

When *mode* is direction, the specified action applies to the current rounding direction. The possible rounding directions are: round towards nearest, round towards zero, round towards $+\infty$, or round towards $-\infty$. The IEEE default rounding direction is *round towards nearest*. This means that when the mathematical result of an operation lies strictly between two adjacent representable numbers, the one nearest to the mathematical result is delivered. (If the mathematical result lies exactly halfway between the two nearest representable numbers, then the result delivered is the one whose least significant bit is zero. The *round towards nearest* mode is sometimes called *round to nearest even* to emphasize this.)

Rounding towards zero is the way many pre-IEEE computers work, and corresponds mathematically to truncating the result. For example, if 2/3 is rounded to 6 decimal digits, the result is .666667 when the rounding mode is round towards nearest, but .666666 when the rounding mode is round towards zero.

When using `ieee_flags` to examine, clear, or set the rounding direction, possible values for the four input parameters are shown in TABLE 3-15.

TABLE 3-15 `ieee_flags` Input Values for the Rounding Direction

Parameter	Possible value (mode is `direction`)
`action`	`get`, `set`, `clear`, `clearall`
`in`	`nearest`, `tozero`, `negative`, `positive`
`out`	`nearest`, `tozero`, `negative`, `positive`

When *mode* is `precision`, the specified action applies to the current rounding precision. On x86 platforms, the possible rounding precisions are: single, double, and extended. The default rounding precision is extended; in this mode, arithmetic operations that deliver a result to a floating point register round their result to the full 64-bit precision of the extended double register format. When the rounding precision is single or double, arithmetic operations that deliver a result to a floating point register round their result to 24 or 53 significant bits, respectively. Although most programs produce results that are at least as accurate, if not more so, when extended rounding precision is used, some programs that require strict adherence to the semantics of IEEE arithmetic will not work correctly in extended rounding precision mode and must be run with the rounding precision set to single or double as appropriate.

Rounding precision cannot be set on systems using SPARC processors. On these systems, calling `ieee_flags` with *mode* = `precision` has no effect on computation.

Finally, when *mode* is `exception`, the specified action applies to the current IEEE exception flags. See Chapter 4 for more information about using `ieee_flags` to examine and control the IEEE exception flags.

`ieee_retrospective`(3m)

The `libsunmath` function `ieee_retrospective` prints information about unrequited exceptions and nonstandard IEEE modes. It reports:

- Outstanding exceptions.
- Enabled traps.
- If rounding direction or precision is set to other than the default.
- If nonstandard arithmetic is in effect.

The necessary information is obtained from the hardware floating-point status register.

`ieee_retrospective` prints information about exception flags that are *raised*, and exceptions for which a *trap* is enabled. These two distinct, if related, pieces of information should not be confused. If an exception flag is raised, then that exception occurred at some point during program execution. If a trap is enabled for an exception, then the exception may not have actually occurred (but if it had, a SIGFPE signal would have been delivered). The `ieee_retrospective` message is meant to alert you about exceptions that may need to be investigated (if the exception flag is *raised*), or to remind you that exceptions may have been handled by a signal handler (if the exception's *trap* is enabled.) Chapter 4 discusses exceptions, signals, and traps, and shows how to investigate the cause of a raised exception.

A program can explicitly call `ieee_retrospective` at any time. Fortran programs compiled with `f95` in `-f77` compatibility mode automatically call `ieee_retrospective` before they exit. C/C++ programs and Fortran programs compiled with `f95` in the default mode do not automatically call `ieee_retrospective`.

Note, though, that the `f95` compiler enables trapping on common exceptions by default, so unless a program either explicitly disables trapping or installs a `SIGFPE` handler, it will immediately abort when such an exception occurs. In `-f77` compatibility mode, the compiler does not enable trapping, so when floating point exceptions occur, the program continues execution and reports those exceptions via the `ieee_retrospective` output on exit.

The syntax for calling this function is:

C, C++	`ieee_retrospective(`*fp*`);`
Fortran	`call ieee_retrospective()`

For the C function, the argument *fp* specifies the file to which the output will be written. The Fortran function always prints output on `stderr`.

The following example shows four of the six `ieee_retrospective` warning messages:

```
Note: IEEE floating-point exception flags raised:
   Inexact; Underflow;
Rounding direction toward zero
IEEE floating-point exception traps enabled:
   overflow;
See the Numerical Computation Guide, ieee_flags(3M),
   ieee_handler(3M), ieee_sun(3m)
```

A warning message appears only if trapping is enabled or an exception was raised.

You can suppress ieee_retrospective messages from Fortran programs by one of three methods. One approach is to clear all outstanding exceptions, disable traps, and restore round-to-nearest, extended precision, and standard modes before the program exits. To do this, call ieee_flags, ieee_handler, and standard_arithmetic as follows:

```
character*8 out
i = ieee_flags('clearall', '', '', out)
call ieee_handler('clear', 'all', 0)
call standard_arithmetic()
```

Note – Clearing outstanding exceptions without investigating their cause is not recommended.

Another way to avoid seeing ieee_retrospective messages is to redirect stderr to a file. Of course, this method should not be used if the program sends output other than ieee_retrospective messages to stderr.

The third approach is to include a dummy ieee_retrospective function in the program, for example:

```
subroutine ieee_retrospective
return
end
```

nonstandard_arithmetic(3m)

As discussed in Chapter 2, IEEE arithmetic handles underflowed results using gradual underflow. On some SPARC systems, gradual underflow is often implemented partly with software emulation of the arithmetic. If many calculations underflow, this may cause performance degradation.

To obtain some information about whether this is a case in a specific program, you can use ieee_retrospective or ieee_flags to determine if underflow exceptions occur, and check the amount of system time used by the program. If a program spends an unusually large amount of time in the operating system, and raises underflow exceptions, gradual underflow may be the cause. In this case, using non-IEEE arithmetic may speed up program execution.

The function nonstandard_arithmetic causes underflowed results to be flushed to zero on those SPARC implementations that have a mode in hardware in which flushing to zero is faster. The trade-off for speed is accuracy, because the benefits of gradual underflow are lost.

The function standard_arithmetic resets the hardware to use the default IEEE arithmetic. Both functions have no effect on implementations that provide only the default IEEE 754 style of arithmetic—SuperSPARC™ is such an implementation.

C99 Floating Point Environment Functions

This section describes the <fenv.h> floating point environment functions in C99. In the Forte Developer compilers release, these functions are available in the libm9x.so library. They provide many of the same capabilities as the ieee_flags function, but they use a more natural C interface, and because they are defined by C99, they may prove to be more portable in the future.

Note – For consistent behavior, do not use both C99 floating point environment functions and exception handling extensions in libm9x.so and the ieee_flags and ieee_handler functions in libsunmath in the same program.

Exception Flag Functions

The fenv.h file defines macros for each of the five IEEE floating point exception flags: FE_INEXACT, FE_UNDERFLOW, FE_OVERFLOW, FE_DIVBYZERO, and FE_INVALID. In addition, the macro FE_ALL_EXCEPT is defined to be the bitwise "or" of all five flag macros. In the following descriptions, the *excepts* parameter may be a bitwise "or" of any of the five flag macros or the value FE_ALL_EXCEPT. For the fegetexceptflag and fesetexceptflag functions, the *flagp* parameter must be a pointer to an object of type fexcept_t. (This type is defined in fenv.h.)

C99 defines the following exception flag functions:

TABLE 3-16 C99 Standard Exception Flag Functions

Function	Action
`feclearexcept(`*excepts*`)`	clear specified flags
`fetestexcept(`*excepts*`)`	return settings of specified flags
`feraiseexcept(`*excepts*`)`	raise specified exceptions
`fegetexceptflag(`*flagp*`, `*excepts*`)`	save specified flags in *flagp
`fesetexceptflag(`*flagp*`, `*excepts*`)`	restore specified flags from *flagp

The `feclearexcept` function clears the specified flags. The fetestexcept function returns a bitwise "or" of the macro values corresponding to the subset of flags specified by the *excepts* argument that are set. For example, if the only flags currently set are inexact, underflow, and division by zero, then

i = fetestexcept(`FE_INVALID | FE_DIVBYZERO`);

would set `i` to `FE_DIVBYZERO`.

The `feraiseexcept` function causes a trap if any of the specified exceptions' trap is enabled. (See Chapter 4 for more information on exception traps.) Otherwise, it merely sets the corresponding flags.

The `fegetexceptflag` and `fesetexceptflag` functions provide a convenient way to temporarily save the state of certain flags and later restore them. In particular, the `fesetexceptflag` function does not cause a trap; it merely restores the values of the specified flags.

Rounding Control

The `fenv.h` file defines macros for each of the four IEEE rounding direction modes: `FE_TONEAREST`, `FE_UPWARD` (toward positive infinity), `FE_DOWNWARD` (toward negative infinity), and `FE_TOWARDZERO`. C99 defines two functions to control rounding direction modes: `fesetround` sets the current rounding direction to the direction specified by its argument (which must be one of the four macros above), and `fegetround` returns the value of the macro corresponding to the current rounding direction.

On x86 platforms, the `fenv.h` file defines macros for each of three rounding precision modes: `FE_FLTPREC` (single precision), `FE_DBLPREC` (double precision), and `FE_LDBLPREC` (extended double precision). Although they are not part of C99, `libm9x.so` on x86 provides two functions to control the rounding precision mode:

fesetprec sets the current rounding precision to the precision specified by its argument (which must be one of the three macros above), and fegetprec returns the value of the macro corresponding to the current rounding precision.

Environment Functions

The fenv.h file defines the data type fenv_t, which represents the entire floating point environment including exception flags, rounding control modes, exception handling modes, and, on SPARC, nonstandard mode. In the descriptions that follow, the *envp* parameter must be a pointer to an object of type fenv_t.

C99 defines four functions to manipulate the floating point environment. libm9x.so provides an additional function that may be useful in multi-threaded programs. These functions are summarized in the following table:

TABLE 3-17 libm9x.so Floating Point Environment Functions

Function	Action
fegetenv(*envp*)	save environment in *envp
fesetenv(*envp*)	restore environment from *envp
feholdexcept(*envp*)	save environment in *envp and establish nonstop mode
feupdateenv(*envp*)	restore environment from *envp and raise exceptions
fex_merge_flags(*envp*)	"or" exception flags from *envp

The fegetenv and fesetenv functions respectively save and restore the floating point environment. The argument to fesetenv may be either a pointer to an environment previously saved by a call to fegetenv or feholdexcept or the constant FE_DFL_ENV defined in fenv.h. The latter represents the default environment with all exception flags clear, rounding to nearest (and to extended double precision on x86), nonstop exception handling mode (i.e., traps disabled), and on SPARC, nonstandard mode disabled.

The feholdexcept function saves the current environment and then clears all exception flags and establishes nonstop exception handling mode for all exceptions. The feupdateenv function restores a saved environment (which may be one saved by a call to fegetenv or feholdexcept or the constant FE_DFL_ENV), then raises those exceptions whose flags were set in the previous environment. If the restored environment has traps enabled for any of those exceptions, a trap occurs; otherwise the flags are set. These two functions may be used in conjunction to make a subroutine call appear to be atomic with regard to exceptions, as the following code sample shows:

```
#include <fenv.h>

void myfunc(...) {
    fenv_t env;

    /* save the environment, clear flags, and disable traps */
    feholdexcept(&env);
    /* do a computation that may incur exceptions */
    ...
    /* check for spurious exceptions */
    if (fetestexcept(...)) {
        /* handle them appropriately and clear their flags */
        ...
        feclearexcept(...);
    }
    /* restore the environment and raise relevant exceptions */
    feupdateenv(&env);
}
```

The `fex_merge_flags` function performs a logical `OR` of the exception flags from the saved environment into the current environment without provoking any traps. This function may be used in a multi-threaded program to preserve information in the parent thread about flags that were raised by a computation in a child thread. See Appendix A for an example showing the use of `fex_merge_flags`.

Implementation Features of `libm` and `libsunmath`

This section describes implementation features of `libm` and `libsunmath`:

- Argument reduction using infinitely precise π, and trigonometric functions scaled in π.
- Data conversion routines for converting floating-point data between IEEE and non-IEEE formats.
- Random number generators.

About the Algorithms

The elementary functions in `libm` and `libsunmath` on SPARC systems are implemented with an ever-changing combination of table-driven and polynomial/rational approximation algorithms. Some elementary functions in `libm` and `libsunmath` on x86 platforms are implemented using the elementary function kernel instructions provided in the x86 instruction set; other functions are implemented using the same table-driven or polynomial/rational approximation algorithms used on SPARC.

Both the table-driven and polynomial/rational approximation algorithms for the common elementary functions in `libm` and the common single precision elementary functions in `libsunmath` deliver results that are accurate to within one unit in the last place (*ulp*). On SPARC, the common quadruple precision elementary functions in `libsunmath` deliver results that are accurate to within one *ulp*, except for the `expm1l` and `log1pl` functions, which deliver results accurate to within two *ulps*. (The common functions include the exponential, logarithm, and power functions and circular trigonometric functions of radian arguments. Other functions, such as the hyperbolic trig functions and higher transcendental functions, are less accurate.) These error bounds have been obtained by direct analysis of the algorithms. Users can also test the accuracy of these routines using `BeEF`, the Berkeley Elementary Function test programs, available from `netlib` in the `ucbtest` package (`http://www.netlib.org/fp/ucbtest.tgz`).

Argument Reduction for Trigonometric Functions

Trigonometric functions for radian arguments outside the range $[-\pi/4,\pi/4]$ are usually computed by reducing the argument to the indicated range by subtracting integral multiples of $\pi/2$.

Because π is not a machine-representable number, it must be approximated. The error in the final computed trigonometric function depends on the rounding errors in argument reduction (with an approximate π as well as the rounding), and approximation errors in computing the trigonometric function of the reduced argument. Even for fairly small arguments, the relative error in the final result might be dominated by the argument reduction error, while even for fairly large arguments, the error due to argument reduction may be no worse than the other errors.

There is widespread misapprehension that trigonometric functions of all large arguments are inherently inaccurate, and all small arguments relatively accurate. This is based on the simple observation that large enough machine-representable numbers are separated by a distance greater than π.

There is no inherent boundary at which computed trigonometric function values suddenly become bad, nor are the inaccurate function values useless. Provided that the argument reduction is done consistently, the fact that the argument reduction is performed with an approximation to π is practically undetectable, because all essential identities and relationships are as well preserved for large arguments as for small.

`libm` and `libsunmath` trigonometric functions use an “infinitely” precise π for argument reduction. The value 2/π is computed to 916 hexadecimal digits and stored in a lookup table to use during argument reduction.

The group of functions `sinpi`, `cospi`, and `tanpi` (see TABLE 3-2) scales the input argument by π to avoid inaccuracies introduced by range reduction.

Data Conversion Routines

In `libm` and `libsunmath`, there is a flexible data conversion routine, `convert_external`, used to convert binary floating-point data between IEEE and non-IEEE formats.

Formats supported include those used by SPARC (IEEE), IBM PC, VAX, IBM S/370, and Cray.

Refer to the man page on `convert_external`(3m) for an example of taking data generated on a Cray, and using the function `convert_external` to convert the data into the IEEE format expected on SPARC systems.

Random Number Facilities

There are three facilities for generating uniform pseudo-random numbers in 32-bit integer, single precision floating point, and double precision floating point formats:

- The functions described in the `addrans`(3m) manual page are based on a family of table-driven additive random number generators.
- The functions described in the `lcrans`(3m) manual page are based on a linear congruential random number generator.
- The functions described in the `mwcrans`(3m) manual page are based on multiply-with-carry random number generators. These functions also include generators that supply uniform pseudo-random numbers in 64-bit integer formats.

In addition, the functions described on the `shufrans`(3m) manual page may be used in conjunction with any of these generators to shuffle an array of pseudo-random numbers, thereby providing even more randomness for applications that need it. (Note that there is no facility for shuffling arrays of 64-bit integers.)

Each of the random number facilities includes routines that generate one random number at a time (i.e., one per function call) as well as routines that generate an array of random numbers in a single call. The functions that generate one random number at a time deliver numbers that lie in the ranges shown in TABLE 3-18.

TABLE 3-18 Intervals for Single-Value Random Number Generators

Function	Lower Bound	Upper Bound
`i_addran_`	-2147483648	2147483647
`r_addran_`	0	0.9999999403953552246
`d_addran_`	0	0.9999999999999998890
`i_lcran_`	1	2147483646
`r_lcran_`	4.656612873077392578E-10	1
`d_lcran_`	4.656612875245796923E-10	0.9999999995343387127
`i_mwcran_`	0	2147483647
`u_mwcran_`	0	4294967295
`i_llmwcran_`	0	9223372036854775807
`u_llmwcran_`	0	18446744073709551615
`r_mwcran_`	0	0.9999999403953552246
`d_mwcran_`	0	0.9999999999999998890

The functions that generate an entire array of random numbers in a single call allow the user to specify the interval in which the generated numbers will lie. Appendix A gives several examples that show how to generate arrays of random numbers uniformly distributed over different intervals.

Note that the `addrans` and `mwcrans` generators are generally more efficient than the `lcrans` generators, but their theory is not as refined. "Random Number Generators: Good Ones Are Hard To Find", by S. Park and K. Miller, *Communications of the ACM*, October 1988, discusses the theoretical properties of linear congruential algorithms. Additive random number generators are discussed in Volume 2 of Knuth's *The Art of Computer Programming*.

CHAPTER 4

Exceptions and Exception Handling

This chapter describes IEEE floating point exceptions and shows how to detect, locate, and handle them.

The floating point environment provided by the Forte Developer compilers and the Solaris operating environment on SPARC and x86 platforms supports all of the exception handling facilities required by the IEEE standard as well as many of the recommended optional facilities. One objective of these facilities is explained in the IEEE 854 Standard (IEEE 854, page 18):

> ... to minimize for users the complications arising from exceptional conditions. The arithmetic system is intended to continue to function on a computation as long as possible, handling unusual situations with reasonable default responses, including setting appropriate flags.

To achieve this objective, the standards specify default results for exceptional operations and require that an implementation provide status flags, which can be sensed, set, or cleared by a user, to indicate that exceptions have occurred. The standards also recommend that an implementation provide a means for a program to trap (i.e., interrupt normal control flow) when an exception occurs. The program can optionally supply a trap handler that handles the exception in an appropriate manner, for example by providing an alternate result for the exceptional operation and resuming execution. This chapter lists the exceptions defined by IEEE 754 along with their default results and describes the features of the floating point environment that support status flags, trapping, and exception handling.

What Is an Exception?

It is hard to define exceptions. To quote W. Kahan,

> An arithmetic exception arises when an attempted atomic arithmetic operation has no result that would be acceptable universally. The meanings of atomic and acceptable vary with time and place. (See *Handling Arithmetic Exceptions* by W. Kahan.)

For example, an exception arises when a program attempts to take the square root of a negative number. (This example is one case of an *invalid operation* exception.) When such an exception occurs, the system responds in one of two ways:

- If the exception's trap is disabled (the default case), the system records the fact that the exception occurred and continues executing the program using the default result specified by IEEE 754 for the excepting operation.
- If the exception's trap is enabled, the system generates a `SIGFPE` signal. If the program has installed a `SIGFPE` signal handler, the system transfers control to that handler; otherwise, the program aborts.

IEEE 754 defines five basic types of floating point exceptions: *invalid operation, division by zero, overflow, underflow* and *inexact*. The first three (invalid, division, and overflow) are sometimes collectively called *common exceptions*. These exceptions can seldom be ignored when they occur. `ieee_handler`(3m) gives an easy way to trap on common exceptions only. The other two exceptions (underflow and inexact) are seen more often—in fact, most floating point operations incur the inexact exception—and they can usually, though not always, be safely ignored.

TABLE 4-1 condenses information found in IEEE Standard 754. It describes the five floating point exceptions and the default response of an IEEE arithmetic environment when these exceptions are raised.

TABLE 4-1 IEEE Floating Point Exceptions

IEEE Exception	Reason Why This Arises	Example	Default Result When Trap is Disabled

Invalid operation	An operand is invalid for the operation about to be performed.
	
	(On x86, this exception is also raised when the floating point stack underflows or overflows, though that is not part of the IEEE standard.)
		`0` × ∞
		`0/0`
		∞/ ∞
		`x REM 0`
		Square root of negative operand
		Any operation with a signaling NaN operand
		Unordered comparison (see note 1)
		Invalid conversion (see note 2)
			Quiet `NaN`

Division by zero	An exact infinite result is produced by an operation on finite operands.	`x /0` for finite, nonzero `x`
		`log(0)`
			Correctly signed infinity

Overflow	The correctly rounded result would be larger in magnitude than the largest finite number representable in the destination format (i.e., the exponent range is exceeded).	Double precision: `DBL_MAX` + 1.0e294 `exp`(709.8) Single precision: (float)`DBL_MAX` `FLT_MAX` + 1.0e32 `expf`(88.8)	Depends on rounding mode (RM), and the sign of the intermediate result: RM+ – RN+∞ –∞ RZ +max –max R–+max –∞ R+ +∞ –max
Underflow	Either the exact result or the correctly rounded result would be smaller in magnitude than the smallest normal number representable in the destination format (see note 3).	Double precision: `nextafter`(min_normal,-∞) `nextafter`(min_subnormal,-∞) `DBL_MIN`/3.0 `exp`(-708.5) Single precision: (float)`DBL_MIN` `nextafterf(FLT_MIN`, -∞) `expf`(-87.4)	Subnormal or zero
Inexact	The rounded result of a valid operation is different from the infinitely precise result. (Most floating point operations raise this exception.)	2.0 / 3.0 (float)1.12345678 `log`(1.1) `DBL_MAX` + `DBL_MAX`, when no overflow trap	The result of the operation (rounded, overflowed, or underflowed)

Notes for Table 4-1

1. Unordered comparison: Any pair of floating point values can be compared, even if they are not of the same format. Four mutually exclusive relations are possible: less than, greater than, equal, or unordered. Unordered means that at least one of the operands is a NaN (not a number).

   Every NaN compares "unordered" with everything, including itself. TABLE 4-2 shows which predicates cause the invalid operation exception when the relation is unordered.

TABLE 4-2 Unordered Comparisons

Predicates			Invalid Exception
math	**c, c++**	**f95**	**(if unordered)**
$=$	`==`	`.EQ.`	no
$\neq$	`!=`	`.NE.`	no
$>$	`>`	`.GT.`	yes
$\geq$	`>=`	`.GE.`	yes
$<$	`<`	`.LT.`	yes
$\leq$	`<=`	`.LE.`	yes

2. Invalid conversion: Attempt to convert NaN or infinity to an integer, or integer overflow on conversion from floating point format.

3. The smallest normal numbers representable in the IEEE single, double, and extended formats are 2^{-126}, 2^{-1022}, and 2^{-16382}, respectively. See Chapter 2 for a description of the IEEE floating point formats.

The x86 floating point environment provides another exception not mentioned in the IEEE standards: the *denormal operand* exception. This exception is raised whenever a floating point operation is performed on a subnormal number.

Exceptions are prioritized in the following order: invalid (highest priority), overflow, division, underflow, inexact (lowest priority). On x86 platforms, the denormal operand exception has the lowest priority of all.

The only combinations of standard exceptions that can occur simultaneously are overflow with inexact and underflow with inexact. On x86, the denormal operand exception can occur with any of the five standard exceptions. If trapping on overflow, underflow, and inexact is enabled, the overflow and underflow traps take precedence over the inexact trap; they all take precedence over a denormal operand trap on x86.

Detecting Exceptions

As required by the IEEE standard, the floating point environments on SPARC and x86 platforms provide status flags that record the occurrence of floating point exceptions. A program can test these flags to determine which exceptions have occurred. The flags can also be explicitly set and cleared. The `ieee_flags` function provides one way to access these flags. In programs written in C or C++, the C99 floating point environment functions in `libm9x.so` provide another.

On SPARC, each exception has two flags associated with it, current and accrued. The current exception flags always indicate the exceptions raised by the last floating point instruction to complete execution. These flags are also accumulated (i.e., "or"-ed) into the accrued exception flags thereby providing a record of all untrapped exceptions that have occurred since the program began execution or since the accrued flags were last cleared by the program. (When a floating point instruction incurs a trapped exception, the current exception flag corresponding to the exception that caused the trap is set, but the accrued flags are unchanged.) Both the current and accrued exception flags are contained in the floating point status register, `%fsr`.

On x86, the floating point status word (SW) provides flags for accrued exceptions as well as flags for the status of the floating point stack.

`ieee_flags`(3m)

The syntax for a call to `ieee_flags`(3m) is:

`i = ieee_flags(`*action, mode, in, out*`);`

A program can test, set, or clear the accrued exception status flags using the `ieee_flags` function by supplying the string `"exception"` as the second argument. For example, to clear the overflow exception flag from Fortran, write:

```
      character*8 out
      call ieee_flags('clear', 'exception', 'overflow', out)
```

To determine whether an exception has occurred from C or C++, use:

```
      i = ieee_flags("get", "exception", in, out);
```

When the action is "get", the string returned in *out* is:

- "not available" — if information on exceptions is not available
- "" (an empty string) — if there are no accrued exceptions or, in the case of x86, the denormal operand is the only accrued exception
- the name of the exception named in the third argument, *in*, if that exception has occurred
- otherwise, the name of the highest priority exception that has occurred.

For example, in the Fortran call:

```
      character*8 out
      i = ieee_flags('get', 'exception', 'division', out)
```

the string returned in out is "division" if the division-by-zero exception has occurred; otherwise it is the name of the highest priority exception that has occurred. Note that *in* is ignored unless it names a particular exception; for example, the argument "all" is ignored in the C call:

```
      i = ieee_flags("get", "exception", "all", out);
```

Besides returning the name of an exception in *out*, ieee_flags returns an integer value that combines all of the exception flags currently raised. This value is the bitwise "or" of all the accrued exception flags, where each flag is represented by a single bit as shown in TABLE 4-3. The positions of the bits corresponding to each exception are given by the fp_exception_type values defined in the file sys/ieeefp.h. (Note that these bit positions are machine-dependent and need not be contiguous.)

TABLE 4-3 Exception Bits

Exception	Bit Position	Accrued Exception Bit
invalid	fp_invalid	i & (1 << fp_invalid)
overflow	fp_overflow	i & (1 << fp_overflow)
division	fp_division	i & (1 << fp_division)
underflow	fp_underflow	i & (1 << fp_underflow)
inexact	fp_inexact	i & (1 << fp_inexact)
denormalized	fp_denormalized	i & (1 << fp_denormalized) *(x86 only)*

This fragment of a C or C++ program shows one way to decode the return value.

```
/*
 *   Decode integer that describes all accrued exceptions.
 *   fp_inexact etc. are defined in <sys/ieeefp.h>
 */

char *out;
int invalid, division, overflow, underflow, inexact;

code = ieee_flags("get", "exception", "", &out);
printf ("out is %s, code is %d, in hex: 0x%08X\n",
           out, code, code);
inexact    =   (code >> fp_inexact)& 0x1;
division   =   (code >> fp_division)& 0x1;
underflow  =   (code >> fp_underflow)& 0x1;
overflow   =   (code >> fp_overflow)& 0x1;
invalid    =   (code >> fp_invalid)& 0x1;
printf("%d %d %d %d %d \n", invalid, division, overflow,
           underflow, inexact);
```

C99 Exception Flag Functions

C/C++ programs can test, set, and clear the floating point exception flags using the C99 floating point environment functions in `libm9x.so`. The header file `fenv.h` defines five macros corresponding to the five standard exceptions: `FE_INEXACT`, `FE_UNDERFLOW`, `FE_OVERFLOW`, `FE_DIVBYZERO`, and `FE_INVALID`. It also defines the macro `FE_ALL_EXCEPT` to be the bitwise "or" of all five exception macros. These macros can be combined to test or clear any subset of the exception flags or raise any combination of exceptions. The following examples show the use of these macros with several of the C99 floating point environment functions; see the `feclearexcept`(3M) manual page for more information. (Note: For consistent behavior, do not use both the C99 floating point environment functions and extensions in `libm9x.so` and the `ieee_flags` and `ieee_handler` functions in `libsunmath` in the same program.)

To clear all five exception flags:

```
feclearexcept(FE_ALL_EXCEPT);
```

To test whether the invalid operation or division by zero flags have been raised:

```
int i;

i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
if (i & FE_INVALID)
    /* invalid flag was raised */
else if (i & FE_DIVBYZERO)
    /* division by zero flag was raised */
```

To simulate raising an overflow exception (note that this will provoke a trap if the overflow trap is enabled):

```
feraiseexcept(FE_OVERFLOW);
```

The `fegetexceptflag` and `fesetexceptflag` functions provide a way to save and restore a subset of the flags. The next example shows one way to use these functions.

```
fexcept_t flags;

/* save the underflow, overflow, and inexact flags */
fegetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT);
/* clear these flags */
feclearexcept(FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT);
/* do a computation that can underflow or overflow */
...
/* check for underflow or overflow */
if (fetestexcept(FE_UNDERFLOW | FE_OVERFLOW) != 0) {
    ...
}
/* restore the underflow, overflow, and inexact flags */
fesetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW, | FE_INEXACT);
```

Locating an Exception

Often, programmers do not write programs with exceptions in mind, so when an exception is detected, the first question asked is: Where did the exception occur? One way to locate where an exception occurs is to test the exception flags at various points throughout a program, but to isolate an exception precisely by this approach can require many tests and carry a significant overhead.

An easier way to determine where an exception occurs is to enable its trap. When an exception whose trap is enabled occurs, the operating system notifies the program by sending a SIGFPE signal (see the signal(5) manual page). Thus, by enabling trapping for an exception, you can determine where the exception occurs either by running under a debugger and stopping on receipt of a SIGFPE signal or by establishing a SIGFPE handler that prints the address of the instruction where the exception occurred. Note that trapping must be enabled for an exception to generate a SIGFPE signal; when trapping is disabled and an exception occurs, the corresponding flag is set and execution continues with the default result specified in TABLE 4-1, but no signal is delivered.

Using the Debuggers to Locate an Exception

This section gives examples showing how to use dbx (source-level debugger) and adb (assembly-level debugger) to investigate the cause of a floating point exception and locate the instruction that raised it. Recall that in order to use the source-level debugging features of dbx, programs should be compiled with the -g flag. Refer to the *Debugging a Program With* dbx manual for more information.

Consider the following C program:

```
#include <stdio.h>
#include <math.h>
double sqrtm1(double x)
{
   return sqrt(x) - 1.0;
}

int main(void)
{
   double x, y;

   x = -4.2;
   y = sqrtm1(x);
   printf("%g  %g\n", x, y);
   return 0;
}
```

Compiling and running this program produces:

```
 -4.2  NaN
```

The appearance of a NaN in the output suggests that an invalid operation exception might have occurred. To determine whether this is the case, you can recompile with the -ftrap option to enable trapping on invalid operations and use either dbx or adb to run the program and stop when a SIGFPE signal is delivered. Alternatively, you can use adb or dbx without recompiling the program by linking with a startup routine that enables the invalid operation trap or by manually enabling the trap.

Using dbx to Locate the Instruction Causing an Exception

The simplest way to locate the code that causes a floating point exception is to recompile with the -g and -ftrap flags and then use dbx to track down the location where the exception occurs. First, recompile the program as follows:

```
example% cc -g -ftrap=invalid ex.c -lm
```

Compiling with -g allows you to use the source-level debugging features of dbx. Specifying -ftrap=invalid causes the program to run with trapping enabled for invalid operation exceptions. Next, invoke dbx, issue the catch fpe command to stop when a SIGFPE is issued, and run the program. On SPARC, the result resembles this:

```
example% dbx a.out
Reading a.out
... etc.
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2532)
signal FPE (invalid floating point operation) in __sqrt at 0xff36b3c4
0xff36b3c4: __sqrt+0x003c:      be      __sqrt+0x98
Current function is sqrtm1
    6           return sqrt(x) - 1.0;
(dbx) print x
x = -4.2
(dbx)
```

The output shows that the exception occurred in the sqrtm1 function as a result of attempting to take the square root of a negative number.

Using adb to Locate the Instruction Causing an Exception

You can also use adb to identify the cause of an exception, although adb can't locate the source file and line number as dbx can. Again, the first step is to recompile with -ftrap:

```
example% cc -ftrap=invalid ex.c -lm
```

Now invoke adb and run the program. When an invalid operation exception occurs, adb stops at an instruction following the one that caused the exception. To find the instruction that caused the exception, disassemble several instructions and look for the last floating point instruction prior to the instruction at which adb has stopped. On SPARC, the result might resemble the following transcript.

```
example% adb a.out
:r
SIGFPE: Arithmetic Exception (invalid floating point operation)
stopped at:
__sqrt+0x3c:    be      __sqrt+0x98
__sqrt+30?4i
__sqrt+0x30:    sethi   %hi(0x7ff00000), %o0
                and     %i0, %o0, %o1
                fsqrtd  %f0, %f30
                be      __sqrt+0x98
<f0=F
                -4.2000000000000002e+00
```

The output shows that the exception was caused by an fsqrtd instruction. Examining the source register shows that the exception was a result of attempting to take the square root of a negative number.

On x86, because instructions do not have a fixed length, finding the correct address from which to disassemble the code might involve some trial and error. In this example, the exception occurs close to the beginning of a function, so we can disassemble from there. (Note that this output assumes the program has been compiled with the -xlibmil flag.) The following might be a typical result.

```
example% adb a.out
:r
SIGFPE: Arithmetic Exception (invalid floating point operation)
stopped at:
sqrtm1+0x16:    fstpl  -0x10(%ebp) [0xfffffff0]
sqrtm1?12i
sqrtm1:
sqrtm1:         pushl  %ebp
                movl   %esp,%ebp
                subl   $0x10,%esp [0x10,-]
                movl   0xc(%ebp),%eax [0xc,-]
                pushl  %eax
                movl   0x8(%ebp),%eax [8,-]
                pushl  %eax
                fldl   (%esp,1)
                fsqrt
                addl   $0x8,%esp [8,-]
                fstpl  -0x10(%ebp) [0xfffffff0]
                fldl   -0x10(%ebp) [0xfffffff0]
$x
80387 chip is present.
cw      0x137e
sw      0x3800
cssel 0x17  ipoff 0x691                datasel 0x1f  dataoff 0x0

 st[0]  -4.2000000000000001776356839              VALID
 st[1]  +0.0                                      EMPTY
 st[2]  +0.0                                      EMPTY
 st[3]  +0.0                                      EMPTY
 st[4]  +0.0                                      EMPTY
 st[5]  +0.0                                      EMPTY
 st[6]  +0.0                                      EMPTY
 st[7]  +0.0                                      EMPTY
```

The output reveals that the exception was caused by a `fsqrt` instruction; examination of the floating point registers reveals that the exception was a result of attempting to take the square root of a negative number.

Enabling Traps Without Recompilation

In the preceding examples, trapping on invalid operation exceptions was enabled by recompiling the main subprogram with the `-ftrap` flag. In some cases, recompiling the main program might not be possible, so you might need to resort to other means to enable trapping. There are several ways to do this.

When you are using `dbx`, you can enable traps manually by directly modifying the floating point status register. On SPARC, this can be somewhat tricky because the operating system does not enable the floating point unit until the first time it is used within a program, at which point the floating point status register is reset with all traps disabled. Thus, you cannot manually enable trapping until after the program has executed at least one floating point instruction. In our example, the floating point unit has already been accessed by the time the `sqrtm1` function is called, so we can set a breakpoint on entry to that function, enable trapping on invalid operation exceptions, instruct `dbx` to stop on the receipt of a `SIGFPE` signal, and continue execution. The steps are as follows (note the use of the `assign` command to modify the `%fsr` to enable trapping on invalid operation exceptions):

```
example% dbx a.out
Reading a.out
... etc.
(dbx) stop in sqrtm1
dbx: warning: 'sqrtm1' has no debugger info -- will trigger on first instruction
(2) stop in sqrtm1
(dbx) run
Running: a.out
(process id 23086)
stopped in sqrtm1 at 0x106d8
0x000106d8: sqrtm1       :      save    %sp, -0x70, %sp
(dbx) assign $fsr=0x08000000
dbx: warning: unknown language, 'ansic' assumed
(dbx) catch fpe
(dbx) cont
signal FPE (invalid floating point operation) in __sqrt at 0xff36b3c4
0xff36b3c4: __sqrt+0x003c:      be      __sqrt+0x98
(dbx)
```

On x86, the startup code that the compiler automatically links into every program initializes the floating point unit before transferring control to the main program. Thus, you can manually enable trapping at any time after the main program begins. The following example shows the steps involved:

```
example% dbx a.out
Reading a.out
... etc.
(dbx) stop in main
dbx: warning: 'main' has no debugger info -- will trigger on first instruction
(2) stop in main
(dbx) run
Running: a.out
(process id 25055)
stopped in main at 0x80506b0
0x080506b0: main       :        pushl  %ebp
(dbx) assign $fctrl=0x137e
dbx: warning: unknown language, 'ansic' assumed
(dbx) catch fpe
(dbx) cont
signal FPE (invalid floating point operation) in sqrtm1 at 0x8050696
0x08050696: sqrtm1+0x0016:      fstpl  -16(%ebp)
(dbx)
```

You can also enable trapping without recompiling the main program or using `dbx` by establishing an *initialization routine* that enables traps. (This might be useful, for example, if you want to abort the program when an exception occurs without running under a debugger.) There are two ways to establish such a routine.

If the object files and libraries that comprise the program are available, you can enable trapping by relinking the program with an appropriate initialization routine. First, create a C source file similar to the following:

```
#include <ieeefp.h>

#pragma init (trapinvalid)

void trapinvalid()
{
     /* FP_X_INV et al are defined in ieeefp.h */
     fpsetmask(FP_X_INV);
}
```

Now compile this file to create an object file and link the original program with this object file:

```
example% cc -c init.c
example% cc ex.o init.o -lm
example% a.out
Arithmetic Exception
```

If relinking is not possible but the program has been dynamically linked, you can enable trapping by using the shared object preloading facility of the runtime linker. To do this on SPARC systems, create the same C source file as above, but compile as follows:

```
example% cc -Kpic -G -ztext init.c -o init.so -lc
```

Now to enable trapping, add the path name of the `init.so` object to the list of preloaded shared objects specified by the environment variable `LD_PRELOAD`, for example:

```
example% env LD_PRELOAD=./init.so a.out
Arithmetic Exception
```

(See the *Linker and Libraries Guide* for more information about creating and preloading shared objects.)

In principle, you can change the way any floating point control modes are initialized by preloading a shared object as described above. Note, though, that initialization routines in shared objects, whether preloaded or explicitly linked, are executed by the runtime linker before it passes control to the startup code that is part of the main executable. The startup code then establishes any nondefault modes selected via the `-ftrap`, `-fround`, `-fns` (SPARC), or `-fprecision` (x86) compiler flags, executes any initialization routines that are part of the main executable (including those that are statically linked), and finally passes control to the main program. Therefore, on SPARC (i) any floating point control modes established by initialization routines in shared objects, such as the traps enabled in the example above, will remain in effect throughout the execution of the program unless they are overridden; (ii) any *nondefault* modes selected via the compiler flags will override modes established by initialization routines in shared objects (but default modes selected via compiler flags will not override previously established modes); and (iii) any modes established either by initialization routines that are part of the main executable or by the main program itself will override both.

On x86, the situation is complicated by the fact that the system kernel initializes the floating point hardware with some nondefault modes whenever a new process is begun, but the startup code automatically supplied by the compiler resets some of those modes to the default before passing control to the main program. Therefore, initialization routines in shared objects, unless they change the floating point control modes, run with trapping enabled for invalid operation, division by zero, and overflow exceptions and with the rounding precision set to round to 53 significant bits. Once the runtime linker passes control to the startup code, this code calls the routine `__fpstart` (found in the standard C library, `libc`), which disables all traps and sets the rounding precision to 64 significant bits. The startup code then establishes any nondefault modes selected by the `-fround`, `-ftrap`, or `-fprecision` flags before executing any statically linked initialization routines and passing control to the main program. As a consequence, in order to enable trapping (or change the rounding precision mode) on x86 platforms by preloading a shared object with an initialization routine, you must override the `__fpstart` routine so that it does not reset the trap enable and rounding precision modes. The substitute `__fpstart` routine should still perform the rest of the initialization functions that the standard routine does; however, the following code shows one way to do this.

```
#include <ieeefp.h>
#include <sunmath.h>
#include <sys/sysi86.h>

#pragma init (trapinvalid)

void trapinvalid()
{
     /* FP_X_INV et al are defined in ieeefp.h */
     fpsetmask(FP_X_INV);
}

extern int  __fltrounds(), __flt_rounds;
extern long _fp_hw;

void __fpstart()
{
    char *out;

    /* perform the same floating point initializations as
       the standard __fpstart() function defined by the
       System V ABI Intel processor supplement _but_ leave
       all trapping modes as is */
    __flt_rounds = __fltrounds();
    sysi86(SI86FPHW, &_fp_hw);
    _fp_hw &= 0xff;
    ieee_flags("set", "precision", "extended", &out);
}
```

Note that the `__fpstart` routine above calls `ieee_flags`, which is contained in `libsunmath`. Thus, in order to create a shared object from this code, you must use the `-R` and `-L` options to specify the location of the shared object `libsunmath.so`. This library is located in the Forte Developer product installation area. Assuming you have installed Forte Developer in the default location, the steps for compiling the code above to create a shared object and preloading this shared object to enable trapping are as follows.

```
 example% cc -Kpic -G -ztext init.c -o init.so -R/opt/SUNWspro/lib
-L/opt/SUNWspro/lib -lsunmath -lc
example% env LD_PRELOAD=./init.so a.out
Arithmetic Exception
```

Using a Signal Handler to Locate an Exception

The previous section presented several methods for enabling trapping at the outset of a program in order to locate the first occurrence of an exception. In contrast, you can isolate any particular occurrence of an exception by enabling trapping within the program itself. If you enable trapping but do not install a `SIGFPE` handler, the program will abort on the next occurrence of the trapped exception. Alternatively, if you install a `SIGFPE` handler, the next occurrence of the trapped exception will cause the system to transfer control to the handler, which can then print diagnostic information, such as the address of the instruction where the exception occurred, and either abort or resume execution. (In order to resume execution with any prospect for a meaningful outcome, the handler might need to supply a result for the exceptional operation as described in the next section.)

You can use `ieee_handler` to simultaneously enable trapping on any of the five IEEE floating point exceptions and either request that the program abort when the specified exception occurs or establish a `SIGFPE` handler. You can also install a `SIGFPE` handler using one of the lower-level functions `sigfpe`(3), `signal`(3c), or `sigaction`(2); however, these functions do not enable trapping as `ieee_handler` does. (Remember that a floating point exception triggers a `SIGFPE` signal only when its trap is enabled.)

`ieee_handler` (3m)

The syntax of a call to `ieee_handler` is:

`i = ieee_handler`(*action*, *exception*, *handler*)

The two input parameters *action* and *exception* are strings. The third input parameter, *handler*, is of type `sigfpe_handler_type`, which is defined in `floatingpoint.h`.

The three input parameters can take the following values:

Input Parameter	C or C++ Type	Possible Value
action	`char *`	`get`, `set`, `clear`
exception	`char *`	`invalid`, `division`, `overflow`, `underflow`, `inexact`, `all`, `common`

handler	`sigfpe_handler_type`	user-defined routine
		`SIGFPE_DEFAULT`
		`SIGFPE_IGNORE`
		`SIGFPE_ABORT`

When the requested action is `"set"`, `ieee_handler` establishes the handling function specified by *handler* for the exceptions named by *exception*. The handling function can be `SIGFPE_DEFAULT` or `SIGFPE_IGNORE`, both of which select the default IEEE behavior, `SIGFPE_ABORT`, which causes the program to abort on the occurrence of any of the named exceptions, or the address of a user-supplied subroutine, which causes that subroutine to be invoked (with the parameters described in the `sigaction`(2) manual page for a signal handler installed with the `SA_SIGINFO` flag set) when any of the named exceptions occurs. If the handler is `SIGFPE_DEFAULT` or `SIGFPE_IGNORE`, `ieee_handler` also disables trapping on the specified exceptions; for any other handler, `ieee_handler` enables trapping. (On x86 platforms, the floating point hardware traps whenever an exception's trap is enabled and its corresponding flag is raised. Therefore, to avoid spurious traps, a program should clear the flag for each specified *exception* before calling `ieee_handler` to enable trapping.)

When the requested *action* is `"clear"`, `ieee_handler` revokes whatever handling function is currently installed for the specified *exception* and disables its trap. (This is the same as `"set"`ting `SIGFPE_DEFAULT`.) The third parameter is ignored when *action* is `"clear"`.

For both the `"set"` and `"clear"` actions, `ieee_handler` returns 0 if the requested action is available and a nonzero value otherwise.

When the requested *action* is `"get"`, `ieee_handler` returns the address of the handler currently installed for the specified *exception* (or `SIGFPE_DEFAULT`, if no handler is installed).

The following examples show a few code fragments illustrating the use of `ieee_handler`. This C code causes the program to abort on division by zero:

```
#include <sunmath.h>
/* uncomment the following line on x86 systems */
    /* ieee_flags("clear", "exception", "division", NULL); */
    if (ieee_handler("set", "division", SIGFPE_ABORT) != 0)
        printf("ieee trapping not supported here \n");
```

Here is the equivalent Fortran code:

```
#include <floatingpoint.h>
c uncomment the following line on x86 systems
c     ieee_flags('clear', 'exception', 'division', %val(0))
      i = ieee_handler('set', 'division', SIGFPE_ABORT)
      if(i.ne.0) print *,'ieee trapping not supported here'
```

This C fragment restores IEEE default exception handling for all exceptions:

```
#include <sunmath.h>
    if (ieee_handler("clear", "all", 0) != 0)
        printf("could not clear exception handlers\n");
```

Here is the same action in Fortran:

```
      i = ieee_handler('clear', 'all', 0)
      if (i.ne.0) print *, 'could not clear exception handlers'
```

Reporting an Exception From a Signal Handler

When a `SIGFPE` handler installed via `ieee_handler` is invoked, the operating system provides additional information indicating the type of exception that occurred, the address of the instruction that caused it, and the contents of the machine's integer and floating point registers. The handler can examine this information and print a message identifying the exception and the location at which it occurred.

To access the information supplied by the system, declare the handler as follows. (The remainder of this chapter presents sample code in C; see Appendix A for examples of SIGFPE handlers in Fortran.)

```
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    ...
}
```

When the handler is invoked, the *sig* parameter contains the number of the signal that was sent. Signal numbers are defined in sys/signal.h; the SIGFPE signal number is 8.

The *sip* parameter points to a structure that records additional information about the signal. For a SIGFPE signal, the relevant members of this structure are sip->si_code and sip->si_addr (see sys/siginfo.h). The significance of these members depends on the system and on what event triggered the SIGFPE signal.

The sip->si_code member is one of the SIGFPE signal types listed in TABLE 4-4. (The tokens shown are defined in sys/machsig.h.)

TABLE 4-4 Types for Arithmetic Exceptions

SIGFPE Type	IEEE Type
FPE_INTDIV	
FPE_INTOVF	
FPE_FLTRES	inexact
FPE_FLTDIV	division
FPE_FLTUND	underflow
FPE_FLTINV	invalid
FPE_FLTOVF	overflow

As the table shows, each type of IEEE floating point exception has a corresponding SIGFPE signal type. Integer division by zero (FPE_INTDIV) and integer overflow (FPE_INTOVF) are also included among the SIGFPE types, but because they are not IEEE floating point exceptions you cannot install handlers for them via ieee_handler. (You can install handlers for these SIGFPE types via sigfpe(3); note, though, that integer overflow is ignored by default on all SPARC and x86 platforms. Special instructions can cause the delivery of a SIGFPE signal of type FPE_INTOVF, but Sun compilers do not generate these instructions.)

For a SIGFPE signal corresponding to an IEEE floating point exception, the sip->si_code member indicates which exception occurred on SPARC systems, while on x86 platforms it indicates the highest priority exception whose flag is raised (excluding the denormal operand flag). The sip->si_addr member holds the address of the instruction that caused the exception on SPARC systems, and on x86 platforms it holds the address of the instruction at which the trap was taken (usually the next floating point instruction following the one that caused the exception).

Finally, the *uap* parameter points to a structure that records the state of the system at the time the trap was taken. The contents of this structure are system-dependent; see sys/reg.h for definitions of some of its members.

Using the information provided by the operating system, we can write a SIGFPE handler that reports the type of exception that occurred and the address of the instruction that caused it. CODE EXAMPLE 4-1 shows such a handler.

CODE EXAMPLE 4-1 SIGFPE Handler

```
#include <stdio.h>
#include <sys/ieeefp.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    code, addr;

#ifdef i386
    unsigned    sw;

    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status &
       ~uap->uc_mcontext.fpregs.fp_reg_set.fpship_state.state[0];
    if (sw & (1 << fp_invalid))
        code = FPE_FLTINV;
    else if (sw & (1 << fp_division))
        code = FPE_FLTDIV;
    else if (sw & (1 << fp_overflow))
        code = FPE_FLTOVF;
    else if (sw & (1 << fp_underflow))
        code = FPE_FLTUND;
    else if (sw & (1 << fp_inexact))
        code = FPE_FLTRES;
    else
        code = 0;
```

CODE EXAMPLE 4-1 `SIGFPE` Handler *(Continued)*

```
    addr = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.
       state[3];
#else
    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
#endif
    fprintf(stderr, "fp exception %x at address %x\n", code,
        addr);
}
int main()
{
    double  x;

    /* trap on common floating point exceptions */
    if (ieee_handler("set", "common", handler) != 0)
        printf("Did not set exception handler\n");
    /* cause an underflow exception (will not be reported) */
    x = min_normal();
    printf("min_normal = %g\n", x);
    x = x / 13.0;
    printf("min_normal / 13.0 = %g\n", x);

    /* cause an overflow exception (will be reported) */
    x = max_normal();
    printf("max_normal = %g\n", x);
    x = x * x;
    printf("max_normal * max_normal = %g\n", x);
    ieee_retrospective(stderr);
    return 0;
}
```

On SPARC systems, the output from this program resembles the following:

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
 Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

On x86 platforms, the operating system saves a copy of the accrued exception flags and then clears them before invoking a SIGFPE handler. Unless the handler takes steps to preserve them, the accrued flags are lost once the handler returns. Thus, the output from the preceding program does not indicate that an underflow exception was raised.

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 8048fe6
max_normal * max_normal = 1.79769e+308
 Note: IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_handler(3M)
```

In most cases, the instruction that causes the exception does not deliver the IEEE default result when trapping is enabled: in the preceding outputs, the value reported for max_normal * max_normal is not the default result for an operation that overflows (i.e., a correctly signed infinity). In general, a SIGFPE handler must supply a result for an operation that causes a trapped exception in order to continue the computation with meaningful values. See "Handling Exceptions" on page 82 for one way to do this.

Using libm9x.so Exception Handling Extensions to Locate an Exception

C/C++ programs can use the exception handling extensions to the C99 floating point environment functions in libm9x.so to locate exceptions in several ways. These extensions include functions that can establish handlers and simultaneously enable traps, just as ieee_handler does, but they provide more flexibility. They also support logging of retrospective diagnostic messages regarding floating point exceptions to a selected file.

fex_set_handling(3m)

The fex_set_handling function allows you to select one of several options, or modes, for handling each type of floating point exception. The syntax of a call to fex_set_handling is:

```
ret = fex_set_handling(ex, mode, handler);
```

The *ex* argument specifies the set of exceptions to which the call applies. It must be a bitwise "or" of the values listed in the first column of TABLE 4-5. (These values are defined in `fenv.h`.)

TABLE 4-5 Exception Codes for `fex_set_handling`

Value	Exception
`FEX_INEXACT`	inexact result
`FEX_UNDERFLOW`	underflow
`FEX_OVERFLOW`	overflow
`FEX_DIVBYZERO`	division by zero
`FEX_INV_ZDZ`	0/0 invalid operation
`FEX_INV_IDI`	infinity/infinity invalid operation
`FEX_INV_ISI`	infinity-infinity invalid operation
`FEX_INV_ZMI`	0*infinity invalid operation
`FEX_INV_SQRT`	square root of negative number
`FEX_INV_SNAN`	operation on signaling NaN
`FEX_INV_INT`	invalid integer conversion
`FEX_INV_CMP`	invalid unordered comparison

For convenience, `fenv.h` also defines the following values: `FEX_NONE` (no exceptions), `FEX_INVALID` (all invalid operation exceptions), `FEX_COMMON` (overflow, division by zero, and all invalid operations), and `FEX_ALL` (all exceptions).

The *mode* argument specifies the exception handling mode to be established for the indicated exceptions. There are five possible modes:

- `FEX_NONSTOP` mode provides the IEEE 754 default nonstop behavior. This is equivalent to leaving the exception's trap disabled. (Note that unlike `ieee_handler`, `fex_set_handling` allows you to establish nondefault handling for certain types of invalid operation exceptions and retain IEEE default handling for the rest.)
- `FEX_NOHANDLER` mode is equivalent to enabling the exception's trap without providing a handler. When an exception occurs, the system transfers control to a previously installed `SIGFPE` handler, if present, or aborts.
- `FEX_ABORT` mode causes the program to call `abort`(3c) when the exception occurs.
- `FEX_SIGNAL` installs the handling function specified by the *handler* argument for the indicated exceptions. When any of these exceptions occurs, the handler is invoked with the same arguments as if it had been installed by `ieee_handler`.

- FEX_CUSTOM installs the handling function specified by *handler* for the indicated exceptions. Unlike FEX_SIGNAL mode, when an exception occurs, the handler is invoked with a simplified argument list. The arguments consist of an integer whose value is one of the values listed in TABLE 4-5 and a pointer to a structure that records additional information about the operation that caused the exception. The contents of this structure are described in the next section and in the fex_set_handling(3m) manual page.

Note that the *handler* parameter is ignored if the specified *mode* is FEX_NONSTOP, FEX_NOHANDLER, or FEX_ABORT. fex_set_handling returns a nonzero value if the specified mode is established for the indicated exceptions, and returns zero otherwise. (In the examples below, the return value is ignored.)

The following examples suggest ways to use fex_set_handling to locate certain types of exceptions. To abort on a 0/0 exception:

```
fex_set_handling(FEX_INV_ZDZ, FEX_ABORT, NULL);
```

To install a SIGFPE handler for overflow and division by zero:

```
fex_set_handling(FEX_OVERFLOW | FEX_DIVBYZERO, FEX_SIGNAL,
    handler);
```

In the previous example, the handler function could print the diagnostic information supplied via the *sip* parameter to a SIGFPE handler, as shown in the previous subsection. By contrast, the following example prints the information about the exception that is supplied to a handler installed in FEX_CUSTOM mode. (See the fex_set_handling(3m) manual page for more information.)

CODE EXAMPLE 4-2 Printing Information Supplied to Handler Installed in FEX_CUSTOM Mode

```
#include <fenv.h>

void handler(int ex, fex_info_t *info)
{
    switch (ex) {
    case FEX_OVERFLOW:
        printf("Overflow in ");
        break;
    case FEX_DIVBYZERO:
        printf("Division by zero in ");
        break;
```

CODE EXAMPLE 4-2 Printing Information Supplied to Handler Installed in `FEX_CUSTOM` Mode *(Continued)*

```
    default:
        printf("Invalid operation in ");
    }
    switch (info->op) {
    case fex_add:
        printf("floating point add\n");
        break;
    case fex_sub:
        printf("floating point subtract\n");
        break;
    case fex_mul:
        printf("floating point multiply\n");
        break;
    case fex_div:
        printf("floating point divide\n");
        break;
    case fex_sqrt:
        printf("floating point square root\n");
        break;
    case fex_cnvt:
        printf("floating point conversion\n");
        break;
    case fex_cmp:
        printf("floating point compare\n");
        break;
    default:
        printf("unknown operation\n");
    }
    switch (info->op1.type) {
    case fex_int:
        printf("operand 1: %d\n", info->op1.val.i);
        break;
    case fex_llong:
        printf("operand 1: %lld\n", info->op1.val.l);
        break;
    case fex_float:
        printf("operand 1: %g\n", info->op1.val.f);
        break;
    case fex_double:
        printf("operand 1: %g\n", info->op1.val.d);
        break;
```

CODE EXAMPLE 4-2 Printing Information Supplied to Handler Installed in `FEX_CUSTOM` Mode *(Continued)*

```
    case fex_ldouble:
        printf("operand 1: %Lg\n", info->op1.val.q);
        break;
    }
    switch (info->op2.type) {
    case fex_int:
        printf("operand 2: %d\n", info->op2.val.i);
        break;
    case fex_llong:
        printf("operand 2: %lld\n", info->op2.val.l);
        break;
    case fex_float:
        printf("operand 2: %g\n", info->op2.val.f);
        break;
    case fex_double:
        printf("operand 2: %g\n", info->op2.val.d);
        break;
    case fex_ldouble:
        printf("operand 2: %Lg\n", info->op2.val.q);
        break;
    }
}
...
fex_set_handling(FEX_COMMON, FEX_CUSTOM, handler);
```

The handler in the preceding example reports the type of exception that occurred, the type of operation that caused it, and the operands. It does not indicate where the exception occurred. To find out where the exception occurred, you can use retrospective diagnostics.

Retrospective Diagnostics

Another way to locate an exception using the `libm9x.so` exception handling extensions is to enable logging of retrospective diagnostic messages regarding floating point exceptions. When you enable logging of retrospective diagnostics, the system records information about certain exceptions. This information includes the type of exception, the address of the instruction that caused it, the manner in which it will be handled, and a stack trace similar to that produced by a debugger. (The stack trace recorded with a retrospective diagnostic message contains only instruction addresses and function names; for additional debugging information such as line numbers, source file names, and argument values, you must use a debugger.)

The log of retrospective diagnostics does not contain information about every single exception that occurs; if it did, a typical log would be huge, and it would be impossible to isolate unusual exceptions. Instead, the logging mechanism eliminates redundant messages. A message is considered redundant under either of two circumstances:

- The same exception has been previously logged at the same location (i.e., with the same instruction address and stack trace), or
- FEX_NONSTOP mode is in effect for the exception and its flag has been previously raised.

In particular, in most programs, only the first occurrence of each type of exception will be logged. (When FEX_NONSTOP handling mode is in effect for an exception, clearing its flag via any of the C99 floating point environment functions allows the next occurrence of that exception to be logged, provided it does not occur at a location at which it was previously logged.)

To enable logging, use the fex_set_log function to specify the file to which messages should be delivered. For example, to log messages to the standard error file, use:

```
fex_set_log(stderr);
```

CODE EXAMPLE 4-3 combines logging of retrospective diagnostics with the shared object preloading facility illustrated in the previous section. By creating the following C source file, compiling it to a shared object, preloading the shared object by supplying its path name in the LD_PRELOAD environment variable, and specifying the names of one or more exceptions (separated by commas) in the FTRAP environment variable, you can simultaneously abort the program on the specified exceptions and obtain retrospective diagnostic output showing where each exception occurs.

CODE EXAMPLE 4-3 Combined Logging of Retrospective Diagnostics With Shared Object Preloading

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fenv.h>

static struct ftrap_string {
    const char  *name;
    int         value;
} ftrap_table[] = {
    { "inexact", FEX_INEXACT },
    { "division", FEX_DIVBYZERO },
```

CODE EXAMPLE 4-3 Combined Logging of Retrospective Diagnostics With Shared Object Preloading *(Continued)*

```
    { "underflow", FEX_UNDERFLOW },
    { "overflow", FEX_OVERFLOW },
    { "invalid", FEX_INVALID },
    { NULL, 0 }
};

#pragma init (set_ftrap)
void set_ftrap()
{
    struct ftrap_string  *f;
    char                 *s, *s0;
    int                  ex = 0;

    if ((s = getenv("FTRAP")) == NULL)
        return;

    if ((s0 = strtok(s, ",")) == NULL)
        return;

    do {
        for (f = &trap_table[0]; f->name != NULL; f++) {
            if (!strcmp(s0, f->name))
                ex |= f->value;
        }
    } while ((s0 = strtok(NULL, ",")) != NULL);

    fex_set_handling(ex, FEX_ABORT, NULL);
    fex_set_log(stderr);
}
```

Using the preceding code with the example program given at the beginning of this section produces the following results on SPARC:

```
example% cc -Kpic -G -ztext init.c -o init.so -R/opt/SUNWspro/lib
-L/opt/SUNWspro/lib -lm9x -lc
example% env FTRAP=invalid LD_PRELOAD=./init.so a.out
Floating point invalid operation (sqrt) at 0x00010c24 sqrtm1_, abort
  0x00010c30  sqrtm1_
  0x00010b48  MAIN_
  0x00010ccc  main
Abort
```

The preceding output shows that the invalid operation exception was raised as a result of a square root operation in the routine `sqrtm1`.

(As noted above, to enable trapping from an initialization routine in a shared object on x86 platforms, you must override the standard `__fpstart` routine.)

Appendix A gives more examples showing typical log outputs. For general information, see the `fex_set_log`(3m) man page.

Handling Exceptions

Historically, most numerical software has been written without regard to exceptions (for a variety of reasons), and many programmers have become accustomed to environments in which exceptions cause a program to abort immediately. Now, some high-quality software packages such as LAPACK are being carefully designed to avoid exceptions such as division by zero and invalid operations and to scale their inputs aggressively to preclude overflow and potentially harmful underflow. Neither of these approaches to dealing with exceptions is appropriate in every situation. However, ignoring exceptions can pose problems when one person writes a program or subroutine that is intended to be used by someone else (perhaps someone who does not have access to the source code), and attempting to avoid all exceptions can require many defensive tests and branches and carry a significant cost (see Demmel and Li, "Faster Numerical Algorithms via Exception Handling," *IEEE Trans. Comput.* 43 (1994), pp. 983–992.)

The default exception response, status flags, and optional trapping facility of IEEE arithmetic are intended to provide a third alternative: continuing a computation in the presence of exceptions and either detecting them after the fact or intercepting and handling them as they occur. As described above, `ieee_flags` or the C99 floating point environment functions can be used to detect exceptions after the fact, and `ieee_handler` or `fex_set_handling` can be used to enable trapping and install a handler to intercept exceptions as they occur. In order to continue the computation, however, the IEEE standard recommends that a trap handler be able to provide a result for the operation that incurred an exception. A `SIGFPE` handler installed via `ieee_handler` or `fex_set_handling` in `FEX_SIGNAL` mode can accomplish this using the *uap* parameter supplied to a signal handler by the Solaris operating environment. An `FEX_CUSTOM` mode handler installed via `fex_set_handling` can provide a result using the *info* parameter supplied to such a handler.

Recall that a SIGFPE signal handler can be declared in C as follows:

```
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    ...
}
```

When a SIGFPE signal handler is invoked as a result of a trapped floating point exception, the *uap* parameter points to a data structure that contains a copy of the machine's integer and floating point registers as well as other system-dependent information describing the exception. If the signal handler returns normally, the saved data are restored and the program resumes execution at the point at which the trap was taken. Thus, by accessing and decoding the information in the data structure that describes the exception and possibly modifying the saved data, a SIGFPE handler can substitute a user-supplied value for the result of an exceptional operation and continue computation.

An FEX_CUSTOM mode handler can be declared as follows:

```
#include <fenv.h>

void handler(int ex, fex_info_t *info)
{
    ...
}
```

When a FEX_CUSTOM handler is invoked, the *ex* parameter indicates which type of exception occurred (it is one of the values listed in TABLE 4-5) and the *info* parameter points to a data structure that contains more information about the exception. Specifically, this structure contains a code representing the arithmetic operation that caused the exception and structures recording the operands, if they are available. It also contains a structure recording the default result that would have been substituted if the exception were not trapped and an integer value holding the bitwise "or" of the exception flags that would have accrued. The handler can modify the latter members of the structure to substitute a different result or change the set of flags that are accrued. (Note that if the handler returns without modifying these data, the program will continue with the default untrapped result and flags just as if the exception were not trapped.)

As an illustration, the following section shows how to substitute a scaled result for an operation that underflows or overflows. See Appendix A for further examples.

Substituting IEEE Trapped Under/Overflow Results

The IEEE standard recommends that when underflow and overflow are trapped, the system should provide a way for a trap handler to substitute an *exponent-wrapped* result, i.e., a value that agrees with what would have been the rounded result of the operation that underflowed or overflowed except that the exponent is wrapped around the end of its usual range, thereby effectively scaling the result by a power of two. The scale factor is chosen to map underflowed and overflowed results as nearly as possible to the middle of the exponent range so that subsequent computations will be less likely to underflow or overflow further. By keeping track of the number of underflows and overflows that occur, a program can scale the final result to compensate for the exponent wrapping. This under/overflow "counting mode" can be used to produce accurate results in computations that would otherwise exceed the range of the available floating point formats. (See P. Sterbenz, *Floating-Point Computation*.)

On SPARC, when a floating point instruction incurs a trapped exception, the system leaves the destination register unchanged. Thus, in order to substitute the exponent-wrapped result, an under/overflow handler must decode the instruction, examine the operand registers, and generate the scaled result itself. CODE EXAMPLE 4-4 shows a handler that performs these steps. (In order to use this handler with code compiled for UltraSPARC systems, compile the handler on a system running Solaris 2.6, Solaris 7, or Solaris 8 and define the preprocessor token `V8PLUS`.)

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems

```
#include <stdio.h>
#include <ieeefp.h>
#include <math.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

#ifdef V8PLUS
/* The upper 32 floating point registers are stored in an area
   pointed to by uap->uc_mcontext.xrs.xrs_prt. Note that this
   pointer is valid ONLY when uap->uc_mcontext.xrs.xrs_id ==
   XRS_ID (defined in sys/procfs.h). */
#include <assert.h>
#include <sys/procfs.h>
#define FPxreg(x)  ((prxregset_t*)uap->uc_mcontext.xrs.xrs_ptr)
->pr_un.pr_v8p.pr_xfr.pr_regs[(x)]
#endif
```

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems *(Continued)*

```
#define FPreg(x)   uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[(x)]

/*
*  Supply the IEEE 754 default result for trapped under/overflow
*/
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    instr, opf, rs1, rs2, rd;
    long double qs1, qs2, qd, qscl;
    double      ds1, ds2, dd, dscl;
    float       fs1, fs2, fd, fscl;

    /* get the instruction that caused the exception */
    instr = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

    /* extract the opcode and source and destination register
       numbers */
    opf = (instr >> 5) & 0x1ff;
    rs1 = (instr >> 14) & 0x1f;
    rs2 = instr & 0x1f;
    rd = (instr >> 25) & 0x1f;
    /* get the operands */
    switch (opf & 3) {
    case 1: /* single precision */
        fs1 = *(float*)&FPreg(rs1);
        fs2 = *(float*)&FPreg(rs2);
        break;

    case 2: /* double precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds1 = *(double*)&FPxreg(rs1 & 0x1e);
        }
        else
            ds1 = *(double*)&FPreg(rs1);
        if (rs2 & 1)
```

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems *(Continued)*

```
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds2 = *(double*)&FPxreg(rs2 & 0x1e);
        }
        else
            ds2 = *(double*)&FPreg(rs2);
#else
        ds1 = *(double*)&FPreg(rs1);
        ds2 = *(double*)&FPreg(rs2);
#endif
        break;

    case 3: /* quad precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            qs1 = *(long double*)&FPxreg(rs1 & 0x1e);
        }
        else
            qs1 = *(long double*)&FPreg(rs1);
        if (rs2 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            qs2 = *(long double*)&FPxreg(rs2 & 0x1e);
        }
        else
            qs2 = *(long double*)&FPreg(rs2);
#else
        qs1 = *(long double*)&FPreg(rs1);
        qs2 = *(long double*)&FPreg(rs2);
#endif
        break;
    }

    /* set up scale factors */
    if (sip->si_code == FPE_FLTOVF) {
        fscl = scalbnf(1.0f, -96);
        dscl = scalbn(1.0, -768);
        qscl = scalbnl(1.0, -12288);
```

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems *(Continued)*

```
    } else {
        fscl = scalbnf(1.0f, 96);
        dscl = scalbn(1.0, 768);
        qscl = scalbnl(1.0, 12288);
    }

    /* disable traps and generate the scaled result */
    fpsetmask(0);
    switch (opf) {
    case 0x41: /* add single */
        fd = fscl * (fscl * fs1 + fscl * fs2);
        break;

    case 0x42: /* add double */
        dd = dscl * (dscl * ds1 + dscl * ds2);
        break;

    case 0x43: /* add quad */
        qd = qscl * (qscl * qs1 + qscl * qs2);
        break;
    case 0x45: /* subtract single */
        fd = fscl * (fscl * fs1 - fscl * fs2);
        break;

    case 0x46: /* subtract double */
        dd = dscl * (dscl * ds1 - dscl * ds2);
        break;

    case 0x47: /* subtract quad */
        qd = qscl * (qscl * qs1 - qscl * qs2);
        break;

    case 0x49: /* multiply single */
        fd = (fscl * fs1) * (fscl * fs2);
        break;

    case 0x4a: /* multiply double */
        dd = (dscl * ds1) * (dscl * ds2);
        break;
```

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems *(Continued)*

```
    case 0x4b: /* multiply quad */
        qd = (qscl * qs1) * (qscl * qs2);
        break;

    case 0x4d: /* divide single */
        fd = (fscl * fs1) / (fs2 / fscl);
        break;

    case 0x4e: /* divide double */
        dd = (dscl * ds1) / (ds2 / dscl);
        break;

    case 0x4f: /* divide quad */
        qd = (qscl * qs1) / (qs2 / dscl);
        break;

    case 0xc6: /* convert double to single */
        fd = (float) (fscl * (fscl * ds1));
        break;
    case 0xc7: /* convert quad to single */
        fd = (float) (fscl * (fscl * qs1));
        break;

    case 0xcb: /* convert quad to double */
        dd = (double) (dscl * (dscl * qs1));
        break;
    }

    /* store the result in the destination */
    if (opf & 0x80) {
        /* conversion operation */
        if (opf == 0xcb) {
            /* convert quad to double */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(double*)&FPxreg(rd & 0x1e) = dd;
            }
```

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems *(Continued)*

```
            else
                *(double*)&FPreg(rd) = dd;
#else
            *(double*)&FPreg(rd) = dd;
#endif
        } else
            /* convert quad/double to single */
            *(float*)&FPreg(rd) = fd;
    } else {
        /* arithmetic operation */
        switch (opf & 3) {
        case 1: /* single precision */
            *(float*)&FPreg(rd) = fd;
            break;
        case 2: /* double precision */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(double*)&FPxreg(rd & 0x1e) = dd;
            }
            else
                *(double*)&FPreg(rd) = dd;
#else
            *(double*)&FPreg(rd) = dd;
#endif
            break;

        case 3: /* quad precision */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(long double*)&FPxreg(rd & 0x1e) = qd;
            }
            else
                *(long double*)&FPreg(rd & 0x1e) = qd;
```

CODE EXAMPLE 4-4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC Systems *(Continued)*

```
#else
            *(long double*)&FPreg(rd & 0x1e) = qd;
#endif
            break;
        }
    }
}

int
main()
{
    volatile float   a, b;
    volatile double  x, y;

    ieee_handler("set", "underflow", ieee_trapped_default);
    ieee_handler("set", "overflow", ieee_trapped_default);
    a = b = 1.0e30f;
    a *= b; /* overflow; will be wrapped to a moderate number */
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );
    a /= b; /* underflow; will wrap back */
    printf( "%g\n", a );

    x = y = 1.0e300;
    x *= y; /* overflow; will be wrapped to a moderate number */
    printf( "%g\n", x );
    x /= y;
    printf( "%g\n", x );
    x /= y; /* underflow; will wrap back */
    printf( "%g\n", x );

    ieee_retrospective(stdout);
    return 0;
}
```

In this example, the variables `a`, `b`, `x`, and `y` have been declared `volatile` only to prevent the compiler from evaluating `a * b`, etc., at compile time. In typical usage, the `volatile` declarations would not be needed.

The output from the preceding program is:

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note: IEEE floating-point exception traps enabled:
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

On x86, the floating point hardware provides the exponent-wrapped result when a floating point instruction incurs a trapped underflow or overflow and its destination is a register. When trapped underflow or overflow occurs on a floating point store instruction, however, the hardware traps without completing the store (and without popping the stack, if the store instruction is a store-and-pop). Thus, in order to implement counting mode, an under/overflow handler must generate the scaled result and fix up the stack when a trap occurs on a store instruction. CODE EXAMPLE 4-5 illustrates such a handler.

CODE EXAMPLE 4-5 Substituting IEEE Trapped Under/Overflow Handler Results for x86 Systems

```
#include <stdio.h>
#include <ieeefp.h>
#include <math.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

/* offsets into the saved fp environment */
#define CW    0    /* control word */
#define SW    1    /* status word */
#define TW    2    /* tag word */
#define OP    4    /* opcode */
#define EA    5    /* operand address */

#define FPenv(x)    uap->uc_mcontext.fpregs.fp_reg_set.
fpchip_state.state[(x)]

#define FPreg(x)    *(long double *)(10*(x)+(char*)&uap->
uc_mcontext.fpregs.fp_reg_set.fpchip_state.state[7])
/*
*  Supply the IEEE 754 default result for trapped under/overflow
```

CODE EXAMPLE 4-5 Substituting IEEE Trapped Under/Overflow Handler Results for x86 Systems *(Continued)*

```
*/
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    double      dscl;
    float       fscl;
    unsigned    sw, op, top;
    int         mask, e;

    /* preserve flags for untrapped exceptions */
    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status;
    FPenv(SW) |= (sw & (FPenv(CW) & 0x3f));
    /* if the excepting instruction is a store, scale the stack
       top, store it, and pop the stack if need be */
    fpsetmask(0);
    op = FPenv(OP) >> 16;
    switch (op & 0x7f8) {
    case 0x110:
    case 0x118:
    case 0x150:
    case 0x158:
    case 0x190:
    case 0x198:
        fscl = scalbnf(1.0f, (sip->si_code == FPE_FLTOVF)?
            -96 : 96);
        *(float *)FPenv(EA) = (FPreg(0) * fscl) * fscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
        break;

    case 0x510:
    case 0x518:
```

CODE EXAMPLE 4-5 Substituting IEEE Trapped Under/Overflow Handler Results for x86 Systems *(Continued)*

```
    case 0x550:
    case 0x558:
    case 0x590:
    case 0x598:
        dscl = scalbn(1.0, (sip->si_code == FPE_FLTOVF)?
            -768 : 768);
        *(double *)FPenv(EA) = (FPreg(0) * dscl) * dscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
        break;
    }
}

int main()
{
    volatile float    a, b;
    volatile double    x, y;

    ieee_handler("set", "underflow", ieee_trapped_default);
    ieee_handler("set", "overflow", ieee_trapped_default);
    a = b = 1.0e30f;
    a *= b;
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );
    x = y = 1.0e300;
    x *= y;
    printf( "%g\n", x );
```

CODE EXAMPLE 4-5 Substituting IEEE Trapped Under/Overflow Handler Results for x86 Systems *(Continued)*

```
    x /= y;
    printf( "%g\n", x );
    x /= y;
    printf( "%g\n", x );

    ieee_retrospective(stdout);
    return 0;
}
```

As on SPARC, the output from the preceding program on x86 is:

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note: IEEE floating-point exception traps enabled:
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

C/C++ programs can use the `fex_set_handling` function in `libm9x.so` to install a `FEX_CUSTOM` handler for underflow and overflow. On SPARC systems, the information supplied to such a handler always includes the operation that caused the exception and the operands, and this information is sufficient to allow the handler to compute the IEEE exponent-wrapped result, as shown above. On x86, the available information might not always indicate which particular operation caused the exception; when the exception is raised by one of the transcendental instructions, for example, the `info->op` parameter is set to `fex_other`. (See the `fenv.h` file for definitions.) Moreover, the x86 hardware delivers an exponent-wrapped result automatically, and this can overwrite one of the operands if the destination of the excepting instruction is a floating point register.

Fortunately, the `fex_set_handling` feature provides a simple way for a handler installed in `FEX_CUSTOM` mode to substitute the IEEE exponent-wrapped result for an operation that underflows or overflows. When either of these exceptions is trapped, the handler can set

```
info->res.type = fex_nodata;
```

to indicate that the exponent-wrapped result should be delivered. Here is an example showing such a handler:

```
#include <stdio.h>
#include <fenv.h>

void handler(int ex, fex_info_t *info) {
    info->res.type = fex_nodata;
}
int main()
{
    volatile float  a, b;
    volatile double x, y;

    fex_set_log(stderr);
    fex_set_handling(FEX_UNDERFLOW | FEX_OVERFLOW, FEX_CUSTOM,
        handler);
    a = b = 1.0e30f;
    a *= b; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    a /= b; /* underflow; will wrap back */
    printf("%g\n", a);

    x = y = 1.0e300;
    x *= y; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", x);
    x /= y;

    printf("%g\n", x);
    x /= y; /* underflow; will wrap back */
    printf("%g\n", x);

    return 0;
}
```

The output from the preceding program resembles the following:

```
Floating point overflow at 0x00010924 main, handler: handler
  0x00010928 main
159.309
1.59309e-28
Floating point underflow at 0x00010994 main, handler: handler
  0x00010998 main
1
Floating point overflow at 0x000109e4 main, handler: handler
  0x000109e8 main
4.14884e+137
4.14884e-163
Floating point underflow at 0x00010a4c main, handler: handler
  0x00010a50 main
1
```

APPENDIX **A**

Examples

This appendix provides examples of how to accomplish some popular tasks. The examples are written either in Fortran or ANSI C, and many depend on the current versions of `libm` and `libsunmath`. These examples were tested with the current C and Fortran compilers in the Solaris operating environment.

IEEE Arithmetic

The following examples show one way you can examine the hexadecimal representations of floating-point numbers. Note that you can also use the debuggers to look at the hexadecimal representations of stored data.

The following C program prints a double precision approximation to π and single precision infinity:

CODE EXAMPLE A-1 Double Precision Example

```
#include <math.h>
#include <sunmath.h>

int main() {
    union {
        float       flt;
        unsigned un;
    } r;
    union {
        double      dbl;
        unsigned    un[2];
    } d;
```

CODE EXAMPLE A-1 Double Precision Example *(Continued)*

```
    /* double precision */
    d.dbl = M_PI;
    (void) printf("DP Approx pi = %08x %08x = %18.17e \n",
       d.un[0], d.un[1], d.dbl);

    /* single precision */
    r.flt = infinityf();
    (void) printf("Single Precision %8.7e : %08x \n",
       r.flt, r.un);

    return 0;
}
```

On SPARC, the output of the preceding program looks like this:

```
DP Approx pi = 400921fb 54442d18 = 3.14159265358979312e+00
Single Precision Infinity: 7f800000
```

The following Fortran program prints the smallest normal numbers in each format:

CODE EXAMPLE A-2 Print Smallest Normal Numbers in Each Format

```
    program print_ieee_values
c
c the purpose of the implicit statements is to ensure
c that the floatingpoint pseudo-intrinsic functions
c are declared with the correct type
c
    implicit real*16 (q)
    implicit double precision (d)
    implicit real (r)
    real*16           z
    double precision  x
    real              r
c
    z = q_min_normal()
    write(*,7) z, z
 7  format('min normal, quad: ',1pe47.37e4,/,' in hex ',z32.32)
c
    x = d_min_normal()
```

CODE EXAMPLE A-2 Print Smallest Normal Numbers in Each Format *(Continued)*

```
      write(*,14) x, x
   14 format('min normal, double: ',1pe23.16,' in hex ',z16.16)
c
      r = r_min_normal()
      write(*,27) r, r
   27 format('min normal, single: ',1pe14.7,' in hex ',z8.8)
c
      end
```

On SPARC, the corresponding output reads:

```
min normal, quad:   3.362103143112093506262677817321752602 6D-4932
 in hex 00010000000000000000000000000000
min normal, double:  2.2250738585072014-308 in hex 0010000000000000
min normal, single:  1.1754944E-38 in hex 00800000
```

The Math Libraries

This section shows examples that use functions from the math library.

Random Number Generator

The following example calls a random number generator to generate an array of numbers and uses a timing function to measure the time it takes to compute the EXP of the given numbers:

CODE EXAMPLE A-3 Random Number Generator

```
#ifdef DP
#define GENERIC double precision
#else
#define GENERIC real
#endif
#define SIZE 400000
```

CODE EXAMPLE A-3 Random Number Generator *(Continued)*

```
      program example
c
      implicit GENERIC (a-h,o-z)
      GENERIC x(SIZE), y, lb, ub
      real tarray(2), u1, u2
c
c compute EXP on random numbers in [-ln2/2,ln2/2]
      lb = -0.3465735903
      ub = 0.3465735903
c
c generate array of random numbers
#ifdef DP
      call d_init_addrans()
      call d_addrans(x,SIZE,lb,ub)
#else
      call r_init_addrans()
      call r_addrans(x,SIZE,lb,ub)
#endif
c
c start the clock
      call dtime(tarray)
      u1 = tarray(1)
c
c compute exponentials

      do 16 i=1,SIZE
      y = exp(x(i))
   16 continue
c
c get the elapsed time
      call dtime(tarray)
      u2 = tarray(1)
      print *,'time used by EXP is ',u2-u1
      print *,'last values for x and exp(x) are ',x(SIZE),y
c
      call flush(6)
      end
```

To compile the preceding example, place the source code in a file with the suffix `F` (not `f`) so that the compiler will automatically invoke the preprocessor, and specify either `-DSP` or `-DDP` on the command line to select single or double precision.

This example shows how to use d_addrans to generate blocks of random data uniformly distributed over a user-specified range:

CODE EXAMPLE A-4 Using d_addrans

```
/*
 * test SIZE*LOOPS random arguments to sin in the range
 * [0, threshold] where
 * threshold = 3E30000000000000 (3.72529029846191406e-09)
 */

#include <math.h>
#include <sunmath.h>

#define SIZE 10000
#define LOOPS 100
int main()
{
    doublex[SIZE], y[SIZE];
    int i, j, n;
    doublelb, ub;
    union {
    unsigned   u[2];
    double     d;
    }  upperbound;

    upperbound.u[0] = 0x3e300000;
    upperbound.u[1] = 0x00000000;

    /* initialize the random number generator */
    d_init_addrans_();

    /* test (SIZE * LOOPS) arguments to sin */
    for (j = 0; j < LOOPS; j++) {

        /*
        * generate a vector, x, of length SIZE,
        * of random numbers to use as
        * input to the trig functions.
        */
        n = SIZE;
        ub = upperbound.d;
        lb = 0.0;
```

CODE EXAMPLE A-4 Using `d_addrans` *(Continued)*

```
        d_addrans_(x, &n, &lb, &ub);

        for (i = 0; i < n; i++)
            y[i] = sin(x[i]);

        /* is sin(x) == x?  It ought to, for tiny x. */
        for (i = 0; i < n; i++)
            if (x[i] != y[i])
                printf(
                " OOPS: %d sin(%18.17e)=%18.17e \n",
                i, x[i], y[i]);
    }
    printf(" comparison ended; no differences\n");
    ieee_retrospective_();
    return 0;
}
```

IEEE Recommended Functions

This Fortran example uses some functions recommended by the IEEE standard:

CODE EXAMPLE A-5 IEEE Recommended Functions

```
c
c   Demonstrate how to call 5 of the more interesting IEEE
c   recommended functions from Fortran. These are implemented
c   with "bit-twiddling", and so are as efficient as you could
c   hope. The IEEE standard for floating-point arithmetic
c   doesn't require these, but recommends that they be
c   included in any IEEE programming environment.
c
c   For example, to accomplish
c   y = x * 2**n,
c   since the hardware stores numbers in base 2,
c   shift the exponent by n places.
c
```

```
c   Refer to
c   ieee_functions(3m)
c   libm_double(3f)
c   libm_single(3f)
c
c   The 5 functions demonstrated here are:
c
c   ilogb(x): returns the base 2 unbiased exponent of x in
c       integer format
c   signbit(x): returns the sign bit, 0 or 1
c   copysign(x,y): returns x with y's sign bit
c   nextafter(x,y): next representable number after x, in
c       the direction y
c   scalbn(x,n): x * 2**n
c
 c      function       double precision      single precision
 c      --------------------------------------------------------
 c      ilogb(x)       i = id_ilogb(x)       i = ir_ilogb(r)
 c      signbit(x)     i = id_signbit(x)     i = ir_signbit(r)
 c      copysign(x,y)  x = d_copysign(x,y)   r = r_copysign(r,s)
 c      nextafter(x,y) z = d_nextafter(x,y)  r = r_nextafter(r,s)
 c      scalbn(x,n)    x = d_scalbn(x,n)     r = r_scalbn(r,n)
    program ieee_functions_demo
    implicit double precision (d)
    implicit real (r)
    double precision   x, y, z, direction
    real               r, s, t, r_direction
    integer            i, scale

    print *
    print *, 'DOUBLE PRECISION EXAMPLES:'
    print *

    x = 32.0d0
    i = id_ilogb(x)
    write(*,1) x, i
 1  format(' The base 2 exponent of ', F4.1, ' is ', I2)

    x = -5.5d0
    y = 12.4d0
    z = d_copysign(x,y)
    write(*,2) x, y, z
 2    format(F5.1, ' was given the sign of ', F4.1,
     *   ' and is now ', F4.1)
```

```
      x = -5.5d0
      i = id_signbit(x)
      print *, 'The sign bit of ', x, ' is ', i

      x = d_min_subnormal()
      direction = -d_infinity()
      y = d_nextafter(x, direction)
      write(*,3) x
   3  format(' Starting from ', 1PE23.16E3,
     -   ', the next representable number ')
      write(*,4) direction, y
   4  format('    towards ', F4.1, ' is ', 1PE23.16E3)

      x = d_min_subnormal()
      direction = 1.0d0
      y = d_nextafter(x, direction)
      write(*,3) x
      write(*,4) direction, y
      x = 2.0d0
      scale = 3
      y = d_scalbn(x, scale)
      write (*,5) x, scale, y
   5  format(' Scaling ', F4.1, ' by 2**', I1, ' is ', F4.1)
      print *
      print *, 'SINGLE PRECISION EXAMPLES:'
      print *

      r = 32.0
      i = ir_ilogb(r)
      write (*,1) r, i

      r = -5.5
      i = ir_signbit(r)
      print *, 'The sign bit of ', r, ' is ', i

      r = -5.5
      s = 12.4
      t = r_copysign(r,s)
      write (*,2) r, s, t

      r = r_min_subnormal()
      r_direction = -r_infinity()
      s = r_nextafter(r, r_direction)
      write(*,3) r
      write(*,4) r_direction, s
```

CODE EXAMPLE A-5 IEEE Recommended Functions *(Continued)*

```
      r = r_min_subnormal()
      r_direction = 1.0e0
      s = r_nextafter(r, r_direction)
      write(*,3) r
      write(*,4) r_direction, s

      r = 2.0
      scale = 3
      s = r_scalbn(r, scale)
      write (*,5) r, scale, y

      print *
      end
```

The output from this program is shown in CODE EXAMPLE A-6.

CODE EXAMPLE A-6 Output of CODE EXAMPLE A-5

```
DOUBLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
-5.5 was given the sign of 12.4 and is now  5.5
The sign bit of    -5.5 is   1
Starting from  4.9406564584124654E-324, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  4.9406564584124654E-324, the next representable
   number towards  1.0 is  9.8813129168249309E-324
Scaling  2.0 by 2**3 is 16.0

SINGLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
The sign bit of    -5.5 is   1
-5.5 was given the sign of 12.4 and is now  5.5
Starting from  1.4012984643248171E-045, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  1.4012984643248171E-045, the next representable
   number towards  1.0 is  2.8025969286496341E-045
Scaling  2.0 by 2**3 is 16.0
```

If using the `f95` compiler with the `-f77` compatibility option, the following additional messages are displayed.

```
Note:IEEE floating-point exception flags raised:
   Inexact; Underflow;
See the Numerical Computation Guide, ieee_flags(3M)
```

IEEE Special Values

This C program calls several of the `ieee_values`(3m) functions:

```
#include <math.h>
#include <sunmath.h>

int main()
{
    double   x;
    float    r;

    x = quiet_nan(0);
    printf("quiet NaN: %.16e = %08x %08x \n",
        x, ((int *) &x)[0], ((int *) &x)[1]);

    x = nextafter(max_subnormal(), 0.0);
    printf("nextafter(max_subnormal,0) = %.16e\n",x);
    printf("                           = %08x %08x\n",
        ((int *) &x)[0], ((int *) &x)[1]);

    r = min_subnormalf();
    printf("single precision min subnormal = %.8e = %08x\n",
        r, ((int *) &r)[0]);

    return 0;
}
```

Remember to specify both `-lsunmath` and `-lm` when linking.

On SPARC, the output looks like this:

```
quiet NaN: NaN = 7fffffff ffffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                           = 000fffff fffffffe
single precision min subnormal = 1.40129846e-45 = 00000001
```

Because the x86 architecture is “little-endian”, the output on x86 is slightly different (the high and low order words of the hexadecimal representations of the double precision numbers are reversed):

```
quiet NaN: NaN = ffffffff 7fffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                           = fffffffe 000fffff
single precision min subnormal = 1.40129846e-45 = 00000001
```

Fortran programs that use `ieee_values` functions should take care to declare those functions’ types:

```
      program print_ieee_values
c
c the purpose of the implicit statements is to insure
c that the floatingpoint pseudo-instrinsic
c functions are declared with the correct type
c
      implicit real*16 (q)
      implicit double precision (d)
      implicit real (r)
      real*16 z, zero, one
      double precision    x
      real                r
c
      zero = 0.0
      one = 1.0
      z = q_nextafter(zero, one)
      x = d_infinity()
      r = r_max_normal()
c
      print *, z
      print *, x
      print *, r
c
      end
```

On SPARC, the output reads as follows:

```
  6.475175119438025110924438958227646-4966
  Inf
  3.40282E+38
```

`ieee_flags` — Rounding Direction

The following example demonstrates how to set the rounding mode to *round towards zero*:

```
#include <math.h>
#include <sunmath.h>

int main()
{
    int         i;
    double      x, y;
    char        *out_1, *out_2, *dummy;

    /* get prevailing rounding direction */
    i = ieee_flags("get", "direction", "", &out_1);

    x = sqrt(.5);
    printf("With rounding direction %s, \n", out_1);
    printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
           ((int *) &x)[0], ((int *) &x)[1], x);

    /* set rounding direction */
    if (ieee_flags("set", "direction", "tozero", &dummy) != 0)
        printf("Not able to change rounding direction!\n");
    i = ieee_flags("get", "direction", "", &out_2);

    x = sqrt(.5);
    /*
     * restore original rounding direction before printf, since
     * printf is also affected by the current rounding direction
     */
    if (ieee_flags("set", "direction", out_1, &dummy) != 0)
        printf("Not able to change rounding direction!\n");
    printf("\nWith rounding direction %s,\n", out_2);
    printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
           ((int *) &x)[0], ((int *) &x)[1], x);

    return 0;
}
```

(SPARC) The output of this short program shows the effects of rounding towards zero:

```
demo% cc rounding_direction.c -lsunmath -lm
demo% a.out
With rounding direction nearest,
sqrt(.5) = 0x3fe6a09e 0x667f3bcd  = 7.071067811865476e-01

With rounding direction tozero,
sqrt(.5) = 0x3fe6a09e 0x667f3bcc  = 7.071067811865475e-01
demo%
```

(x86) The output of this short program shows the effects of rounding towards zero:

```
demo% cc rounding_direction.c -lsunmath -lm
demo% a.out
With rounding direction nearest,
sqrt(.5) = 0x667f3bcd 0x3fe6a09e  = 7.071067811865476e-01

With rounding direction tozero,
sqrt(.5) = 0x667f3bcc 0x3fe6a09e  = 7.071067811865475e-01
demo%
```

To set rounding direction towards zero from a Fortran program:

```
program ieee_flags_demo
character*16 out

i = ieee_flags('set', 'direction', 'tozero', out)
if (i.ne.0) print *, 'not able to set rounding direction'

i = ieee_flags('get', 'direction', '', out)
print *, 'Rounding direction is: ', out

end
```

The output is as follows:

```
demo% f95 ieee_flags_demo.f
demo% a.out
 Rounding direction is: tozero
```

If the program is compiled using the `f95` compiler with the `-f77` compatibility option, the output includes the following additional messages.

```
demo% f95 -f77 ieee_flags_demo.f
ieee_flags_demo.f:
 MAIN ieee_flags_demo:
demo% a.out
 Rounding direction is: tozero
 Note: Rounding direction toward zero
 See the Numerical Computation Guide, ieee_flags(3M)
```

C99 Floating Point Environment Functions

The next example illustrates the use of several of the C99 floating point environment functions. The `norm` function computes the Euclidean norm of a vector and uses the environment functions to handle underflow and overflow. The main program calls this function with vectors that are scaled to ensure that underflows and overflows occur, as the retrospective diagnostic output shows.

CODE EXAMPLE A-7 C99 Floating Point Environment Functions

```
#include <stdio.h>
#include <math.h>
#include <sunmath.h>
#include <fenv.h>

/*
*  Compute the euclidean norm of the vector x avoiding
*  premature underflow or overflow
*/
double norm(int n, double *x)
{
    fenv_t  env;
    double  s, b, d, t;
    int     i, f;

    /* save the environment, clear flags, and establish nonstop
       exception handling */
    feholdexcept(&env);
```

CODE EXAMPLE A-7 C99 Floating Point Environment Functions *(Continued)*

```
    /* attempt to compute the dot product x.x */
    d = 1.0; /* scale factor */
    s = 0.0;
    for (i = 0; i < n; i++)
        s += x[i] * x[i];

    /* check for underflow or overflow */
    f = fetestexcept(FE_UNDERFLOW | FE_OVERFLOW);
    if (f & FE_OVERFLOW) {
        /* first attempt overflowed, try again scaling down */
        feclearexcept(FE_OVERFLOW);
        b = scalbn(1.0, -640);
        d = 1.0 / b;
        s = 0.0;
        for (i = 0; i < n; i++) {
            t = b * x[i];
            s += t * t;
        }
    }
    else if (f & FE_UNDERFLOW && s < scalbn(1.0, -970)) {
        /* first attempt underflowed, try again scaling up */
        b = scalbn(1.0, 1022);
        d = 1.0 / b;
        s = 0.0;
        for (i = 0; i < n; i++) {
            t = b * x[i];
            s += t * t;
        }
    }

    /* hide any underflows that have occurred so far */
    feclearexcept(FE_UNDERFLOW);

    /* restore the environment, raising any other exceptions
       that have occurred */
    feupdateenv(&env);

    /* take the square root and undo any scaling */
    return d * sqrt(s);
}
```

CODE EXAMPLE A-7 C99 Floating Point Environment Functions *(Continued)*

```
int main()
{
    double x[100], l, u;
    int    n = 100;

    fex_set_log(stdout);

    l = 0.0;
    u = min_normal();
    d_lcrans_(x, &n, &l, &u);
    printf("norm: %g\n", norm(n, x));
    l = sqrt(max_normal());
    u = l * 2.0;
    d_lcrans_(x, &n, &l, &u);
    printf("norm: %g\n", norm(n, x));

    return 0;
}
```

On SPARC, compiling and running this program produces the following:

```
demo% cc norm.c -R/opt/SUNWspro/lib -L/opt/SUNWspro/lib -lm9x
-lsunmath -lm
demo% a.out
Floating point underflow at 0x000153a8 __d_lcrans_, nonstop mode
  0x000153b4  __d_lcrans_
  0x00011594  main
Floating point underflow at 0x00011244 norm, nonstop mode
  0x00011248  norm
  0x000115b4  main
norm: 1.32533e-307
Floating point overflow at 0x00011244 norm, nonstop mode
  0x00011248  norm
  0x00011660  main
norm: 2.02548e+155
```

CODE EXAMPLE A-8 shows the effect of the `fesetprec` function on x86. (This function is not available on SPARC.) The `while` loops attempt to determine the available precision by finding the largest power of two that rounds off entirely when it is added to one. As the first loop shows, this technique does not always work as expected on architectures like x86 that evaluate all intermediate results in extended precision. Thus, the `fesetprec` function may be used to guarantee that all results will be rounded to the desired precision, as the second loop shows.

CODE EXAMPLE A-8 `fesetprec` Function (x86)

```
#include <math.h>
#include <fenv.h>

int main()
{
    double  x;

    x = 1.0;
    while (1.0 + x != 1.0)
        x *= 0.5;
    printf("%d significant bits\n", -ilogb(x));

    fesetprec(FE_DBLPREC);
    x = 1.0;
    while (1.0 + x != 1.0)
        x *= 0.5;
    printf("%d significant bits\n", -ilogb(x));

    return 0;
}
```

The output from this program on x86 systems is:

```
64 significant bits
53 significant bits
```

Finally, CODE EXAMPLE A-9 shows one way to use the environment functions in a multi-threaded program to propagate floating point modes from a parent thread to a child thread and recover exception flags raised in the child thread when it joins with the parent. (See the *Solaris Multithreaded Programming Guide* for more information on writing multi-threaded programs.)

CODE EXAMPLE A-9 Using Environment Functions in Multi-Thread Program

```
#include <thread.h>
#include <fenv.h>

fenv_t  env;

void child(void *p)
{
    /* inherit the parent's environment on entry */
    fesetenv(&env);
    ...
    /* save the child's environment before exit */
    fegetenv(&env);
}

void parent()
{
    thread_t tid;
    void *arg;
    ...
    /* save the parent's environment before creating the child */
    fegetenv(&env);
    thr_create(NULL, NULL, child, arg, NULL, &tid);
    ...
    /* join with the child */
    thr_join(tid, NULL, &arg);
    /* merge exception flags raised in the child into the
       parent's environment */
    fex_merge_flags(&env);
    ...
}
```

Exceptions and Exception Handling

ieee_flags — Accrued Exceptions

Generally, a user program *examines* or *clears* the accrued exception bits. CODE EXAMPLE A-10 is a C program that examines the accrued exception flags.

CODE EXAMPLE A-10 Examining the Accrued Exception Flags

```
#include <sunmath.h>
#include <sys/ieeefp.h>

int main()
{
    int     code, inexact, division, underflow, overflow, invalid;
    double  x;
    char    *out;

    /* cause an underflow exception */
    x = max_subnormal() / 2.0;

    /* this statement insures that the previous */
    /* statement is not optimized away          */
    printf("x = %g\n",x);

    /* find out which exceptions are raised */
    code = ieee_flags("get", "exception", "", &out);

    /* decode the return value */
    inexact =      (code >> fp_inexact)     & 0x1;
    underflow =    (code >> fp_underflow)   & 0x1;
    division =     (code >> fp_division)    & 0x1;
    overflow =     (code >> fp_overflow)    & 0x1;
    invalid =      (code >> fp_invalid)     & 0x1;
```

CODE EXAMPLE A-10 Examining the Accrued Exception Flags *(Continued)*

```
     /* "out" is the raised exception with the highest priority */
    printf(" Highest priority exception is: %s\n", out);
    /* The value 1 means the exception is raised, */
    /* 0 means it isn't.                          */
    printf("%d %d %d %d %d\n", invalid, overflow, division,
    underflow, inexact);
    ieee_retrospective_();
    return 0;
}
```

The output from running this program:

```
demo% a.out
x = 1.11254e-308
 Highest priority exception is: underflow
0 0 0 1 1
 Note:IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 See the Numerical Computation Guide, ieee_flags(3M)
```

The same can be done from Fortran:

CODE EXAMPLE A-11 Examining the Accrued Exception Flags – Fortran

```
/*
A Fortran example that:
    *  causes an underflow exception
    *  uses ieee_flags to determine which exceptions are raised
    *  decodes the integer value returned by ieee_flags
    *  clears all outstanding exceptions
Remember to save this program in a file with the suffix .F, so that
the c preprocessor is invoked to bring in the header file
floatingpoint.h.
*/
#include <floatingpoint.h>
```

CODE EXAMPLE A-11 Examining the Accrued Exception Flags – Fortran *(Continued)*

```
      program decode_accrued_exceptions
      double precision   x
      integer            accrued, inx, div, under, over, inv
      character*16       out
      double precision   d_max_subnormal
c Cause an underflow exception
      x = d_max_subnormal() / 2.0

c Find out which exceptions are raised
      accrued = ieee_flags('get', 'exception', '', out)

c Decode value returned by ieee_flags using bit-shift intrinsics
      inx   = and(rshift(accrued, fp_inexact)  , 1)
      under = and(rshift(accrued, fp_underflow), 1)
      div   = and(rshift(accrued, fp_division) , 1)
      over  = and(rshift(accrued, fp_overflow) , 1)
      inv   = and(rshift(accrued, fp_invalid)  , 1)

c The exception with the highest priority is returned in "out"
      print *, "Highest priority exception is ", out

c The value 1 means the exception is raised; 0 means it is not
      print *, inv, over, div, under, inx

c Clear all outstanding exceptions
      i = ieee_flags('clear', 'exception', 'all', out)
      end
```

The output is as follows:

```
 Highest priority exception is underflow
   0  0  0  1  1
```

While it is unusual for a user program to *set* exception flags, it can be done. This is demonstrated in the following C example.

```
#include <sunmath.h>

int main()
{
    int     code;
    char    *out;

    if (ieee_flags("clear", "exception", "all", &out) != 0)
        printf("could not clear exceptions\n");
    if (ieee_flags("set", "exception", "division", &out) != 0)
        printf("could not set exception\n");
    code = ieee_flags("get", "exception", "", &out);
    printf("out is: %s , fp exception code is: %X \n",
    out, code);

    return 0;
}
```

On SPARC, the output from the preceding program is:

```
out is: division , fp exception code is: 2
```

On x86, the output is:

```
out is: division , fp exception code is: 4
```

`ieee_handler` — Trapping Exceptions

Note – The examples below apply only to the Solaris operating environment.

Here is a Fortran program that installs a signal handler to locate an exception (for SPARC systems only):

CODE EXAMPLE A-12 Trap on Underflow – SPARC

```
      program demo

c declare signal handler function
      external fp_exc_hdl
      double precision   d_min_normal
      double precision   x

c set up signal handler
      i = ieee_handler('set', 'common', fp_exc_hdl)
      if (i.ne.0) print *, 'ieee trapping not supported here'

c cause an underflow exception (it will not be trapped)
      x = d_min_normal() / 13.0
      print *, 'd_min_normal() / 13.0 = ', x

c cause an overflow exception
c the value printed out is unrelated to the result
      x = 1.0d300*1.0d300
      print *, '1.0d300*1.0d300 = ', x

      end
c
c the floating-point exception handling function
c
      integer function fp_exc_hdl(sig, sip, uap)
      integer sig, code, addr
      character label*16
c
c The structure /siginfo/ is a translation of siginfo_t
c from <sys/siginfo.h>
c
```

CODE EXAMPLE A-12 Trap on Underflow – SPARC *(Continued)*

```
      structure /fault/
      integer address
      end structure

      structure /siginfo/
      integer si_signo
      integer si_code
      integer si_errno
      record /fault/ fault
      end structure

      record /siginfo/ sip

c See <sys/machsig.h> for list of FPE codes
c Figure out the name of the SIGFPE
      code = sip.si_code
      if (code.eq.3) label = 'division'
      if (code.eq.4) label = 'overflow'
      if (code.eq.5) label = 'underflow'
      if (code.eq.6) label = 'inexact'
      if (code.eq.7) label = 'invalid'
      addr = sip.fault.address

c Print information about the signal that happened
      write (*,77) code, label, addr
 77   format ('floating-point exception code ', i2, ',',
     *         a17, ',', ' at address ', z8 )

      end
```

The output is:

```
 d_min_normal() / 13.0 =     1.7115952757748-309
floating-point exception code  4, overflow       , at address
1131C
 1.0d300*1.0d300 =     1.0000000000000+300
 Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
    ieee_handler(3M)
```

(SPARC) Here is a more complex C example:

CODE EXAMPLE A-13 Trap on Invalid, Division by 0, Overflow, Underflow, and Inexact – SPARC

```
/*
 * Generate the 5 IEEE exceptions: invalid, division,
 * overflow, underflow and inexact.
 *
 * Trap on any floating point exception, print a message,
 * and continue.
 *
 * Note that you could also inquire about raised exceptions by
 *    i = ieee("get","exception","",&out);
 * where out contains the name of the highest exception
 * raised, and i can be decoded to find out about all the
 * exceptions raised.
 */

#include <sunmath.h>
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>

extern void trap_all_fp_exc(int sig, siginfo_t *sip,
    ucontext_t *uap);
```

CODE EXAMPLE A-13 Trap on Invalid, Division by 0, Overflow, Underflow, and Inexact – SPARC *(Continued)*

```
int main()
{
    doublex, y, z;
    char*out;

    /*
     * Use ieee_handler to establish "trap_all_fp_exc"
     * as the signal handler to use whenever any floating
     * point exception occurs.
     */

    if (ieee_handler("set", "all", trap_all_fp_exc) != 0)
    printf(" IEEE trapping not supported here.\n");
    /* disable trapping (uninteresting) inexact exceptions */
    if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
        printf("Trap handler for inexact not cleared.\n");

    /* raise invalid */
    if (ieee_flags("clear", "exception", "all", &out) != 0)
        printf(" could not clear exceptions\n");
    printf("1. Invalid: signaling_nan(0) * 2.5\n");
    x = signaling_nan(0);
    y = 2.5;
    z = x * y;

    /* raise division */
    if (ieee_flags("clear", "exception", "all", &out) != 0)
    printf(" could not clear exceptions\n");
    printf("2. Div0: 1.0 / 0.0\n");
    x = 1.0;
    y = 0.0;
    z = x / y;

    /* raise overflow */
    if (ieee_flags("clear", "exception", "all", &out) != 0)
        printf(" could not clear exceptions\n");
    printf("3. Overflow: -max_normal() - 1.0e294\n");
    x = -max_normal();
    y = -1.0e294;
    z = x + y;
```

CODE EXAMPLE A-13 Trap on Invalid, Division by 0, Overflow, Underflow, and Inexact – SPARC *(Continued)*

```
    /* raise underflow */
    if (ieee_flags("clear", "exception", "all", &out) != 0)
        printf(" could not clear exceptions\n");
    printf("4. Underflow: min_normal() * min_normal()\n");
    x = min_normal();
    y = x;
    z = x * y;

    /* enable trapping on inexact exception */
    if (ieee_handler("set", "inexact", trap_all_fp_exc) != 0)
        printf("Could not set trap handler for inexact.\n");
    /* raise inexact */
    if (ieee_flags("clear", "exception", "all", &out) != 0)
        printf(" could not clear exceptions\n");
    printf("5. Inexact: 2.0 / 3.0\n");
    x = 2.0;
    y = 3.0;
    z = x / y;

    /* don't trap on inexact */
    if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
        printf(" could not reset inexact trap\n");

    /* check that we're not trapping on inexact anymore */
    if (ieee_flags("clear", "exception", "all", &out) != 0)
        printf(" could not clear exceptions\n");
    printf("6. Inexact trapping disabled; 2.0 / 3.0\n");
    x = 2.0;
    y = 3.0;
    z = x / y;

    /* find out if there are any outstanding exceptions */
    ieee_retrospective_();

    /* exit gracefully */
    return 0;
}

void trap_all_fp_exc(int sig, siginfo_t *sip, ucontext_t *uap) {
    char*label = "undefined";
```

CODE EXAMPLE A-13 Trap on Invalid, Division by 0, Overflow, Underflow, and Inexact – SPARC *(Continued)*

```
/* see /usr/include/sys/machsig.h for SIGFPE codes */
    switch (sip->si_code) {
    case FPE_FLTRES:
        label = "inexact";
        break;
    case FPE_FLTDIV:
        label = "division";
        break;
    case FPE_FLTUND:
        label = "underflow";
        break;
    case FPE_FLTINV:
        label = "invalid";
        break;
    case FPE_FLTOVF:
        label = "overflow";
        break;
    }

    printf(
    " signal %d, sigfpe code %d: %s exception at address %x\n",
        sig, sip->si_code, label, sip->_data._fault._addr);
}
```

The output is similar to the following:

```
1. Invalid: signaling_nan(0) * 2.5
   signal 8, sigfpe code 7: invalid exception at address 10da8
2. Div0: 1.0 / 0.0
   signal 8, sigfpe code 3: division exception at address 10e44
3. Overflow: -max_normal() - 1.0e294
   signal 8, sigfpe code 4: overflow exception at address 10ee8
4. Underflow: min_normal() * min_normal()
   signal 8, sigfpe code 5: underflow exception at address 10f80
5. Inexact: 2.0 / 3.0
   signal 8, sigfpe code 6: inexact exception at address 1106c
6. Inexact trapping disabled; 2.0 / 3.0
Note: IEEE floating-point exception traps enabled:
   underflow; overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_handler(3M)
```

(SPARC) CODE EXAMPLE A-14 shows how you can use `ieee_handler` and the include files to modify the default result of certain exceptional situations:

CODE EXAMPLE A-14 Modifying the Default Result of Exceptional Situations

```
/*
 * Cause a division by zero exception and use the
 * signal handler to substitute MAXDOUBLE (or MAXFLOAT)
 * as the result.
 *
 * compile with the flag -Xa
 */

#include <values.h>
#include <siginfo.h>
#include <ucontext.h>

void division_handler(int sig, siginfo_t *sip, ucontext_t *uap);

int main() {
    double     x, y, z;
    float      r, s, t;
    char       *out;

    /*
     * Use ieee_handler to establish division_handler as the
     * signal handler to use for the IEEE exception division.
     */
    if (ieee_handler("set","division",division_handler)!=0) {
    printf(" IEEE trapping not supported here.\n");
    }

    /* Cause a division-by-zero exception */
    x = 1.0;
    y = 0.0;
    z = x / y;

    /*
     * Check to see that the user-supplied value, MAXDOUBLE,
     * is indeed substituted in place of the IEEE default
     * value, infinity.
     */
    printf("double precision division: %g/%g = %g \n",x,y,z);
```

CODE EXAMPLE A-14 Modifying the Default Result of Exceptional Situations *(Continued)*

```
    /* Cause a division-by-zero exception */
    r = 1.0;
    s = 0.0;
    t = r / s;

    /*
     * Check to see that the user-supplied value, MAXFLOAT,
     * is indeed substituted in place of the IEEE default
     * value, infinity.
     */
    printf("single precision division: %g/%g = %g \n",r,s,t);

    ieee_retrospective_();

    return 0;
}

void division_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
    int             inst;
    unsigned        rd, mask, single_prec=0;
    float           f_val = MAXFLOAT;
    double          d_val = MAXDOUBLE;
    long            *f_val_p = (long *) &f_val;

    /* Get instruction that caused exception. */
    inst = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

    /*
     * Decode the destination register. Bits 29:25 encode the
     * destination register for any SPARC floating point
     * instruction.
     */
    mask = 0x1f;
    rd = (mask & (inst >> 25));

    /*
     * Is this a single precision or double precision
     * instruction?  Bits 5:6 encode the precision of the
     * opcode; if bit 5 is 1, it's sp, else, dp.
     */
```

CODE EXAMPLE A-14 Modifying the Default Result of Exceptional Situations *(Continued)*

```
    mask = 0x1;
    single_prec = (mask & (inst >> 5));

    /* put user-defined value into destination register */
    if (single_prec) {
    uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[rd] =
        f_val_p[0];
    } else {
    uap->uc_mcontext.fpregs.fpu_fr.fpu_dregs[rd/2] = d_val;
    }
}
```

As expected, the output is:

```
double precision division: 1/0 = 1.79769e+308
single precision division: 1/0 = 3.40282e+38
Note: IEEE floating-point exception traps enabled:
   division by zero;
See the Numerical Computation Guide, ieee_handler(3M)
```

`ieee_handler` — Abort on Exceptions

You can use `ieee_handler` to force a program to abort in case of certain floating-point exceptions:

```
#include <floatingpoint.h>
    program abort
c
    ieeer = ieee_handler('set', 'division', SIGFPE_ABORT)
    if (ieeer .ne. 0) print *, ' ieee trapping not supported'
    r = 14.2
    s = 0.0
    r = r/s
c
    print *, 'you should not see this; system should abort'
c
    end
```

`libm9x.so` Exception Handling Features

The following examples show how to use some of the exception handling features provided by `libm9x.so`. The first example is based on the following task: given a number x and coefficients $a_0, a_1, \ldots, a_N$, and $b_0, b_1, \ldots, b_{N-1}$, evaluate the function $f(x)$ and its first derivative $f'(x)$, where f is the continued fraction

$$f(x) = a_0 + b_0/(x + a_1 + b_1/(x + \ldots /(x + a_{N-1} + b_{N-1}/(x + a_N))\ldots)).$$

Computing f is straightforward in IEEE arithmetic: even if one of the intermediate divisions overflows or divides by zero, the default value specified by the standard (a correctly signed infinity) turns out to yield the correct result. Computing f', on the other hand, can be more difficult because the simplest form for evaluating it can have removable singularities. If the computation encounters one of these singularities, it will attempt to evaluate one of the indeterminate forms 0/0, 0*infinity, or infinity/infinity, all of which raise invalid operation exceptions. W. Kahan has proposed a method for handling these exceptions via a feature called "presubstitution".

Presubstitution is an extension of the IEEE default response to exceptions that lets the user specify in advance the value to be substituted for the result of an exceptional operation. Using `libm9x.so`, a program can implement presubstitution easily by installing a handler in the `FEX_CUSTOM` exception handling mode. This mode allows the handler to supply any value for the result of an exceptional operation simply by storing that value in the data structure pointed to by the *info* parameter passed to the handler. Here is a sample program to compute the continued fraction and its derivative using presubstitution implemented with a `FEX_CUSTOM` handler.

CODE EXAMPLE A-15 Computing the Continued Fraction and Its Derivative Using the `FEX_CUSTOM` Handler

```
#include <stdio.h>
#include <sunmath.h>
#include <fenv.h>
volatile double p;
void handler(int ex, fex_info_t *info)
{
    info->res.type = fex_double;
    if (ex == FEX_INV_ZMI)
        info->res.val.d = p;
    else
        info->res.val.d = infinity();
}
```

CODE EXAMPLE A-15 Computing the Continued Fraction and Its Derivative Using the `FEX_CUSTOM` Handler *(Continued)*

```
/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double x, double *pf, double *pf1)
{
    fex_handler_t    oldhdl; /* for saving/restoring handlers */
    volatile double  t;
    double           f, f1, d, d1, q;
    int              j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    fex_set_handling(FEX_DIVBYZERO, FEX_NONSTOP, NULL);
    fex_set_handling(FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI,
        FEX_CUSTOM, handler);

    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        /* the following assignment to the volatile variable t
           is needed to maintain the correct sequencing between
           assignments to p and evaluation of f1 */
        t = f1 = (-d1 / d) * q;
        p = b[j-1] * d1 / b[j];
        f = a[j] + q;
    }

    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}
```

CODE EXAMPLE A-15 Computing the Continued Fraction and Its Derivative Using the FEX_CUSTOM Handler *(Continued)*

```
/* For the following coefficients, x = -3, 1, 4, and 5 will all
   encounter intermediate exceptions */
double a[] = { -1.0, 2.0, -3.0, 4.0, -5.0 };
double b[] = { 2.0, 4.0, 6.0, 8.0 };

int main()
{
    double  x, f, f1;
    int     i;

    feraiseexcept(FE_INEXACT); /* prevent logging of inexact */
    fex_set_log(stdout);
    fex_set_handling(FEX_COMMON, FEX_ABORT, NULL);
    for (i = -5; i <= 5; i++) {
        x = i;
        continued_fraction(4, a, b, x, &f, &f1);
        printf("f(% g) = %12g, f’(% g) = %12g\n", x, f, x, f1);
    }
    return 0;
}
```

Several comments about the program are in order. On entry, the function `continued_fraction` saves the current exception handling modes for division by zero and all invalid operation exceptions. It then establishes nonstop exception handling for division by zero and a `FEX_CUSTOM` handler for the three indeterminate forms. This handler will substitute infinity for both 0/0 and infinity/infinity, but it will substitute the value of the global variable `p` for 0*infinity. Note that `p` must be recomputed each time through the loop that evaluates the function in order to supply the correct value to substitute for a subsequent 0*infinity invalid operation. Note also that `p` must be declared `volatile` to prevent the compiler from eliminating it, since it is not explicitly mentioned elsewhere in the loop. Finally, to prevent the compiler from moving the assignment to `p` above or below the computation that can incur the exception for which `p` provides the presubstitution value, the result of that computation is also assigned to a `volatile` variable (called `t` in the program). The final call to `fex_setexcepthandler` restores the original handling modes for division by zero and the invalid operations.

The main program enables logging of retrospective diagnostics by calling the `fex_set_log` function. Before it does so, it raises the inexact flag; this has the effect of preventing the logging of inexact exceptions. (Recall that in `FEX_NONSTOP` mode, an exception is not logged if its flag is raised, as explained in the section “Retrospective Diagnostics” on page 79.) The main program also establishes

FEX_ABORT mode for the common exceptions to ensure that any unusual exceptions not explicitly handled by continued_fraction will cause program termination. Finally, the program evaluates a particular continued fraction at several different points. As the following sample output shows, the computation does indeed encounter intermediate exceptions:

```
f(-5) =     -1.59649,   f’(-5) =      -0.1818
f(-4) =     -1.87302,   f’(-4) =    -0.428193
Floating point division by zero at 0x08048dbe continued_fraction,
nonstop mode
  0x08048dc1  continued_fraction
  0x08048eda  main
Floating point invalid operation (inf/inf) at 0x08048dcf
continued_fraction, handler: handler
  0x08048dd2  continued_fraction
  0x08048eda  main
Floating point invalid operation (0*inf) at 0x08048dd2
continued_fraction, handler: handler
  0x08048dd8  continued_fraction
  0x08048eda  main
f(-3) =           -3,   f’(-3) =     -3.16667
f(-2) = -4.44089e-16,   f’(-2) =     -3.41667
f(-1) =     -1.22222,   f’(-1) =    -0.444444
f( 0) =     -1.33333,   f’( 0) =     0.203704
f( 1) =           -1,   f’( 1) =     0.333333
f( 2) =    -0.777778,   f’( 2) =      0.12037
f( 3) =    -0.714286,   f’( 3) =    0.0272109
f( 4) =    -0.666667,   f’( 4) =     0.203704
f( 5) =    -0.777778,   f’( 5) =    0.0185185
```

(The exceptions that occur in the computation of $f'(x)$ at $x = 1$, 4, and 5 do not result in retrospective diagnostic messages because they occur at the same site in the program as the exceptions that occur when $x = –3$.)

The preceding program may not represent the most efficient way to handle the exceptions that can occur in the evaluation of a continued fraction and its derivative. One reason is that the presubstitution value must be recomputed in each iteration of the loop regardless of whether or not it is needed. In this case, the computation of the presubstitution value involves a floating point division, and on modern SPARC and x86 processors, floating point division is a relatively slow operation. Moreover, the loop itself already involves two divisions, and because most SPARC and x86 processors cannot overlap the execution of two different division operations, divisions are likely to be a bottleneck in the loop; adding another division would exacerbate the bottleneck.

It is possible to rewrite the loop so that only one division is needed, and in particular, the computation of the presubstitution value need not involve a division. (To rewrite the loop in this way, one must precompute the ratios of adjacent elements of the coefficients in the b array.) This would remove the bottleneck of multiple division operations, but it would not eliminate all of the arithmetic operations involved in the computation of the presubstitution value. Furthermore, the need to assign both the presubstitution value and the result of the operation to be presubstituted to volatile variables introduces additional memory operations that slow the program. While those assignments are necessary to prevent the compiler from reordering certain key operations, they effectively prevent the compiler from reordering other unrelated operations, too. Thus, handling the exceptions in this example via presubstitution requires additional memory operations and precludes some optimizations that might otherwise be possible. Can these exceptions be handled more efficiently?

In the absence of special hardware support for fast presubstitution, the most efficient way to handle exceptions in this example may be to use flags, as the following version does:

CODE EXAMPLE A-16 Using Flags to Handle Exceptions

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double         x, double *pf, double *pf1)
{
    fex_handler_t  oldhdl;
    fexcept_t      oldinvflag;
    double         f, f1, d, d1, pd1, q;
    int            j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);
    fegetexceptflag(&oldinvflag, FE_INVALID);

    fex_set_handling(FEX_DIVBYZERO | FEX_INV_ZDZ | FEX_INV_IDI |
        FEX_INV_ZMI, FEX_NONSTOP, NULL);
    feclearexcept(FE_INVALID);
```

CODE EXAMPLE A-16 Using Flags to Handle Exceptions *(Continued)*

```
    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        f1 = (-d1 / d) * q;
        f = a[j] + q;
    }

    if (fetestexcept(FE_INVALID)) {
        /* recompute and test for NaN */
        f1 = pd1 = 0.0;
        f = a[N];
        for (j = N - 1; j >= 0; j--) {
            d = x + f;
            d1 = 1.0 + f1;
            q = b[j] / d;
            f1 = (-d1 / d) * q;
            if (isnan(f1))
                f1 = b[j] * pd1 / b[j+1];
            pd1 = d1;
            f = a[j] + q;
        }
    }

    fesetexceptflag(&oldinvflag, FE_INVALID);
    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}
```

In this version, the first loop attempts the computation of *f(x)* and *f′(x)* in the default nonstop mode. If the invalid flag is raised, the second loop recomputes *f(x)* and *f′(x)* explicitly testing for the appearance of a NaN. Usually, no invalid operation exception occurs, so the program only executes the first loop. This loop has no references to `volatile` variables and no extra arithmetic operations, so it will run as fast as the compiler can make it go. The cost of this efficiency is the need to write a second loop nearly identical to the first to handle the case when an exception occurs. This trade-off is typical of the dilemmas that floating point exception handling can pose.

Using libm9x.so With Fortran Programs

libm9x.so is primarily intended to be used from C/C++ programs, but by using the Sun Fortran language interoperability features, you can call some libm9x.so functions from Fortran programs as well.

Note – For consistent behavior, do not use both the libm9x.so exception handling functions and the ieee_flags and ieee_handler functions in the same program.

The following example shows a Fortran version of the program to evaluate a continued fraction and its derivative using presubstitution (SPARC only):

CODE EXAMPLE A-17 Evaluating a Continued Fraction and Its Derivative Using Presubstitution – SPARC

```
c
c Presubstitution handler
c
      subroutine handler(ex, info)

      structure /fex_numeric_t/
          integer type
          union
          map
              integer i
          end map

          map
              integer*8 l
          end map
          map
              real f
          end map
          map
              real*8 d
          end map
          map
              real*16 q
          end map
          end union
      end structure
```

CODE EXAMPLE A-17 Evaluating a Continued Fraction and Its Derivative Using Presubstitution – SPARC *(Continued)*

```
      structure /fex_info_t/
          integer op, flags
          record /fex_numeric_t/ op1, op2, res
      end structure

      integer ex
      record /fex_info_t/ info

      common /presub/ p
      double precision  p, d_infinity
      volatile          p

c 4 = fex_double; see <fenv.h> for this and other constants
      info.res.type = 4

c x'80' = FEX_INV_ZMI
      if (loc(ex) .eq. x'80') then
          info.res.d = p
      else
          info.res.d = d_infinity()
      endif
      return
      end

c
c Evaluate the continued fraction given by coefficients a(j) and
c b(j) at the point x; return the function value in f and the
c derivative in f1
c

      subroutine continued_fraction(n, a, b, x, f, f1)

      integer           n
      double precision  a(*), b(*), x, f, f1

      common            /presub/ p
      integer           j, oldhdl
      dimension        oldhdl(24)
      double precision d, d1, q, p, t
      volatile         p, t
```

CODE EXAMPLE A-17 Evaluating a Continued Fraction and Its Derivative Using Presubstitution – SPARC *(Continued)*

```
      external fex_getexcepthandler, fex_setexcepthandler
      external fex_set_handling, handler
c$pragma c(fex_getexcepthandler, fex_setexcepthandler)
c$pragma c(fex_set_handling)

c x'ff2' = FEX_DIVBYZERO | FEX_INVALID
      call fex_getexcepthandler(oldhdl, %val(x'ff2'))

c x'2' = FEX_DIVBYZERO, 0 = FEX_NONSTOP
      call fex_set_handling(%val(x'2'), %val(0), %val(0))

c x'b0' = FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI, 3 = FEX_CUSTOM
      call fex_set_handling(%val(x'b0'), %val(3), handler)

      f1 = 0.0d0
      f = a(n+1)
      do j = n, 1, -1
          d = x + f
          d1 = 1.0d0 + f1
          q = b(j) / d
          f1 = (-d1 / d) * q
c
c         the following assignment to the volatile variable t
c         is needed to maintain the correct sequencing between
c         assignments to p and evaluation of f1

          t = f1
          p = b(j-1) * d1 / b(j)
          f = a(j) + q
      end do

      call fex_setexcepthandler(oldhdl, %val(x'ff2'))
      return
      end
```

CODE EXAMPLE A-17 Evaluating a Continued Fraction and Its Derivative Using Presubstitution – SPARC *(Continued)*

```
c Main program
c
      program cf
      integer            i
      double precision   a, b, x, f, f1
      dimension          a(5), b(4)
      data a /-1.0d0, 2.0d0, -3.0d0, 4.0d0, -5.0d0/
      data b /2.0d0, 4.0d0, 6.0d0, 8.0d0/

      external fex_set_handling
c$pragma c(fex_set_handling)

c x'ffa' = FEX_COMMON, 1 = FEX_ABORT
      call fex_set_handling(%val(x'ffa'), %val(1), %val(0))
      do i = -5, 5
          x = dble(i)
          call continued_fraction(4, a, b, x, f, f1)
          write (*, 1) i, f, i, f1
      end do
    1 format('f(', I2, ') = ', G12.6, ', f’'(', I2, ') = ', G12.6)
      end
```

The output from this program reads:

```
f(-5) = -1.59649    , f’(-5) = -.181800
f(-4) = -1.87302    , f’(-4) = -.428193
f(-3) = -3.00000    , f’(-3) = -3.16667
f(-2) = -.444089E-15, f’(-2) = -3.41667
f(-1) = -1.22222    , f’(-1) = -.444444
f( 0) = -1.33333    , f’( 0) = 0.203704
f( 1) = -1.00000    , f’( 1) = 0.333333
f( 2) = -.777778    , f’( 2) = 0.120370
f( 3) = -.714286    , f’( 3) = 0.272109E-01
f( 4) = -.666667    , f’( 4) = 0.203704
f( 5) = -.777778    , f’( 5) = 0.185185E-01
 Note: IEEE floating-point exception flags raised:
    Inexact;  Division by Zero;  Invalid Operation;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M)
```

Miscellaneous

sigfpe — Trapping Integer Exceptions

The previous section showed examples of using ieee_handler. In general, when there is a choice between using ieee_handler or sigfpe, the former is recommended.

Note – sigfpe is available only in the Solaris operating environment.

(SPARC) There are instances, such as trapping integer arithmetic exceptions, when sigfpe is the handler to be used. CODE EXAMPLE A-18 traps on integer division by zero.

CODE EXAMPLE A-18 Trapping Integer Exceptions

```
/* Generate the integer division by zero exception */

#include <siginfo.h>
#include <ucontext.h>
#include <signal.h>

void int_handler(int sig, siginfo_t *sip, ucontext_t *uap);

int main() {
    int   a, b, c;

/*
 * Use sigfpe(3) to establish "int_handler" as the signal handler
 * to use on integer division by zero
 */

/*
 * Integer division-by-zero aborts unless a signal
 * handler for integer division by zero is set up
 */
    sigfpe(FPE_INTDIV, int_handler);
```

CODE EXAMPLE A-18 Trapping Integer Exceptions *(Continued)*

```
    a = 4;
    b = 0;
    c = a / b;
    printf("%d / %d = %d\n\n", a, b, c);
    return 0;
}

void int_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
    printf("Signal %d, code %d, at addr %x\n",
    sig, sip->si_code, sip->_data._fault._addr);

/*
 * Increment the program counter; the operating system does this
 * automatically for floating-point exceptions but not for
 * integer division by zero.
 */
    uap->uc_mcontext.gregs[REG_PC] =
    uap->uc_mcontext.gregs[REG_nPC];
}
```

Calling Fortran From C

Here is a simple example of a C driver calling Fortran subroutines. Refer to the appropriate C and Fortran manuals for more information on working with C and Fortran. The following is the C driver (save it in a file named `driver.c`):

CODE EXAMPLE A-19 Calling Fortran From C

```
/*
 * a demo program that shows:
 * 1. how to call f95 subroutine from C, passing an array argument
 * 2. how to call single precision f95 function from C
 * 3. how to call double precision f95 function from C
 */

extern int      demo_one_(double *);
extern float    demo_two_(float *);
extern double   demo_three_(double *);
```

CODE EXAMPLE A-19 Calling Fortran From C *(Continued)*

```
int main()
{
    doublearray[3][4];
    floatf, g;
    doublex, y;
    int i, j;

    for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++)
        array[i][j] = i + 2*j;

    g = 1.5;
    y = g;

    /* pass an array to a fortran function (print the array) */
    demo_one_(&array[0][0]);
    printf(" from the driver\n");
    for (i = 0; i < 3; i++) {
    for (j = 0; j < 4; j++)
        printf("    array[%d][%d] = %e\n",
            i, j, array[i][j]);
    printf("\n");
    }

    /* call a single precision fortran function */
    f = demo_two_(&g);
    printf(
       " f = sin(g) from a single precision fortran function\n");
    printf("    f, g: %8.7e, %8.7e\n", f, g);
    printf("\n");

    /* call a double precision fortran function */
    x = demo_three_(&y);
    printf(
     " x = sin(y) from a double precision fortran function\n");
    printf("    x, y: %18.17e, %18.17e\n", x, y);

    ieee_retrospective_();
    return 0;
}
```

Save the Fortran subroutines in a file named `drivee.f`:

```
      subroutine demo_one(array)
      double precision array(4,3)
      print *, 'from the fortran routine:'
      do 10 i =1,4
      do 20 j = 1,3
         print *, '   array[', i, '][', j, '] = ', array(i,j)
 20   continue
      print *
 10   continue
      return
      end

      real function demo_two(number)
      real number
      demo_two = sin(number)
      return
      end

      double precision function demo_three(number)
      double precision number
      demo_three = sin(number)
      return
      end
```

Then, perform the compilation and linking:

```
cc -c driver.c
f95 -c drivee.f
    demo_one:
    demo_two:
    demo_three:
f95 -o driver driver.o drivee.o
```

The output looks like this:

```
from the fortran routine:
   array[ 1 ][ 1 ] =  0.0E+0
   array[ 1 ][ 2 ] =  1.0
   array[ 1 ][ 3 ] =  2.0

   array[ 2 ][ 1 ] =  2.0
   array[ 2 ][ 2 ] =  3.0
   array[ 2 ][ 3 ] =  4.0

   array[ 3 ][ 1 ] =  4.0
   array[ 3 ][ 2 ] =  5.0
   array[ 3 ][ 3 ] =  6.0

   array[ 4 ][ 1 ] =  6.0
   array[ 4 ][ 2 ] =  7.0
   array[ 4 ][ 3 ] =  8.0

from the driver
   array[0][0] = 0.000000e+00
   array[0][1] = 2.000000e+00
   array[0][2] = 4.000000e+00
   array[0][3] = 6.000000e+00

   array[1][0] = 1.000000e+00
   array[1][1] = 3.000000e+00
   array[1][2] = 5.000000e+00
   array[1][3] = 7.000000e+00

   array[2][0] = 2.000000e+00
   array[2][1] = 4.000000e+00
   array[2][2] = 6.000000e+00
   array[2][3] = 8.000000e+00

f = sin(g) from a single precision fortran function
   f, g: 9.9749500e-01, 1.5000000e+00

x = sin(y) from a double precision fortran function
   x, y: 9.97494986604054446e-01, 1.50000000000000000e+00
```

Useful Debugging Commands

TABLE A-1 shows examples of debugging commands for the SPARC architecture.

TABLE A-1 Some Debugging Commands (SPARC)

Action	dbx	adb
Set breakpoint		
at function	`stop in myfunct`	`myfunct:b`
at line number	`stop at 29`	
at absolute address		`23a8:b`
at relative address		`main+0x40:b`
Run until breakpoint met	`run`	`:r`
Examine source code	`list`	`<pc,10?ia`
Examine a fp register		
IEEE single precision	`print $f0`	`<f0=X`
decimal equivalent (Hex)	`print -fx $f0`	`<f0=f`
IEEE double precision	`print $f0f1`	`<f0=X; <f1=X`

decimal equivalent (Hex)	`print -flx $f0f1`
	`print -flx $d0`
		`<f0=F`

Examine all fp registers	`regs -F`	`$x for f0-f15`
		`$X for f16-f31`

Examine all registers	`regs`	`$r; $x; $X`
Examine fp status register	`print -fx $fsr`	`<fsr=X`
Put single precision 1.0 in `f0`	`assign $f0=1.0`	`3f800000>f0`
Put double prec 1.0 in `f0/f1`	`assign $f0f1=1.0`	`3ff00000>f0; 0>f1`
Continue execution	`cont`	`:c`
Single step	`step (or next)`	`:s`
Exit the debugger	`quit`	`$q`

TABLE A-2 shows examples of debugging commands for the x86 architecture.

TABLE A-2 Some Debugging Commands (x86)

Action	dbx	adb
Set breakpoint		
at function	`stop in myfunct`	`myfunct:b`
at line number	`stop at 29`	
at absolute address		`23a8:b`
at relative address		`main+0x40:b`
Run until breakpoint met	`run`	`:r`
Examine source code	`list`	`<pc,10?ia`
Examine fp registers	`print $st0` `...` `print $st7`	`$x`
Examine all registers	`examine &$gs/19X`	`$r`
Examine fp status register	`examine &$fstat/X`	`<fstat=X` or `$x`
Continue execution	`cont`	`:c`
Single step	`step (or next)`	`:s`
Exit the debugger	`quit`	`$q`

The following examples show two ways to set a breakpoint at the beginning of the code corresponding to a routine `myfunction` in `adb`. First you can say:

```
myfunction:b
```

Second, you can determine the absolute address that corresponds to the beginning of the piece of code corresponding to `myfunction`, and then set a break at that absolute address:

```
myfunction=X
            23a8
23a8:b
```

The main subroutine in a Fortran program compiled with `f95` is known as `MAIN_` to `adb`. To set a breakpoint at `MAIN_` in `adb`:

```
MAIN_:b
```

When examining the contents of floating-point registers, the hex value shown by the `dbx` command `regs -F` is the base-16 representation, not the number's decimal representation. For SPARC, the `adb` commands `$x` and `$X` display both the hexadecimal representation, and the decimal value. For x86, the `adb` command `$x` displays only the decimal value. For SPARC, the double precision values show the decimal value next to the odd-numbered register.

Because the operating system disables the floating-point unit until it is first used by a process, you cannot modify the floating-point registers until they have been accessed by the program being debugged.

(SPARC) When displaying floating point numbers, you should keep in mind that the size of registers is 32 bits, a single precision floating-point number occupies 32 bits (hence it fits in one register), and double precision floating-point numbers occupy 64 bits (therefore two registers are used to hold a double precision number). In the hexadecimal representation 32 bits correspond to 8-digit numbers. In the following snapshot of FPU registers displayed with `adb`, the display is organized as follows:

<name of fpu register> <IEEE hex value> <single precision> <double precision>

(SPARC) The third column holds the single precision decimal interpretation of the hexadecimal pattern shown in the second column. The fourth column interprets pairs of registers. For example, the fourth column of the `f11` line interprets `f10` and `f11` as a 64-bit IEEE double precision number.

(SPARC) Because `f10` and `f11` are used to hold a double precision value, the interpretation (on the `f10` line) of the first 32 bits of that value, `7ff00000`, as `+NaN`, is irrelevant. The interpretation of all 64 bits, `7ff00000 00000000`, as `+Infinity`, happens to be the meaningful translation.

(SPARC) The adb command $x, that was used to display the first 16 floating-point data registers, also displayed fsr (the floating-point status register):

```
$x
fsr    40020
f0  400921fb     +2.1426990e+00
f1  54442d18     +3.3702806e+12     +3.1415926535897931e+00
f2         2     +2.8025969e-45
f3         0     +0.0000000e+00     +4.2439915819305446e-314
f4  40000000     +2.0000000e+00
f5         0     +0.0000000e+00     +2.0000000000000000e+00
f6  3de0b460     +1.0971904e-01
f7         0     +0.0000000e+00     +1.2154188766544394e-10
f8  3de0b460     +1.0971904e-01
f9         0     +0.0000000e+00     +1.2154188766544394e-10
f10 7ff00000     +NaN
f11        0     +0.0000000e+00     +Infinity
f12 ffffffff     -NaN
f13 ffffffff     -NaN                -NaN
f14 ffffffff     -NaN
f15 ffffffff     -NaN                -NaN
```

(x86) The corresponding output on x86 looks like:

```
$x
80387 chip is present.
cw      0x137f
sw      0x3920
cssel 0x17  ipoff 0x2d93             datasel 0x1f  dataoff 0x5740

 st[0]  +3.24999988079071044921875 e-1            VALID
 st[1]  +5.6539133243479549034419688 e73          EMPTY
 st[2]  +2.0000000000000008881784197              EMPTY
 st[3]  +1.8073218308070440556016047 e-1          EMPTY
 st[4]  +7.9180300235748291015625 e-1             EMPTY
 st[5]  +4.201639036693904927233234 e-13          EMPTY
 st[6]  +4.201639036693904927233234 e-13          EMPTY
 st[7]  +2.7224999213218694649185636              EMPTY
```

Note – *(x86)* cw is the control word; sw is the status word.

APPENDIX **B**

SPARC Behavior and Implementation

This chapter discusses issues related to the floating-point units used in SPARC workstations and describes a way to determine which code generation flags are best suited for a particular workstation.

Floating-Point Hardware

This section lists a number of SPARC floating-point units and describes the instruction sets and exception handling features they support. See the *SPARC Architecture Manual* Version 8 Appendix N, “SPARC IEEE 754 Implementation Recommendations”, and Version 9 Appendix B, “IEEE Std 754-1985 Requirements for SPARC-V9”, for brief descriptions of what happens when a floating-point trap is taken, the distinction between trapped and untrapped underflow, and recommended possible courses of action for SPARC implementations that provide a non-IEEE (nonstandard) arithmetic mode.

TABLE B-1 lists the hardware floating-point implementations used by SPARC workstations. Many early SPARC systems have floating-point units derived from cores developed by TI or Weitek:

- TI family – includes the TI8847 and the TMS390C602A
- Weitek family – includes the 1164/1165, the 3170, and 3171

These two families of FPUs have been licensed to other workstation vendors, so chips from other semiconductor manufacturers may be found in some SPARC workstations. Some of these other chips are also shown in the table.

TABLE B-1 SPARC Floating-Point Options

FPU	Description or Processor Name	Appropriate for Machines	Notes	Optimum -xchip and -xarch
Weitek 1164/1165-based FPU or no FPU	Kernel emulates floating-point instructions	Obsolete	Slow; not recommended	`-xchip=old` `-xarch=v7`
TI 8847-based FPU	TI 8847; controller from Fujitsu or LSI	Sun-4/1xx Sun-4/2xx Sun-4/3xx Sun-4/4xx SPARCstation 1 (4/60)	1989 Most SPARCstation 1 workstations have Weitek 3170	`-xchip=old` `-xarch=v7`
Weitek 3170-based FPU		SPARCstation 1 (4/60) SPARCstation 1+ (4/65)	1989, 1990	`-xchip=old` `-xarch=v7`
TI 602a		SPARCstation 2 (4/75)	1990	`-xchip=old` `-xarch=v7`
Weitek 3172-based FPU		SPARCstation SLC (4/20) SPARCstation IPC (4/40)	1990	`-xchip=old` `-xarch=v7`
Weitek 8601 or Fujitsu 86903	Integrated CPU and FPU	SPARCstation IPX (4/50) SPARCstation ELC (4/25)	1991 IPX uses 40 MHz CPU/FPU; ELC uses 33 MHz	`-xchip=old` `-xarch=v7`
Cypress 602	Resides on Mbus Module	SPARCserver 6xx	1991	`-xchip=old` `-xarch=v7`
TI TMS390S10 (STP1010)	microSPARC-I	SPARCstation LX SPARCclassic	1992 No FsMULd in hardware	`-xchip=micro` `-xarch=v8a`
Fujitsu 86904 (STP1012)	microSPARC-II	SPARCstation 4 and 5 SPARCstation Voyager	No FsMULd in hardware	`-xchip=micro2` `-xarch=v8a`
TI TMS390Z50 (STP1020A)	SuperSPARC-I	SPARCserver 6xx SPARCstation 10 SPARCstation 20 SPARCserver 1000 SPARCcenter 2000		`-xchip=super` `-xarch=v8`

TABLE B-1 SPARC Floating-Point Options *(Continued)*

FPU	Description or Processor Name	Appropriate for Machines	Notes	Optimum -xchip and -xarch

STP1021A	SuperSPARC-II	SPARCserver 6xx
		SPARCstation 10
		SPARCstation 20
		SPARCserver 1000
		SPARCcenter 2000
				`-xchip=super2`
				`-xarch=v8`

Ross RT620	hyperSPARC	SPARCstation 10/HSxx
		SPARCstation 20/HSxx
				`-xchip=hyper`
				`-xarch=v8`

Fujitsu 86907	TurboSPARC	SPARCstation 4 and 5		`-xchip=micro2`
				`-xarch=v8`

STP1030A	UltraSPARC I	Ultra-1, Ultra-2
		Ex000
			V9+VIS	`-xchip=ultra`
				`-xarch=v8plusa`

STP1031	UltraSPARC II	Ultra-2, E450
		Ultra-30, Ultra-60, Ultra-80, Ex500
		Ex000, E10000
			V9+VIS	`-xchip=ultra2`
				`-xarch=v8plusa`

SME1040	UltraSPARC IIi	Ultra-5, Ultra-10	V9+VIS	`-xchip=ultra2i`
				`-xarch=v8plusa`

	UltraSPARC IIe	Sun Blade 100	V9+VIS	`-xchip=ultra2e`
				`-xarch=v8plusa`

	UltraSPARC III	Sun Blade 1000	V9+VIS	`-xchip=ultra3`
				`-xarch=v8plusa`

The last column in the preceding table shows the compiler flags to use to obtain the fastest code for each FPU. These flags control two independent attributes of code generation: the `-xarch` flag determines the instruction set the compiler may use, and the `-xchip` flag determines the assumptions the compiler will make about a processor's performance characteristics in scheduling the code. Because all SPARC floating-point units implement at least the floating-point instruction set defined in the *SPARC Architecture Manual* Version 7, a program compiled with `-xarch=v7` will run on any SPARC system, although it may not take full advantage of the features of later processors. Likewise, a program compiled with a particular `-xchip` value will run on any SPARC system that supports the instruction set specified with `-xarch`, but it may run more slowly on systems with processors other than the one specified.

The floating-point units listed in the table preceding the microSPARC-I implement the floating-point instruction set defined in the *SPARC Architecture Manual* Version 7. Programs that must run on systems with these FPUs should be compiled with `-xarch=v7`. The compilers make no special assumptions regarding the performance characteristics of these processors, so they all share the single `-xchip` option

`-xchip=old`. (Not all of the systems listed in TABLE B-1 are still supported by Forte Developer compilers; they are listed solely for historical purposes. Refer to the appropriate version of the Numerical Computation Guide for the code generation flags to use with compilers supporting these systems.)

The microSPARC-I and microSPARC-II floating-point units implement the floating-point instruction set defined in the *SPARC Architecture Manual* Version 8 except for the FsMULd and quad precision instructions. Programs compiled with `-xarch=v8` will run on systems with these processors, but because unimplemented floating-point instructions must be emulated by the system kernel, programs that use FsMULd extensively (such as Fortran programs that perform a lot of single precision complex arithmetic), may encounter severe performance degradation. To avoid this, compile programs for systems with these processors with `-xarch=v8a`.

The SuperSPARC-I, SuperSPARC-II, hyperSPARC, and TurboSPARC floating-point units implement the floating-point instruction set defined in the *SPARC Architecture Manual* Version 8 except for the quad precision instructions. To get the best performance on systems with these processors, compile with `-xarch=v8`.

The UltraSPARC I, UltraSPARC II, UltraSPARC IIe, UltraSPARC IIi, and UltraSPARC III floating-point units implement the floating-point instruction set defined in the *SPARC Architecture Manual* Version 9 except for the quad precision instructions; in particular, they provide 32 double precision floating-point registers. To allow the compiler to use these registers, compile with `-xarch=v8plus` (for programs that run under a 32-bit OS) or `-xarch=v9` (for programs that run under a 64-bit OS). These processors also provide extensions to the standard instruction set. The additional instructions, known as the Visual Instruction Set or VIS, are rarely generated automatically by the compilers, but they may be used in assembly code. Therefore, to take full advantage of the instruction set these processors support, use `-xarch=v8plusa` (32-bit) or `-xarch=v9a` (64-bit).

The `-xarch` and `-xchip` options can be specified simultaneously using the `-xtarget` macro option. (That is, the `-xtarget` flag simply expands to a suitable combination of `-xarch`, `-xchip`, and `-xcache` flags.) The default code generation option is `-xtarget=generic`. See the `cc`(1), `CC`(1), and `f95`(1) man pages and the compiler manuals for more information including a complete list of `-xarch`, `-xchip`, and `-xtarget` values. Additional `-xarch` information is provided in the *Fortran User's Guide*, *C User's Guide*, and *C++ User's Guide*.

Floating-Point Status Register and Queue

All SPARC floating-point units, regardless of which version of the SPARC architecture they implement, provide a floating-point status register (FSR) that contains status and control bits associated with the FPU. All SPARC FPUs that implement deferred floating-point traps provide a floating-point queue (FQ) that contains information about currently executing floating-point instructions. The FSR can be accessed by user software to detect floating-point exceptions that have occurred and to control rounding direction, trapping, and nonstandard arithmetic modes. The FQ is used by the operating system kernel to process floating-point traps and is normally invisible to user software.

Software accesses the floating-point status register via `STFSR` and `LDFSR` instructions that store the FSR in memory and load it from memory, respectively. In SPARC assembly language, these instructions are written as follows:

```
        st      %fsr, [addr]  ! store FSR at specified address
        ld      [addr], %fsr  ! load FSR from specified address
```

The inline template file `libm.il` located in the directory containing the libraries supplied with the Forte Developer compilers contains examples showing the use of `STFSR` and `LDFSR` instructions.

FIGURE B-1 shows the layout of bit fields in the floating-point status register.

RD	res	TEM	NS	res	ver	ftt	qne	res	fcc	aexc	cexc
31:30	29:28	27:23	22	21:20	19:17	16:14	13	12	11:10	9:5	4:0

FIGURE B-1 SPARC Floating-Point Status Register

In versions 7 and 8 of the SPARC architecture, the FSR occupies 32 bits as shown. In version 9, the FSR is extended to 64 bits, of which the lower 32 match the figure; the upper 32 are largely unused, containing only three additional floating point condition code fields.

Here `res` refers to bits that are reserved, `ver` is a read-only field that identifies the version of the FPU, and `ftt` and `qne` are used by the system when it processes floating-point traps. The remaining fields are described in the following table.

TABLE B-2 Floating-Point Status Register Fields

Field	Contains
RM	rounding direction mode
TEM	trap enable modes
NS	nonstandard mode
fcc	floating point condition code
aexc	accrued exception flags
cexc	current exception flags

The `RM` field holds two bits that specify the rounding direction for floating-point operations. The `NS` bit enables nonstandard arithmetic mode on SPARC FPUs that implement it; on others, this bit is ignored. The `fcc` field holds floating-point condition codes generated by floating-point compare instructions and used by branch and conditional move operations. Finally, the `TEM`, `aexc`, and `cexc` fields contain five bits that control trapping and record accrued and current exception flags for each of the five IEEE 754 floating-point exceptions. These fields are subdivided as shown in TABLE B-3.

TABLE B-3 Exception Handling Fields

Field	**Corresponding bits in register**				
`TEM`, trap enable modes	NVM 27	OFM 26	UFM 25	DZM 24	NXM 23
`aexc`, accrued exception flags	nva 9	ofa 8	ufa 7	dza 6	nxa 5
`cexc`, current exception flags	nvc 4	ofc 3	ufc 2	dzc 1	nxc 0

(The symbols NV, OF, UF, DZ, and NX above stand for the invalid operation, overflow, underflow, division-by-zero, and inexact exceptions respectively.)

Special Cases Requiring Software Support

In most cases, SPARC floating-point units execute instructions completely in hardware without requiring software support. There are four situations, however, when the hardware will not successfully complete a floating-point instruction:

- The floating-point unit is disabled.
- The instruction is not implemented by the hardware (such as `fsqrt[sd]` on Weitek 1164/1165-based FPUs, `fsmuld` on microSPARC-I and microSPARC-II FPUs, or quad precision instructions on any SPARC FPU).
- The hardware is unable to deliver the correct result for the instruction's operands.
- The instruction would cause an IEEE 754 floating-point exception and that exception's trap is enabled.

In each situation, the initial response is the same: the process "traps" to the system kernel, which determines the cause of the trap and takes the appropriate action. (The term "trap" refers to an interruption of the normal flow of control.) In the first three situations, the kernel emulates the trapping instruction in software. Note that the emulated instruction can also incur an exception whose trap is enabled.

In the first three situations above, if the emulated instruction does not incur an IEEE floating-point exception whose trap is enabled, the kernel completes the instruction. If the instruction is a floating-point compare, the kernel updates the condition codes to reflect the result; if the instruction is an arithmetic operation, it delivers the appropriate result to the destination register. It also updates the current exception flags to reflect any (untrapped) exceptions raised by the instruction, and it "or"s those exceptions into the accrued exception flags. It then arranges to continue execution of the process at the point at which the trap was taken.

When an instruction executed by hardware or emulated by the kernel software incurs an IEEE floating-point exception whose trap is enabled, the instruction is not completed. The destination register, floating point condition codes, and accrued exception flags are unchanged, the current exception flags are set to reflect the particular exception that caused the trap, and the kernel sends a `SIGFPE` signal to the process.

The following pseudo-code summarizes the handling of floating-point traps. Note that the `aexc` field can normally only be cleared by software.

```
FPop provokes a trap;
if trap type is fp_disabled, unimplemented_FPop, or
  unfinished_FPop then
    emulate FPop;
texc ¨ all IEEE exceptions generated by FPop;
if (texc and TEM) = 0 then
    f[rd]  ¨ fp_result;  // if fpop is an arithmetic op
    fcc ¨ fcc_result;  // if fpop is a compare
    cexc ¨ texc;
    aexc ¨ (aexc or texc);
else
    cexc ¨ trapped IEEE exception generated by FPop;
    throw SIGFPE;
```

A program will encounter severe performance degradation when many floating-point instructions must be emulated by the kernel. The relative frequency with which this happens can depend on several factors including, of course, the type of trap.

Under normal circumstances, the `fp_disabled` trap should occur only once per process. The system kernel disables the floating-point unit when a process is first started, so the first floating-point operation executed by the process will cause a trap. After processing the trap, the kernel enables the floating-point unit, and it remains enabled for the duration of the process. (It is possible to disable the floating-point unit for the entire system, but this is not recommended and is done only for kernel or hardware debugging purposes.)

An `unimplemented_FPop` trap will obviously occur any time the floating-point unit encounters an instruction it does not implement. Since most current SPARC floating-point units implement at least the instruction set defined by the *SPARC Architecture Manual* Version 8 except for the quad precision instructions, and the Forte Developer compilers do not generate quad precision instructions, this type of trap should not occur on most systems. As mentioned above, two notable exceptions are the microSPARC-I and microSPARC-II processors, which do not implement the FsMULd instruction. To avoid `unimplemented_FPop` traps on these processors, compile programs with the `-xarch=v8a` option.

The remaining two trap types, `unfinished_FPop` and trapped IEEE exceptions, are usually associated with special computational situations involving NaNs, infinities, and subnormal numbers.

IEEE Floating-Point Exceptions, NaNs, and Infinities

When a floating-point instruction encounters an IEEE floating-point exception whose trap is enabled, the instruction is not completed; instead the system delivers a `SIGFPE` signal to the process. If the process has established a `SIGFPE` signal handler, that handler is invoked, and otherwise, the process aborts. Since trapping is most often enabled for the purpose of aborting the program when an exception occurs, either by invoking a signal handler that prints a message and terminates the program or by resorting to the system default behavior when no signal handler is installed, most programs do not incur many trapped IEEE floating-point exceptions. As described in Chapter 4, however, it is possible to arrange for a signal handler to supply a result for the trapping instruction and continue execution. Note that severe performance degradation can result if many floating-point exceptions are trapped and handled in this way.

Most SPARC floating-point units will also trap on at least some cases involving infinite or NaN operands or IEEE floating-point exceptions even when trapping is disabled or an instruction would not cause an exception whose trap is enabled. This happens when the hardware does not support such special cases; instead it generates an `unfinished_FPop` trap and leaves the kernel emulation software to complete the instruction. Different SPARC FPUs vary as to the conditions that result in an `unfinished_FPop` trap: for example, most early SPARC FPUs as well as the hyperSPARC FPU trap on all IEEE floating-point exceptions regardless of whether trapping is enabled, while UltraSPARC FPUs can trap “pessimistically” when a floating-point exception’s trap is enabled and the hardware is unable to determine whether or not an instruction would raise that exception. On the other hand, the SuperSPARC-I, SuperSPARC-II, TurboSPARC, microSPARC-I, and microSPARC-II FPUs handle all exceptional cases in hardware and never generate `unfinished_FPop` traps.

Since most `unfinished_FPop` traps occur in conjunction with floating-point exceptions, a program can avoid incurring an excessive number of these traps by employing exception handling (i.e., testing the exception flags, trapping and substituting results, or aborting on exceptions). Of course, care must be taken to balance the cost of handling exceptions with that of allowing exceptions to result in `unfinished_FPop` traps.

Subnormal Numbers and Nonstandard Arithmetic

The most common situations in which some SPARC floating-point units will trap with an `unfinished_FPop` involve subnormal numbers. Many SPARC FPUs will trap whenever a floating-point operation involves subnormal operands or must generate a nonzero subnormal result (i.e., a result that incurs gradual underflow). Because underflow is somewhat rare but difficult to program around, and because the accuracy of underflowed intermediate results often has little effect on the overall accuracy of the final result of a computation, the SPARC architecture includes a *nonstandard arithmetic mode* that provides a way for a user to avoid the performance degradation associated with `unfinished_FPop` traps involving subnormal numbers.

The SPARC architecture does not precisely define nonstandard arithmetic mode; it merely states that when this mode is enabled, processors that support it may produce results that do not conform to the IEEE 754 standard. However, all existing SPARC implementations that support this mode use it to disable gradual underflow, replacing all subnormal operands and results with zero. (There is one exception: Weitek 1164/1165 FPUs only flush subnormal results to zero in nonstandard mode, they do not treat subnormal operands as zero.)

Not all SPARC implementations provide a nonstandard mode. Specifically, the SuperSPARC-I, SuperSPARC-II, TurboSPARC, microSPARC-I, and microSPARC-II floating-point units handle subnormal operands and generate subnormal results entirely in hardware, so they do not need to support nonstandard arithmetic. (Any attempt to enable nonstandard mode on these processors is ignored.) Therefore, gradual underflow incurs no performance loss on these processors.

To determine whether gradual underflows are affecting the performance of a program, you should first determine whether underflows are occurring at all and then check how much system time is used by the program. To determine whether underflows are occurring, you can use the math library function `ieee_retrospective()` to see if the underflow exception flag is raised when the program exits. Fortran programs call `ieee_retrospective()` by default. C and C++

programs need to call `ieee_retrospective()` explicitly prior to exit. If any underflows have occurred, `ieee_retrospective()` prints a message similar to the following:

```
Note: IEEE floating-point exception flags raised:
 Inexact; Underflow;
See the Numerical Computation Guide, ieee_flags(3M)
```

If the program encounters underflows, you might want to determine how much system time the program is using by timing the program execution with the `time` command.

```
demo% /bin/time myprog > myprog.output
305.3 real      32.4 user      271.9 sys
```

If the system time (the third figure shown above) is unusually high, multiple underflows might be the cause. If so, and if the program does not depend on the accuracy of gradual underflow, you can enable nonstandard mode for better performance. There are two ways to do this. First, you can compile with the `-fns` flag (which is implied as part of the macros `-fast` and `-fnonstd`) to enable nonstandard mode at program startup. Second, the value-added math library `libsunmath` provides two functions to enable and disable nonstandard mode, respectively: calling `nonstandard_arithmetic()` enables nonstandard mode (if it is supported), while calling `standard_arithmetic()` restores IEEE behavior. The C and Fortran syntax for calling these functions is as follows:

C, C++	`nonstandard_arithmetic();`
	`standard_arithmetic();`

Fortran	`call nonstandard_arithmetic()`
	`call standard_arithmetic()`

Caution – Since nonstandard arithmetic mode defeats the accuracy benefits of gradual underflow, you should use it with caution. For more information about gradual underflow, see Chapter 2.

Nonstandard Arithmetic and Kernel Emulation

On SPARC floating-point units that implement nonstandard mode, enabling this mode causes the hardware to treat subnormal operands as zero and flush subnormal results to zero. The kernel software that is used to emulate trapped floating-point instructions, however, does not implement nonstandard mode, in part because the effect of this mode is undefined and implementation-dependent and because the added cost of handling gradual underflow is negligible compared to the cost of emulating a floating-point operation in software.

If a floating-point operation that would be affected by nonstandard mode is interrupted (for example, it has been issued but not completed when a context switch occurs or another floating-point instruction causes a trap), it will be emulated by kernel software using standard IEEE arithmetic. Thus, under unusual circumstances, a program running in nonstandard mode *might* produce slightly varying results depending on system load. This behavior has *not* been observed in practice. It would affect only those programs that are very sensitive to whether one particular operation out of millions is executed with gradual underflow or with abrupt underflow.

`fpversion`(1) Function — Finding Information About the FPU

The `fpversion` utility distributed with the compilers identifies the installed CPU and estimates the processor and system bus clock speeds. `fpversion` determines the CPU and FPU types by interpreting the identification information stored by the CPU and FPU. It estimates their clock speeds by timing a loop that executes simple instructions that run in a predictable amount of time. The loop is executed many times to increase the accuracy of the timing measurements. For this reason, `fpversion` is not instantaneous; it can take several seconds to run.

`fpversion` also reports the best `-xtarget` code generation option to use for the host system.

On an Ultra 4 workstation, `fpversion` displays information similar to the following. (There may be variations due to differences in timing or machine configuration.)

```
demo% fpversion
 A SPARC-based CPU is available.
 CPU’s clock rate appears to be approximately 461.1 MHz.
 Kernel says CPU’s clock rate is 480.0 MHz.
 Kernel says main memory’s clock rate is 120.0 MHz.

 Sun-4 floating-point controller version 0 found.
 An UltraSPARC chip is available.
 FPU's frequency appears to be approximately 492.7 MHz.

 Use “-xtarget=ultra2 -xcache=16/32/1:2048/64/1” code-
generation option.

 Hostid = hardware_host_id
```

See the `fpversion(1)` manual page for more information.

APPENDIX C

x86 Behavior and Implementation

This appendix discusses x86 and SPARC compatibility issues related to the floating-point units used in x86 platforms.

The hardware is 80386, 80486, and Pentium™ microprocessors from x86 and compatible microprocessors from other manufacturers. While great effort went into compatibility with the SPARC platform, several differences exist.

On x86:

- The floating-point registers are 80-bits wide. Because intermediate results of arithmetic computations can be in extended precision, computation results can differ. The `-fstore` flag minimizes these discrepancies. However, using the `-fstore` flag introduces a penalty in performance.
- Each time a single or double precision floating-point number is loaded or stored, a conversion to or from double extended precision occurs. Thus loads and stores of floating-point numbers can cause exceptions.
- Gradual underflow is implemented entirely in hardware. There is no nonstandard mode.
- The `fpversion` utility is not provided.
- The extended double format admits certain bit patterns that do not represent any floating point values (see TABLE 2-8). The hardware generally treats these "unsupported formats" like NaNs, but the math libraries are not consistent in their handling of such representations. Since these bit patterns are never generated by the hardware, they can only be created by invalid memory references (such as reading beyond the end of an array) or from explicit coercions of data in memory from one type to another (via C's `union` construct, for example). Therefore, in most numerical programs, these bit patterns do not arise.

APPENDIX **D**

What Every Computer Scientist Should Know About Floating-Point Arithmetic

Note – This appendix is an edited reprint of the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys. Copyright 1991, Association for Computing Machinery, Inc., reprinted by permission.

Abstract

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. This paper presents a tutorial on those aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with numerous examples of how computer builders can better support floating-point.

Categories and Subject Descriptors: (Primary) C.0 [Computer Systems Organization]: General —— *instruction set design*; D.3.4 [Programming Languages]: Processors —— *compilers, optimization*; G.1.0 [Numerical Analysis]: General —— *computer arithmetic, error analysis, numerical algorithms* (Secondary)

D.2.1 [Software Engineering]: Requirements/Specifications —— *languages*; D.3.4 Programming Languages]: Formal Definitions and Theory —— *semantics*; D.4.1 Operating Systems]: Process Management —— *synchronization*.

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow.

Introduction

Builders of computer systems often need information about floating-point arithmetic. There are, however, remarkably few sources of detailed information about it. One of the few books on the subject, *Floating-Point Computation* by Pat Sterbenz, is long out of print. This paper is a tutorial on those aspects of floating-point arithmetic (*floating-point* hereafter) that have a direct connection to systems building. It consists of three loosely connected parts. The first section, "Rounding Error" on page 162, discusses the implications of using different rounding strategies for the basic operations of addition, subtraction, multiplication and division. It also contains background information on the two methods of measuring rounding error, ulps and `relative error`. The second part discusses the IEEE floating-point standard, which is becoming rapidly accepted by commercial hardware manufacturers. Included in the IEEE standard is the rounding method for basic operations. The discussion of the standard draws on the material in the section "Rounding Error" on page 162. The third part discusses the connections between floating-point and the design of various aspects of computer systems. Topics include instruction set design, optimizing compilers and exception handling.

I have tried to avoid making statements about floating-point without also giving reasons why the statements are true, especially since the justifications involve nothing more complicated than elementary calculus. Those explanations that are not central to the main argument have been grouped into a section called "The Details," so that they can be skipped if desired. In particular, the proofs of many of the theorems appear in this section. The end of each proof is marked with the ∎ symbol. When a proof is not included, the ∎ appears immediately following the statement of the theorem.

Rounding Error

Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Although there are infinitely many integers, in most programs the result of integer computations can be stored in 32 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. This rounding error is the characteristic feature of floating-point computation. The section "Relative Error and Ulps" on page 165 describes how it is measured.

Since most floating-point calculations have rounding error anyway, does it matter if the basic arithmetic operations introduce a little bit more rounding error than necessary? That question is a main theme throughout this section. The section "Guard Digits" on page 166 discusses *guard* digits, a means of reducing the error when subtracting two nearby numbers. Guard digits were considered sufficiently important by IBM that in 1968 it added a guard digit to the double precision format in the System/360 architecture (single precision already had a guard digit), and retrofitted all existing machines in the field. Two examples are given to illustrate the utility of guard digits.

The IEEE standard goes further than just requiring the use of a guard digit. It gives an algorithm for addition, subtraction, multiplication, division and square root, and requires that implementations produce the same result as that algorithm. Thus, when a program is moved from one machine to another, the results of the basic operations will be the same in every bit if both machines support the IEEE standard. This greatly simplifies the porting of programs. Other uses of this precise specification are given in "Exactly Rounded Operations" on page 173.

Floating-point Formats

Several different representations of real numbers have been proposed, but by far the most widely used is the floating-point representation.[1] Floating-point representations have a base β (which is always assumed to be even) and a precision p. If $\beta = 10$ and $p = 3$, then the number 0.1 is represented as 1.00×10^{-1}. If $\beta = 2$ and $p = 24$, then the decimal number 0.1 cannot be represented exactly, but is approximately $1.10011001100110011001101 \times 2^{-4}$.

In general, a floating-point number will be represented as $\pm\, d.dd\ldots\, d \times \beta^e$, where $d.dd\ldots\, d$ is called the *significand*[2] and has p digits. More precisely $\pm\, d_0 \,.\, d_1\, d_2 \,\ldots\, d_{p-1} \times \beta^e$ represents the number

$$\pm\left(d_0 + d_1\beta^{-1} + \ldots + d_{p-1}\beta^{-(p-1)}\right)\beta^e, (0 \le d_i < \beta)\,. \qquad (1)$$

1. Examples of other representations are *floating slash* and *signed logarithm* [Matula and Kornerup 1985; Swartzlander and Alexopoulos 1975].

2. This term was introduced by Forsythe and Moler [1967], and has generally replaced the older term *mantissa*.

The term *floating-point number* will be used to mean a real number that can be exactly represented in the format under discussion. Two other parameters associated with floating-point representations are the largest and smallest allowable exponents, e_{max} and e_{min}. Since there are β^p possible significands, and $e_{max} - e_{min} + 1$ possible exponents, a floating-point number can be encoded in

$$[\log_2(e_{max} - e_{min} + 1)] + [\log_2(\beta^p)] + 1$$

bits, where the final +1 is for the sign bit. The precise encoding is not important for now.

There are two reasons why a real number might not be exactly representable as a floating-point number. The most common situation is illustrated by the decimal number 0.1. Although it has a finite decimal representation, in binary it has an infinite repeating representation. Thus when $\beta = 2$, the number 0.1 lies strictly between two floating-point numbers and is exactly representable by neither of them. A less common situation is that a real number is out of range, that is, its absolute value is larger than $\beta \times \beta^{e_{max}}$ or smaller than $1.0 \times \beta^{e_{min}}$. Most of this paper discusses issues due to the first reason. However, numbers that are out of range will be discussed in the sections "Infinity" on page 184 and "Denormalized Numbers" on page 187.

Floating-point representations are not necessarily unique. For example, both 0.01×10^1 and 1.00×10^{-1} represent 0.1. If the leading digit is nonzero ($d_0 \neq 0$ in equation (1) above), then the representation is said to be *normalized*. The floating-point number 1.00×10^{-1} is normalized, while 0.01×10^1 is not. When $\beta = 2$, $p = 3$, $e_{min} = -1$ and $e_{max} = 2$ there are 16 normalized floating-point numbers, as shown in FIGURE D-1. The bold hash marks correspond to numbers whose significand is 1.00. Requiring that a floating-point representation be normalized makes the representation unique. Unfortunately, this restriction makes it impossible to represent zero! A natural way to represent 0 is with $1.0 \times \beta^{e_{min} - 1}$, since this preserves the fact that the numerical ordering of nonnegative real numbers corresponds to the lexicographic ordering of their floating-point representations.[1] When the exponent is stored in a k bit field, that means that only $2^k - 1$ values are available for use as exponents, since one must be reserved to represent 0.

Note that the $\times$ in a floating-point number is part of the notation, and different from a floating-point multiply operation. The meaning of the $\times$ symbol should be clear from the context. For example, the expression $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$ involves only a single floating-point multiplication.

1. This assumes the usual arrangement where the exponent is stored to the left of the significand.

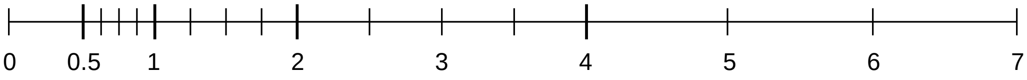


FIGURE D-1 Normalized Numbers When $\beta = 2$, $p = 3$, $e_{min} = -1$, $e_{max} = 2$

Relative Error and Ulps

Since rounding error is inherent in floating-point computation, it is important to have a way to measure this error. Consider the floating-point format with $\beta = 10$ and $p = 3$, which will be used throughout this section. If the result of a floating-point computation is 3.12×10^{-2}, and the answer when computed to infinite precision is .0314, it is clear that this is in error by 2 units in the last place. Similarly, if the real number .0314159 is represented as 3.14×10^{-2}, then it is in error by .159 units in the last place. In general, if the floating-point number $d.d\ldots d \times \beta^e$ is used to represent z, then it is in error by $|d.d\ldots d - (z/\beta^e)|\beta^{p-1}$ units in the last place.[1, 2] The term *ulps* will be used as shorthand for "units in the last place." If the result of a calculation is the floating-point number nearest to the correct result, it still might be in error by as much as .5 ulp. Another way to measure the difference between a floating-point number and the real number it is approximating is *relative error*, which is simply the difference between the two numbers divided by the real number. For example the relative error committed when approximating 3.14159 by 3.14×10^0 is $.00159/3.14159 \approx .0005$.

To compute the relative error that corresponds to .5 ulp, observe that when a real number is approximated by the closest possible floating-point number $d.dd...dd \times \beta^e$, the error can be as large as $0.00...00\beta' \times \beta^e$, where β' is the digit $\beta/2$, there are p units in the significand of the floating-point number, and p units of 0 in the significand of the error. This error is $((\beta/2)\beta^{-p}) \times \beta^e$. Since numbers of the form $d.dd\ldots dd \times \beta^e$ all have the same absolute error, but have values that range between β^e and $\beta \times \beta^e$, the relative error ranges between $((\beta/2)\beta^{-p}) \times \beta^e/\beta^e$ and $((\beta/2)\beta^{-p}) \times \beta^e/\beta^{e+1}$. That is,

$$\frac{1}{2}\beta^{-p} \le \frac{1}{2}\text{ulp} \le \frac{\beta}{2}\beta^{-p} \qquad (2)$$

In particular, the relative error corresponding to .5 ulp can vary by a factor of β. This factor is called the *wobble*. Setting $\varepsilon = (\beta/2)\beta^{-p}$ to the largest of the bounds in (2) above, we can say that when a real number is rounded to the closest floating-point number, the relative error is always bounded by e, which is referred to as *machine epsilon*.

1. Unless the number z is larger than $\beta^{e_{max}}+1$ or smaller than $\beta^{e_{min}}$. Numbers which are out of range in this fashion will not be considered until further notice.

2. Let z' be the floating-point number that approximates z. Then $|d.d\ldots d - (z/\beta^e)|\beta^{p-1}$ is equivalent to $|z'-z|/\text{ulp}(z')$. A more accurate formula for measuring error is $|z'-z|/\text{ulp}(z)$. – Ed.

In the example above, the relative error was $.00159/3.14159 \approx .0005$. In order to avoid such small numbers, the relative error is normally written as a factor times ε, which in this case is $\varepsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = .005$. Thus the relative error would be expressed as $(.00159/3.14159)/.005)\ \varepsilon \approx 0.1\varepsilon$.

To illustrate the difference between ulps and relative error, consider the real number $x = 12.35$. It is approximated by $\tilde{x} = 1.24 \times 10^1$. The error is 0.5 ulps, the relative error is 0.8ε. Next consider the computation $8\tilde{x}$. The exact value is $8x = 98.8$, while the computed value is $8\tilde{x} = 9.92 \times 10^1$. The error is now 4.0 ulps, but the relative error is still 0.8ε. The error measured in ulps is 8 times larger, even though the relative error is the same. In general, when the base is β, a fixed relative error expressed in ulps can wobble by a factor of up to β. And conversely, as equation (2) above shows, a fixed error of .5 ulps results in a relative error that can wobble by β.

The most natural way to measure rounding error is in ulps. For example rounding to the nearest floating-point number corresponds to an error of less than or equal to .5 ulp. However, when analyzing the rounding error caused by various formulas, relative error is a better measure. A good illustration of this is the analysis in the section "Proof" on page 206. Since ε can overestimate the effect of rounding to the nearest floating-point number by the wobble factor of β, error estimates of formulas will be tighter on machines with a small β.

When only the order of magnitude of rounding error is of interest, ulps and ε may be used interchangeably, since they differ by at most a factor of β. For example, when a floating-point number is in error by n ulps, that means that the number of contaminated digits is $\log_\beta n$. If the relative error in a computation is $n\varepsilon$, then

$$\text{contaminated digits} \approx \log_\beta n. \tag{3}$$

Guard Digits

One method of computing the difference between two floating-point numbers is to compute the difference exactly and then round it to the nearest floating-point number. This is very expensive if the operands differ greatly in size. Assuming $p = 3$, $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ would be calculated as

$x = 2.15 \times 10^{12}$
$y = .0000000000000000125 \times 10^{12}$
$x - y = 2.1499999999999999875 \times 10^{12}$

which rounds to 2.15×10^{12}. Rather than using all these digits, floating-point hardware normally operates on a fixed number of digits. Suppose that the number of digits kept is p, and that when the smaller operand is shifted right, digits are simply discarded (as opposed to rounding). Then $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ becomes

$x = 2.15 \times 10^{12}$
$y = 0.00 \times 10^{12}$
$x - y = 2.15 \times 10^{12}$

The answer is exactly the same as if the difference had been computed exactly and then rounded. Take another example: 10.1 – 9.93. This becomes

$x = 1.01 \times 10^1$
$y = 0.99 \times 10^1$
$x - y = .02 \times 10^1$

The correct answer is .17, so the computed difference is off by 30 ulps and is wrong in every digit! How bad can the error be?

Theorem 1

Using a floating-point format with parameters β *and p, and computing differences using p digits, the relative error of the result can be as large as* $\beta - 1$.

Proof

A relative error of $\beta - 1$ in the expression $x - y$ occurs when $x = 1.00\ldots0$ and $y = .\rho\rho\ldots\rho$, where $\rho = \beta - 1$. Here y has p digits (all equal to ρ). The exact difference is $x - y = \beta^{-p}$. However, when computing the answer using only p digits, the rightmost digit of y gets shifted off, and so the computed difference is β^{-p+1}. Thus the error is $\beta^{-p} - \beta^{-p+1} = \beta^{-p} (\beta - 1)$, and the relative error is $\beta^{-p}(\beta - 1)/\beta^{-p} = \beta - 1$. ∎

When $\beta=2$, the relative error can be as large as the result, and when $\beta=10$, it can be 9 times larger. Or to put it another way, when $\beta=2$, equation (3) shows that the number of contaminated digits is $\log_2(1/\varepsilon) = \log_2(2^p) = p$. That is, all of the p digits in the result are wrong! Suppose that one extra digit is added to guard against this situation (a *guard digit*). That is, the smaller number is truncated to $p + 1$ digits, and then the result of the subtraction is rounded to p digits. With a guard digit, the previous example becomes

$x = 1.010 \times 10^1$
$y = 0.993 \times 10^1$
$x - y = .017 \times 10^1$

and the answer is exact. With a single guard digit, the relative error of the result may be greater than ε, as in 110 – 8.59.

$x = 1.10 \times 10^2$
$y = .085 \times 10^2$
$x - y = 1.015 \times 10^2$

This rounds to 102, compared with the correct answer of 101.41, for a relative error of .006, which is greater than $\varepsilon = .005$. In general, the relative error of the result can be only slightly larger than ε. More precisely,

Theorem 2

If x and y are floating-point numbers in a format with parameters β *and p, and if subtraction is done with p + 1 digits (i.e. one guard digit), then the relative rounding error in the result is less than* 2ε.

This theorem will be proven in "Rounding Error" on page 206. Addition is included in the above theorem since x and y can be positive or negative.

Cancellation

The last section can be summarized by saying that without a guard digit, the relative error committed when subtracting two nearby quantities can be very large. In other words, the evaluation of any expression containing a subtraction (or an addition of quantities with opposite signs) could result in a relative error so large that *all* the digits are meaningless (Theorem 1). When subtracting nearby quantities, the most significant digits in the operands match and cancel each other. There are two kinds of cancellation: catastrophic and benign.

Catastrophic cancellation occurs when the operands are subject to rounding errors. For example in the quadratic formula, the expression $b^2 - 4ac$ occurs. The quantities b^2 and $4ac$ are subject to rounding errors since they are the results of floating-point multiplications. Suppose that they are rounded to the nearest floating-point number, and so are accurate to within .5 ulp. When they are subtracted, cancellation can cause many of the accurate digits to disappear, leaving behind mainly digits contaminated by rounding error. Hence the difference might have an error of many ulps. For example, consider $b = 3.34$, $a = 1.22$, and $c = 2.28$. The exact value of $b^2 - 4ac$ is .0292. But b^2 rounds to 11.2 and $4ac$ rounds to 11.1, hence the final answer is .1 which is an error by 70 ulps, even though 11.2 - 11.1 is exactly equal to .1[1]. The subtraction did not introduce any error, but rather exposed the error introduced in the earlier multiplications.

Benign cancellation occurs when subtracting exactly known quantities. If x and y have no rounding error, then by Theorem 2 if the subtraction is done with a guard digit, the difference x-y has a very small relative error (less than 2ε).

1. 700, not 70. Since .1 - .0292 = .0708, the error in terms of ulp(0.0292) is 708 ulps. – Ed.

A formula that exhibits catastrophic cancellation can sometimes be rearranged to eliminate the problem. Again consider the quadratic formula

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{4}$$

When $b^2 » ac$, then $b^2 - 4ac$ does not involve a cancellation and

$$\sqrt{b^2 - 4ac} \approx |b| .$$

But the other addition (subtraction) in one of the formulas will have a catastrophic cancellation. To avoid this, multiply the numerator and denominator of r_1 by

$$-b - \sqrt{b^2 - 4ac}$$

(and similarly for r_2) to obtain

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{5}$$

If $b^2 » ac$ and $b > 0$, then computing r_1 using formula (4) will involve a cancellation. Therefore, use formula (5) for computing r_1 and (4) for r_2. On the other hand, if $b < 0$, use (4) for computing r_1 and (5) for r_2.

The expression x^2 - y^2 is another formula that exhibits catastrophic cancellation. It is more accurate to evaluate it as $(x - y)(x + y)$.[1] Unlike the quadratic formula, this improved form still has a subtraction, but it is a benign cancellation of quantities without rounding error, not a catastrophic one. By Theorem 2, the relative error in $x - y$ is at most 2ε. The same is true of $x + y$. Multiplying two quantities with a small relative error results in a product with a small relative error (see the section "Rounding Error" on page 206).

In order to avoid confusion between exact and computed values, the following notation is used. Whereas $x - y$ denotes the exact difference of x and y, $x \ominus y$ denotes the computed difference (i.e., with rounding error). Similarly $\oplus$, $\otimes$, and $\oslash$ denote computed addition, multiplication, and division, respectively. All caps indicate the computed value of a function, as in `LN(x)` or `SQRT(x)`. Lowercase functions and traditional mathematical notation denote their exact values as in $\ln(x)$ and $\sqrt{x}$.

Although $(x \ominus y) \otimes (x \oplus y)$ is an excellent approximation to $x^2 - y^2$, the floating-point numbers x and y might themselves be approximations to some true quantities $\hat{x}$ and $\hat{y}$. For example, $\hat{x}$ and $\hat{y}$ might be exactly known decimal numbers that

1. Although the expression $(x - y)(x + y)$ does not cause a catastrophic cancellation, it is slightly less accurate than $x^2 - y^2$ if $x » y$ or $x « y$. In this case, $(x - y)(x + y)$ has three rounding errors, but $x^2 - y^2$ has only two since the rounding error committed when computing the smaller of x^2 and y^2 does not affect the final subtraction.

cannot be expressed exactly in binary. In this case, even though $x \ominus y$ is a good approximation to $x - y$, it can have a huge relative error compared to the true expression $\hat{x} - \hat{y}$, and so the advantage of $(x + y)(x - y)$ over $x^2 - y^2$ is not as dramatic. Since computing $(x + y)(x - y)$ is about the same amount of work as computing $x^2 - y^2$, it is clearly the preferred form in this case. In general, however, replacing a catastrophic cancellation by a benign one is not worthwhile if the expense is large, because the input is often (but not always) an approximation. But eliminating a cancellation entirely (as in the quadratic formula) is worthwhile even if the data are not exact. Throughout this paper, it will be assumed that the floating-point inputs to an algorithm are exact and that the results are computed as accurately as possible.

The expression $x^2 - y^2$ is more accurate when rewritten as $(x - y)(x + y)$ because a catastrophic cancellation is replaced with a benign one. We next present more interesting examples of formulas exhibiting catastrophic cancellation that can be rewritten to exhibit only benign cancellation.

The area of a triangle can be expressed directly in terms of the lengths of its sides a, b, and c as

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{ where } s = (a+b+c)/2 \tag{6}$$

(Suppose the triangle is very flat; that is, $a \approx b + c$. Then $s \approx a$, and the term $(s - a)$ in formula (6) subtracts two nearby numbers, one of which may have rounding error. For example, if $a = 9.0$, $b = c = 4.53$, the correct value of s is 9.03 and A is 2.342.... Even though the computed value of s (9.05) is in error by only 2 ulps, the computed value of A is 3.04, an error of 70 ulps.

There is a way to rewrite formula (6) so that it will return accurate results even for flat triangles [Kahan 1986]. It is

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, a \ge b \ge c \tag{7}$$

If a, b, and c do not satisfy $a \ge b \ge c$, rename them before applying (7). It is straightforward to check that the right-hand sides of (6) and (7) are algebraically identical. Using the values of a, b, and c above gives a computed area of 2.35, which is 1 ulp in error and much more accurate than the first formula.

Although formula (7) is much more accurate than (6) for this example, it would be nice to know how well (7) performs in general.

Theorem 3

The rounding error incurred when using (7) *to compute the area of a triangle is at most* 11ε*, provided that subtraction is performed with a guard digit,* $e \le .005$*, and that square roots are computed to within* $1/2$ ulp.

The condition that $e < .005$ is met in virtually every actual floating-point system. For example when $\beta = 2$, $p \ge 8$ ensures that $e < .005$, and when $\beta = 10$, $p \ge 3$ is enough.

In statements like Theorem 3 that discuss the relative error of an expression, it is understood that the expression is computed using floating-point arithmetic. In particular, the relative error is actually of the expression

$$\texttt{SQRT}((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4 \qquad (8)$$

Because of the cumbersome nature of (8), in the statement of theorems we will usually say *the computed value of E* rather than writing out E with circle notation.

Error bounds are usually too pessimistic. In the numerical example given above, the computed value of (7) is 2.35, compared with a true value of 2.34216 for a relative error of 0.7ε, which is much less than 11ε. The main reason for computing error bounds is not to get precise bounds but rather to verify that the formula does not contain numerical problems.

A final example of an expression that can be rewritten to use benign cancellation is $(1 + x)^n$, where $x \ll 1$. This expression arises in financial calculations. Consider depositing $100 every day into a bank account that earns an annual interest rate of 6%, compounded daily. If $n = 365$ and $i = .06$, the amount of money accumulated at the end of one year is

$$100\frac{(1 + i/n)^n - 1}{i/n}$$

dollars. If this is computed using $\beta = 2$ and $p = 24$, the result is $37615.45 compared to the exact answer of $37614.05, a discrepancy of $1.40. The reason for the problem is easy to see. The expression $1 + i/n$ involves adding 1 to .0001643836, so the low order bits of i/n are lost. This rounding error is amplified when $1 + i/n$ is raised to the nth power.

The troublesome expression $(1 + i/n)^n$ can be rewritten as $e^{n\ln(1 + i/n)}$, where now the problem is to compute $\ln(1 + x)$ for small x. One approach is to use the approximation $\ln(1 + x) \approx x$, in which case the payment becomes $37617.26, which is off by $3.21 and even less accurate than the obvious formula. But there is a way to compute $\ln(1 + x)$ very accurately, as Theorem 4 shows [Hewlett-Packard 1982]. This formula yields $37614.07, accurate to within two cents!

Theorem 4 assumes that `LN(x)` approximates ln(x) to within 1/2 ulp. The problem it solves is that when x is small, `LN`($1 \oplus x$) is not close to ln($1 + x$) because $1 \oplus x$ has lost the information in the low order bits of x. That is, the computed value of ln($1 + x$) is not close to its actual value when $x \ll 1$.

Theorem 4

If ln(1 + x) is computed using the formula

$$\ln(1+x) = \begin{cases} x & \text{for } 1 \oplus x = 1 \\ \dfrac{x\ln(1+x)}{(1+x)-1} & \text{for } 1 \oplus x \neq 1 \end{cases}$$

the relative error is at most 5ε when $0 \le x < 3/4$, provided subtraction is performed with a guard digit, e < 0.1, and ln is computed to within 1/2 ulp.

This formula will work for any value of x but is only interesting for $x \ll 1$, which is where catastrophic cancellation occurs in the naive formula ln($1 + x$). Although the formula may seem mysterious, there is a simple explanation for why it works. Write ln($1 + x$) as

$$x\left(\frac{\ln(1+x)}{x}\right) = x\mu(x).$$

The left hand factor can be computed exactly, but the right hand factor $\mu(x) = \ln(1 + x)/x$ will suffer a large rounding error when adding 1 to x. However, μ is almost constant, since $\ln(1 + x) \approx x$. So changing x slightly will not introduce much error. In other words, if $\tilde{x} \approx x$, computing $x\mu(\tilde{x})$ will be a good approximation to $x\mu(x) = \ln(1 + x)$. Is there a value for $\tilde{x}$ for which $\tilde{x}$ and $\tilde{x} + 1$ can be computed accurately? There is; namely $\tilde{x} = (1 \oplus x) \ominus 1$, because then $1 + \tilde{x}$ is exactly equal to $1 \oplus x$.

The results of this section can be summarized by saying that a guard digit guarantees accuracy when nearby precisely known quantities are subtracted (benign cancellation). Sometimes a formula that gives inaccurate results can be rewritten to have much higher numerical accuracy by using benign cancellation; however, the procedure only works if subtraction is performed using a guard digit. The price of a guard digit is not high, because it merely requires making the adder one bit wider. For a 54 bit double precision adder, the additional cost is less than 2%. For this price, you gain the ability to run many algorithms such as formula (6) for computing the area of a triangle and the expression ln($1 + x$). Although most modern computers have a guard digit, there are a few (such as Cray systems) that do not.

Exactly Rounded Operations

When floating-point operations are done with a guard digit, they are not as accurate as if they were computed exactly then rounded to the nearest floating-point number. Operations performed in this manner will be called *exactly rounded*.[1] The example immediately preceding Theorem 2 shows that a single guard digit will not always give exactly rounded results. The previous section gave several examples of algorithms that require a guard digit in order to work properly. This section gives examples of algorithms that require exact rounding.

So far, the definition of rounding has not been given. Rounding is straightforward, with the exception of how to round halfway cases; for example, should 12.5 round to 12 or 13? One school of thought divides the 10 digits in half, letting {0, 1, 2, 3, 4} round down, and {5, 6, 7, 8, 9} round up; thus 12.5 would round to 13. This is how rounding works on Digital Equipment Corporation's VAX computers. Another school of thought says that since numbers ending in 5 are halfway between two possible roundings, they should round down half the time and round up the other half. One way of obtaining this 50% behavior to require that the rounded result have its least significant digit be even. Thus 12.5 rounds to 12 rather than 13 because 2 is even. Which of these methods is best, round up or round to even? Reiser and Knuth [1975] offer the following reason for preferring round to even.

Theorem 5

Let x and y be floating-point numbers, and define $x_0 = x$, $x_1 = (x_0 \ominus y) \oplus y, \ldots, x_n = (x_{n-1} \ominus y) \oplus y$. If $\oplus$ and $\ominus$ are exactly rounded using round to even, then either $x_n = x$ for all n or $x_n = x_1$ for all $n \geq 1$. ▮

To clarify this result, consider $\beta = 10$, $p = 3$ and let $x = 1.00$, $y = -.555$. When rounding up, the sequence becomes

$$x_0 \ominus y = 1.56,\ x_1 = 1.56 \ominus .555 = 1.01,\ x_1 \ominus y = 1.01 \oplus .555 = 1.57,$$

and each successive value of x_n increases by .01, until $x_n = 9.45$ ($n \leq 845$)[2]. Under round to even, x_n is always 1.00. This example suggests that when using the round up rule, computations can gradually drift upward, whereas when using round to even the theorem says this cannot happen. Throughout the rest of this paper, round to even will be used.

One application of exact rounding occurs in multiple precision arithmetic. There are two basic approaches to higher precision. One approach represents floating-point numbers using a very large significand, which is stored in an array of words, and codes the routines for manipulating these numbers in assembly language. The

1. Also commonly referred to as *correctly rounded*. – Ed.

2. When n = 845, x_n= 9.45, x_n + 0.555 = 10.0, and 10.0 - 0.555 = 9.45. Therefore, $x_n = x_{845}$ for $n > 845$.

second approach represents higher precision floating-point numbers as an array of ordinary floating-point numbers, where adding the elements of the array in infinite precision recovers the high precision floating-point number. It is this second approach that will be discussed here. The advantage of using an array of floating-point numbers is that it can be coded portably in a high level language, but it requires exactly rounded arithmetic.

The key to multiplication in this system is representing a product xy as a sum, where each summand has the same precision as x and y. This can be done by splitting x and y. Writing $x = x_h + x_l$ and $y = y_h + y_l$, the exact product is

$$xy = x_h\, y_h + x_h\, y_l + x_l\, y_h + x_l\, y_l.$$

If x and y have p bit significands, the summands will also have p bit significands provided that x_l, x_h, y_h, y_l can be represented using $[p/2]$ bits. When p is even, it is easy to find a splitting. The number $x_0.x_1 \ldots x_{p-1}$ can be written as the sum of $x_0.x_1 \ldots x_{p/2-1}$ and $0.0 \ldots 0x_{p/2} \ldots x_{p-1}$. When p is odd, this simple splitting method will not work. An extra bit can, however, be gained by using negative numbers. For example, if $\beta = 2$, $p = 5$, and $x = .10111$, x can be split as $x_h = .11$ and $x_l = -.00001$. There is more than one way to split a number. A splitting method that is easy to compute is due to Dekker [1971], but it requires more than a single guard digit.

Theorem 6

Let p be the floating-point precision, with the restriction that p is even when $\beta > 2$, *and assume that floating-point operations are exactly rounded. Then if* $k = [p/2]$ *is half the precision (rounded up) and* $m = \beta^k + 1$, *x can be split as* $x = x_h + x_l$, *where*

$$x_h = (m \otimes x) \ominus (m \otimes x \ominus x),\ x_l = x \ominus x_h,$$

and each x_i *is representable using* $[p/2]$ *bits of precision.*

To see how this theorem works in an example, let $\beta = 10$, $p = 4$, $b = 3.476$, $a = 3.463$, and $c = 3.479$. Then $b^2 - ac$ rounded to the nearest floating-point number is .03480, while $b \otimes b = 12.08$, $a \otimes c = 12.05$, and so the computed value of $b^2 - ac$ is .03. This is an error of 480 ulps. Using Theorem 6 to write $b = 3.5 - .024$, $a = 3.5 - .037$, and $c = 3.5 - .021$, b^2 becomes $3.5^2 - 2 \times 3.5 \times .024 + .024^2$. Each summand is exact, so $b^2 = 12.25 - .168 + .000576$, where the sum is left unevaluated at this point. Similarly, $ac = 3.5^2 - (3.5 \times .037 + 3.5 \times .021) + .037 \times .021 = 12.25 - .2030 + .000777$. Finally, subtracting these two series term by term gives an estimate for $b^2 - ac$ of $0 \oplus .0350 \ominus .000201 = .03480$, which is identical to the exactly rounded result. To show that Theorem 6 really requires exact rounding, consider $p = 3$, $\beta = 2$, and $x = 7$. Then $m = 5$, $mx = 35$, and $m \otimes x = 32$. If subtraction is performed with a single guard digit, then $(m \otimes x) \ominus x = 28$. Therefore, $x_h = 4$ and $x_l = 3$, hence x_l is not representable with $[p/2] = 1$ bit.

As a final example of exact rounding, consider dividing m by 10. The result is a floating-point number that will in general not be equal to $m/10$. When $\beta = 2$, multiplying $m/10$ by 10 will restore m, provided exact rounding is being used. Actually, a more general fact (due to Kahan) is true. The proof is ingenious, but readers not interested in such details can skip ahead to section "The IEEE Standard" on page 176.

Theorem 7

When $\beta = 2$, if m and n are integers with $|m| < 2^{p-1}$ and n has the special form $n = 2^i + 2^j$, then $(m \oslash n) \otimes n = m$, provided floating-point operations are exactly rounded.

Proof

Scaling by a power of two is harmless, since it changes only the exponent, not the significand. If $q = m/n$, then scale n so that $2^{p-1} \le n < 2^p$ and scale m so that $1/2 < q < 1$. Thus, $2^{p-2} < m < 2^p$. Since m has p significant bits, it has at most one bit to the right of the binary point. Changing the sign of m is harmless, so assume that $q > 0$.

If $\bar{q} = m \oslash n$, to prove the theorem requires showing that

$$|n\bar{q} - m| \le \frac{1}{4} \tag{9}$$

That is because m has at most 1 bit right of the binary point, so $n\bar{q}$ will round to m. To deal with the halfway case when $|n\bar{q} - m| = 1/4$, note that since the initial unscaled m had $|m| < 2^{p-1}$, its low-order bit was 0, so the low-order bit of the scaled m is also 0. Thus, halfway cases will round to m.

Suppose that $q = .q_1q_2 \ldots$, and let $\hat{q} = .q_1q_2 \ldots q_p1$. To estimate $|n\bar{q} - m|$, first compute

$$|\hat{q} - q| = |N/2^{p+1} - m/n|,$$

where N is an odd integer. Since $n = 2^i + 2^j$ and $2^{p-1} \le n < 2^p$, it must be that $n = 2^{p-1} + 2^k$ for some $k \le p - 2$, and thus

$$|\hat{q} - q| = \left|\frac{nN - 2^{p+1}m}{n2^{p+1}}\right| = \left|\frac{(2^{p-1-k} + 1)N - 2^{p+1-k}m}{n2^{p+1-k}}\right|.$$

The numerator is an integer, and since N is odd, it is in fact an odd integer. Thus,

$$|\hat{q} - q| \geq 1/(n2^{p+1-k}).$$

Assume $q < \hat{q}$ (the case $q > \hat{q}$ is similar).[1] Then $n\bar{q} < m$, and

$$|m-n\bar{q}| = m-n\bar{q} = n(q-\bar{q}) = n(q-(\hat{q} - 2^{-p-1})) \leq n\left(2^{-p-1} - \frac{1}{n2^{p+1-k}}\right)$$
$$=(2^{p-1}+2^k)2^{-p-1}-2^{-p-1+k} = \frac{1}{4}$$

This establishes (9) and proves the theorem.[2] ∎

The theorem holds true for any base β, as long as $2^i + 2^j$ is replaced by $\beta^i + \beta^j$. As β gets larger, however, denominators of the form $\beta^i + \beta^j$ are farther and farther apart.

We are now in a position to answer the question, Does it matter if the basic arithmetic operations introduce a little more rounding error than necessary? The answer is that it does matter, because accurate basic operations enable us to prove that formulas are "correct" in the sense they have a small relative error. The section "Cancellation" on page 168 discussed several algorithms that require guard digits to produce correct results in this sense. If the input to those formulas are numbers representing imprecise measurements, however, the bounds of Theorems 3 and 4 become less interesting. The reason is that the benign cancellation $x - y$ can become catastrophic if x and y are only approximations to some measured quantity. But accurate operations are useful even in the face of inexact data, because they enable us to establish exact relationships like those discussed in Theorems 6 and 7. These are useful even if every floating-point variable is only an approximation to some actual value.

The IEEE Standard

There are two different IEEE standards for floating-point computation. IEEE 754 is a binary standard that requires β = 2, p = 24 for single precision and p = 53 for double precision [IEEE 1987]. It also specifies the precise layout of bits in a single and double precision. IEEE 854 allows either β = 2 or β = 10 and unlike 754, does not specify how floating-point numbers are encoded into bits [Cody et al. 1984]. It does not require a particular value for p, but instead it specifies constraints on the allowable values of p for single and double precision. The term *IEEE Standard* will be used when discussing properties common to both standards.

1. Notice that in binary, q cannot equal $\hat{q}$. – Ed.

2. Left as an exercise to the reader: extend the proof to bases other than 2. – Ed.

This section provides a tour of the IEEE standard. Each subsection discusses one aspect of the standard and why it was included. It is not the purpose of this paper to argue that the IEEE standard is the best possible floating-point standard but rather to accept the standard as given and provide an introduction to its use. For full details consult the standards themselves [IEEE 1987; Cody et al. 1984].

Formats and Operations

Base

It is clear why IEEE 854 allows $\beta = 10$. Base ten is how humans exchange and think about numbers. Using $\beta = 10$ is especially appropriate for calculators, where the result of each operation is displayed by the calculator in decimal.

There are several reasons why IEEE 854 requires that if the base is not 10, it must be 2. The section "Relative Error and Ulps" on page 165 mentioned one reason: the results of error analyses are much tighter when β is 2 because a rounding error of .5 ulp wobbles by a factor of β when computed as a relative error, and error analyses are almost always simpler when based on relative error. A related reason has to do with the effective precision for large bases. Consider $\beta = 16$, $p = 1$ compared to $\beta = 2$, $p = 4$. Both systems have 4 bits of significand. Consider the computation of 15/8. When $\beta = 2$, 15 is represented as 1.111×2^3, and 15/8 as 1.111×2^0. So 15/8 is exact. However, when $\beta = 16$, 15 is represented as $F \times 16^0$, where F is the hexadecimal digit for 15. But 15/8 is represented as 1×16^0, which has only one bit correct. In general, base 16 can lose up to 3 bits, so that a precision of p hexadecimal digits can have an effective precision as low as $4p - 3$ rather than $4p$ binary bits. Since large values of β have these problems, why did IBM choose $\beta = 16$ for its system/370? Only IBM knows for sure, but there are two possible reasons. The first is increased exponent range. Single precision on the system/370 has $\beta = 16$, $p = 6$. Hence the significand requires 24 bits. Since this must fit into 32 bits, this leaves 7 bits for the exponent and one for the sign bit. Thus the magnitude of representable numbers ranges from about 16^{-2^6} to about $16^{2^6} = 2^{2^8}$. To get a similar exponent range when $\beta = 2$ would require 9 bits of exponent, leaving only 22 bits for the significand. However, it was just pointed out that when $\beta = 16$, the effective precision can be as low as $4p - 3 = 21$ bits. Even worse, when $\beta = 2$ it is possible to gain an extra bit of precision (as explained later in this section), so the $\beta = 2$ machine has 23 bits of precision to compare with a range of 21 - 24 bits for the $\beta = 16$ machine.

Another possible explanation for choosing $\beta = 16$ has to do with shifting. When adding two floating-point numbers, if their exponents are different, one of the significands will have to be shifted to make the radix points line up, slowing down the operation. In the $\beta = 16$, $p = 1$ system, all the numbers between 1 and 15 have the same exponent, and so no shifting is required when adding any of the $\binom{15}{2} = 105$

possible pairs of distinct numbers from this set. However, in the $\beta = 2$, $p = 4$ system, these numbers have exponents ranging from 0 to 3, and shifting is required for 70 of the 105 pairs.

In most modern hardware, the performance gained by avoiding a shift for a subset of operands is negligible, and so the small wobble of $\beta = 2$ makes it the preferable base. Another advantage of using $\beta = 2$ is that there is a way to gain an extra bit of significance.[1] Since floating-point numbers are always normalized, the most significant bit of the significand is always 1, and there is no reason to waste a bit of storage representing it. Formats that use this trick are said to have a *hidden* bit. It was already pointed out in "Floating-point Formats" on page 163 that this requires a special convention for 0. The method given there was that an exponent of $e_{\min} - 1$ and a significand of all zeros represents not $1.0 \times 2^{e_{min} - 1}$, but rather 0.

IEEE 754 single precision is encoded in 32 bits using 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand. However, it uses a hidden bit, so the significand is 24 bits ($p = 24$), even though it is encoded using only 23 bits.

Precision

The IEEE standard defines four different precisions: single, double, single-extended, and double-extended. In IEEE 754, single and double precision correspond roughly to what most floating-point hardware provides. Single precision occupies a single 32 bit word, double precision two consecutive 32 bit words. Extended precision is a format that offers at least a little extra precision and exponent range (TABLE D-1).

TABLE D-1 IEEE 754 Format Parameters

	Format			
Parameter	**Single**	**Single-Extended**	**Double**	**Double-Extended**
p	24	$\geq$ 32	53	$\geq$ 64
$e_{\max}$	+127	$\geq$ 1023	+1023	$>$ 16383
$e_{\min}$	-126	$\leq$ -1022	-1022	$\leq$ -16382
Exponent width in bits	8	$\leq$ 11	11	$\geq$ 15
Format width in bits	32	$\geq$ 43	64	$\geq$ 79

1. This appears to have first been published by Goldberg [1967], although Knuth ([1981], page 211) attributes this idea to Konrad Zuse.

The IEEE standard only specifies a lower bound on how many extra bits extended precision provides. The minimum allowable double-extended format is sometimes referred to as *80-bit format*, even though the table shows it using 79 bits. The reason is that hardware implementations of extended precision normally do not use a hidden bit, and so would use 80 rather than 79 bits.[1]

The standard puts the most emphasis on extended precision, making no recommendation concerning double precision, but strongly recommending that *Implementations should support the extended format corresponding to the widest basic format supported, …*

One motivation for extended precision comes from calculators, which will often display 10 digits, but use 13 digits internally. By displaying only 10 of the 13 digits, the calculator appears to the user as a "black box" that computes exponentials, cosines, etc. to 10 digits of accuracy. For the calculator to compute functions like exp, log and cos to within 10 digits with reasonable efficiency, it needs a few extra digits to work with. It is not hard to find a simple rational expression that approximates log with an error of 500 units in the last place. Thus computing with 13 digits gives an answer correct to 10 digits. By keeping these extra 3 digits hidden, the calculator presents a simple model to the operator.

Extended precision in the IEEE standard serves a similar function. It enables libraries to efficiently compute quantities to within about .5 ulp in single (or double) precision, giving the user of those libraries a simple model, namely that each primitive operation, be it a simple multiply or an invocation of log, returns a value accurate to within about .5 ulp. However, when using extended precision, it is important to make sure that its use is transparent to the user. For example, on a calculator, if the internal representation of a displayed value is not rounded to the same precision as the display, then the result of further operations will depend on the hidden digits and appear unpredictable to the user.

To illustrate extended precision further, consider the problem of converting between IEEE 754 single precision and decimal. Ideally, single precision numbers will be printed with enough digits so that when the decimal number is read back in, the single precision number can be recovered. It turns out that 9 decimal digits are enough to recover a single precision binary number (see the section "Binary to Decimal Conversion" on page 215). When converting a decimal number back to its unique binary representation, a rounding error as small as 1 ulp is fatal, because it will give the wrong answer. Here is a situation where extended precision is vital for an efficient algorithm. When single-extended is available, a very straightforward method exists for converting a decimal number to a single precision binary one. First read in the 9 decimal digits as an integer N, ignoring the decimal point. From TABLE D-1, $p \geq 32$, and since $10^9 < 2^{32} \approx 4.3 \times 10^9$, N can be represented exactly in single-extended. Next find the appropriate power 10^P necessary to scale N. This will be a combination of the exponent of the decimal number, together with the position

1. According to Kahan, extended precision has 64 bits of significand because that was the widest precision across which carry propagation could be done on the Intel 8087 without increasing the cycle time [Kahan 1988].

of the (up until now) ignored decimal point. Compute $10^{|P|}$. If $|P| \le 13$, then this is also represented exactly, because $10^{13} = 2^{13}5^{13}$, and $5^{13} < 2^{32}$. Finally multiply (or divide if $p < 0$) N and $10^{|P|}$. If this last operation is done exactly, then the closest binary number is recovered. The section "Binary to Decimal Conversion" on page 215 shows how to do the last multiply (or divide) exactly. Thus for $|P| \le 13$, the use of the single-extended format enables 9-digit decimal numbers to be converted to the closest binary number (i.e. exactly rounded). If $|P| > 13$, then single-extended is not enough for the above algorithm to always compute the exactly rounded binary equivalent, but Coonen [1984] shows that it is enough to guarantee that the conversion of binary to decimal and back will recover the original binary number.

If double precision is supported, then the algorithm above would be run in double precision rather than single-extended, but to convert double precision to a 17-digit decimal number and back would require the double-extended format.

Exponent

Since the exponent can be positive or negative, some method must be chosen to represent its sign. Two common methods of representing signed numbers are sign/magnitude and two's complement. Sign/magnitude is the system used for the sign of the significand in the IEEE formats: one bit is used to hold the sign, the rest of the bits represent the magnitude of the number. The two's complement representation is often used in integer arithmetic. In this scheme, a number in the range $[-2^{p-1}, 2^{p-1} - 1]$ is represented by the smallest nonnegative number that is congruent to it modulo 2^p.

The IEEE binary standard does not use either of these methods to represent the exponent, but instead uses a *biased* representation. In the case of single precision, where the exponent is stored in 8 bits, the bias is 127 (for double precision it is 1023). What this means is that if $\bar{k}$ is the value of the exponent bits interpreted as an unsigned integer, then the exponent of the floating-point number is $\bar{k} - 127$. This is often called the *unbiased exponent* to distinguish from the biased exponent $\bar{k}$.

Referring to TABLE D-1, single precision has $e_{\max} = 127$ and $e_{\min} = -126$. The reason for having $|e_{\min}| < e_{\max}$ is so that the reciprocal of the smallest number $(1/2^{e_{\min}})$ will not overflow. Although it is true that the reciprocal of the largest number will underflow, underflow is usually less serious than overflow. The section "Base" on page 177 explained that $e_{\min} - 1$ is used for representing 0, and "Special Quantities" on page 182 will introduce a use for $e_{\max} + 1$. In IEEE single precision, this means that the biased exponents range between $e_{\min} - 1 = -127$ and $e_{\max} + 1 = 128$, whereas the unbiased exponents range between 0 and 255, which are exactly the nonnegative numbers that can be represented using 8 bits.

Operations

The IEEE standard requires that the result of addition, subtraction, multiplication and division be exactly rounded. That is, the result must be computed exactly and then rounded to the nearest floating-point number (using round to even). The section “Guard Digits” on page 166 pointed out that computing the exact difference or sum of two floating-point numbers can be very expensive when their exponents are substantially different. That section introduced guard digits, which provide a practical way of computing differences while guaranteeing that the relative error is small. However, computing with a single guard digit will not always give the same answer as computing the exact result and then rounding. By introducing a second guard digit and a third *sticky* bit, differences can be computed at only a little more cost than with a single guard digit, but the result is the same as if the difference were computed exactly and then rounded [Goldberg 1990]. Thus the standard can be implemented efficiently.

One reason for completely specifying the results of arithmetic operations is to improve the portability of software. When a program is moved between two machines and both support IEEE arithmetic, then if any intermediate result differs, it must be because of software bugs, not from differences in arithmetic. Another advantage of precise specification is that it makes it easier to reason about floating-point. Proofs about floating-point are hard enough, without having to deal with multiple cases arising from multiple kinds of arithmetic. Just as integer programs can be proven to be correct, so can floating-point programs, although what is proven in that case is that the rounding error of the result satisfies certain bounds. Theorem 4 is an example of such a proof. These proofs are made much easier when the operations being reasoned about are precisely specified. Once an algorithm is proven to be correct for IEEE arithmetic, it will work correctly on any machine supporting the IEEE standard.

Brown [1981] has proposed axioms for floating-point that include most of the existing floating-point hardware. However, proofs in this system cannot verify the algorithms of sections “Cancellation” on page 168 and “Exactly Rounded Operations” on page 173, which require features not present on all hardware. Furthermore, Brown’s axioms are more complex than simply defining operations to be performed exactly and then rounded. Thus proving theorems from Brown’s axioms is usually more difficult than proving them assuming operations are exactly rounded.

There is not complete agreement on what operations a floating-point standard should cover. In addition to the basic operations +, -, $\times$ and /, the IEEE standard also specifies that square root, remainder, and conversion between integer and floating-point be correctly rounded. It also requires that conversion between internal formats and decimal be correctly rounded (except for very large numbers). Kulisch and Miranker [1986] have proposed adding inner product to the list of operations that are precisely specified. They note that when inner products are computed in IEEE arithmetic, the final answer can be quite wrong. For example sums are a special case of inner products, and the sum $((2 \times 10^{-30} + 10^{30}) - 10^{30}) - 10^{-30}$ is exactly equal to

10^{-30}, but on a machine with IEEE arithmetic the computed result will be -10^{-30}. It is possible to compute inner products to within 1 ulp with less hardware than it takes to implement a fast multiplier [Kirchner and Kulish 1987].1 2

All the operations mentioned in the standard are required to be exactly rounded except conversion between decimal and binary. The reason is that efficient algorithms for exactly rounding all the operations are known, except conversion. For conversion, the best known efficient algorithms produce results that are slightly worse than exactly rounded ones [Coonen 1984].

The IEEE standard does not require transcendental functions to be exactly rounded because of the *table maker's dilemma*. To illustrate, suppose you are making a table of the exponential function to 4 places. Then exp(1.626) = 5.0835. Should this be rounded to 5.083 or 5.084? If exp(1.626) is computed more carefully, it becomes 5.08350. And then 5.083500. And then 5.0835000. Since exp is transcendental, this could go on arbitrarily long before distinguishing whether exp(1.626) is 5.083500…0*ddd* or 5.0834999…9*ddd*. Thus it is not practical to specify that the precision of transcendental functions be the same as if they were computed to infinite precision and then rounded. Another approach would be to specify transcendental functions algorithmically. But there does not appear to be a single algorithm that works well across all hardware architectures. Rational approximation, CORDIC,3 and large tables are three different techniques that are used for computing transcendentals on contemporary machines. Each is appropriate for a different class of hardware, and at present no single algorithm works acceptably over the wide range of current hardware.

Special Quantities

On some floating-point hardware every bit pattern represents a valid floating-point number. The IBM System/370 is an example of this. On the other hand, the VAX™ reserves some bit patterns to represent special numbers called *reserved operands*. This idea goes back to the CDC 6600, which had bit patterns for the special quantities `INDEFINITE` and `INFINITY`.

The IEEE standard continues in this tradition and has NaNs (*Not a Number*) and infinities. Without any special quantities, there is no good way to handle exceptional situations like taking the square root of a negative number, other than aborting computation. Under IBM System/370 FORTRAN, the default action in response to

1. Some arguments against including inner product as one of the basic operations are presented by Kahan and LeBlanc [1985].

2. Kirchner writes: It is possible to compute inner products to within 1 ulp in hardware in one partial product per clock cycle. The additionally needed hardware compares to the multiplier array needed anyway for that speed.

3. CORDIC is an acronym for Coordinate Rotation Digital Computer and is a method of computing transcendental functions that uses mostly shifts and adds (i.e., very few multiplications and divisions) [Walther 1971]. It is the method additionally needed hardware compares to the multiplier array needed anyway for that speed. d used on both the Intel 8087 and the Motorola 68881.

computing the square root of a negative number like -4 results in the printing of an error message. Since every bit pattern represents a valid number, the return value of square root must be some floating-point number. In the case of System/370 FORTRAN, $\sqrt{-4} = 2$ is returned. In IEEE arithmetic, a NaN is returned in this situation.

The IEEE standard specifies the following special values (see TABLE D-2): ± 0, denormalized numbers, ±∞ and NaNs (there is more than one NaN, as explained in the next section). These special values are all encoded with exponents of either $e_{\max} + 1$ or $e_{\min} - 1$ (it was already pointed out that 0 has an exponent of $e_{\min} - 1$).

TABLE D-2 IEEE 754 Special Values

Exponent	Fraction	Represents
$e = e_{\min} - 1$	$f = 0$	± 0
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$
$e_{\min} \leq e \leq e_{\max}$	—	$1.f \times 2^e$
$e = e_{\max} + 1$	$f = 0$	∞
$e = e_{\max} + 1$	$f \neq 0$	NaN

NaNs

Traditionally, the computation of 0/0 or $\sqrt{-1}$ has been treated as an unrecoverable error which causes a computation to halt. However, there are examples where it makes sense for a computation to continue in such a situation. Consider a subroutine that finds the zeros of a function *f*, say `zero(f)`. Traditionally, zero finders require the user to input an interval [*a*, *b*] on which the function is defined and over which the zero finder will search. That is, the subroutine is called as `zero(f, a, b)`. A more useful zero finder would not require the user to input this extra information. This more general zero finder is especially appropriate for calculators, where it is natural to simply key in a function, and awkward to then have to specify the domain. However, it is easy to see why most zero finders require a domain. The zero finder does its work by probing the function `f` at various values. If it probed for a value outside the domain of `f`, the code for `f` might well compute 0/0 or $\sqrt{-1}$, and the computation would halt, unnecessarily aborting the zero finding process.

This problem can be avoided by introducing a special value called NaN, and specifying that the computation of expressions like 0/0 and $\sqrt{-1}$ produce NaN, rather than halting. A list of some of the situations that can cause a NaN are given in TABLE D-3. Then when `zero(f)` probes outside the domain of `f`, the code for `f` will return NaN, and the zero finder can continue. That is, `zero(f)` is not "punished" for making an incorrect guess. With this example in mind, it is easy to see what the result of combining a NaN with an ordinary floating-point number should be.

Suppose that the final statement of `f` is `return(-b + sqrt(d))/(2*a)`. If $d < 0$, then `f` should return a NaN. Since $d < 0$, `sqrt(d)` is a NaN, and `-b + sqrt(d)` will be a NaN, if the sum of a NaN and any other number is a NaN. Similarly if one operand of a division operation is a NaN, the quotient should be a NaN. In general, whenever a NaN participates in a floating-point operation, the result is another NaN.

TABLE D-3 Operations That Produce a NaN

Operation	`NaN` Produced By
+	$\infty + (-\infty)$
$\times$	$0 \times \infty$
/	$0/0$, ∞/∞
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$ (when $x < 0$)

Another approach to writing a zero solver that doesn't require the user to input a domain is to use signals. The zero-finder could install a signal handler for floating-point exceptions. Then if `f` was evaluated outside its domain and raised an exception, control would be returned to the zero solver. The problem with this approach is that every language has a different method of handling signals (if it has a method at all), and so it has no hope of portability.

In IEEE 754, NaNs are often represented as floating-point numbers with the exponent $e_{\max} + 1$ and nonzero significands. Implementations are free to put system-dependent information into the significand. Thus there is not a unique NaN, but rather a whole family of NaNs. When a NaN and an ordinary floating-point number are combined, the result should be the same as the NaN operand. Thus if the result of a long computation is a NaN, the system-dependent information in the significand will be the information that was generated when the first NaN in the computation was generated. Actually, there is a caveat to the last statement. If both operands are NaNs, then the result will be one of those NaNs, but it might not be the NaN that was generated first.

Infinity

Just as NaNs provide a way to continue a computation when expressions like 0/0 or $\sqrt{-1}$ are encountered, infinities provide a way to continue when an overflow occurs. This is much safer than simply returning the largest representable number. As an example, consider computing $\sqrt{x^2 + y^2}$, when $\beta = 10$, $p = 3$, and $e_{\max} = 98$. If $x = 3 \times 10^{70}$ and $y = 4 \times 10^{70}$, then x^2 will overflow, and be replaced by 9.99×10^{98}. Similarly y^2, and $x^2 + y^2$ will each overflow in turn, and be replaced by 9.99×10^{98}. So the final result will be $\sqrt{9.99 \times 10^{98}} = 3.16 \times 10^{49}$, which is drastically wrong: the correct answer is 5×10^{70}. In IEEE arithmetic, the result of x^2 is ∞, as is y^2, $x^2 + y^2$ and $\sqrt{x^2 + y^2}$. So the final result is ∞, which is safer than returning an ordinary floating-point number that is nowhere near the correct answer.[1]

The division of 0 by 0 results in a NaN. A nonzero number divided by 0, however, returns infinity: $1/0 = \infty$, $-1/0 = -\infty$. The reason for the distinction is this: if $f(x) \to 0$ and $g(x) \to 0$ as x approaches some limit, then $f(x)/g(x)$ could have any value. For example, when $f(x) = \sin x$ and $g(x) = x$, then $f(x)/g(x) \to 1$ as $x \to 0$. But when $f(x) = 1 - \cos x$, $f(x)/g(x) \to 0$. When thinking of 0/0 as the limiting situation of a quotient of two very small numbers, 0/0 could represent anything. Thus in the IEEE standard, 0/0 results in a NaN. But when $c > 0$, $f(x) \to c$, *and* $g(x) \to 0$, then $f(x)/g(x) \to \pm\infty$, for any analytic functions f and g. If $g(x) < 0$ for small x, then $f(x)/g(x) \to -\infty$, otherwise the limit is $+\infty$. So the IEEE standard defines $c/0 = \pm\infty$, as long as $c \neq 0$. The sign of ∞ depends on the signs of c and 0 in the usual way, so that $-10/0 = -\infty$, and $-10/-0 = +\infty$. You can distinguish between getting ∞ because of overflow and getting ∞ because of division by zero by checking the status flags (which will be discussed in detail in section "Flags" on page 192). The overflow flag will be set in the first case, the division by zero flag in the second.

The rule for determining the result of an operation that has infinity as an operand is simple: replace infinity with a finite number x and take the limit as $x \to \infty$. Thus $3/\infty = 0$, because

$$\lim_{x \to \infty} 3/x = 0 .$$

Similarly, $4 - \infty = -\infty$, and $\sqrt{\infty} = \infty$. When the limit doesn't exist, the result is a NaN, so ∞/∞ will be a NaN (TABLE D-3 has additional examples). This agrees with the reasoning used to conclude that 0/0 should be a NaN.

When a subexpression evaluates to a NaN, the value of the entire expression is also a NaN. In the case of $\pm\infty$ however, the value of the expression might be an ordinary floating-point number because of rules like $1/\infty = 0$. Here is a practical example that makes use of the rules for infinity arithmetic. Consider computing the function $x/(x^2 + 1)$. This is a bad formula, because not only will it overflow when x is larger than $\sqrt{\beta}\beta^{e_{\max}/2}$, but infinity arithmetic will give the wrong answer because it will yield 0, rather than a number near $1/x$. However, $x/(x^2 + 1)$ can be rewritten as $1/(x + x^{-1})$. This improved expression will not overflow prematurely and because of infinity arithmetic will have the correct value when $x = 0$: $1/(0 + 0^{-1}) = 1/(0 + \infty) = 1/\infty = 0$. Without infinity arithmetic, the expression $1/(x + x^{-1})$ requires a test for $x = 0$, which not only adds extra instructions, but may also disrupt a pipeline. This example illustrates a general fact, namely that infinity arithmetic often avoids the need for special case checking; however, formulas need to be carefully inspected to make sure they do not have spurious behavior at infinity (as $x/(x^2 + 1)$ did).

1. Fine point: Although the default in IEEE arithmetic is to round overflowed numbers to ∞, it is possible to change the default (see "Rounding Modes" on page 191)

Signed Zero

Zero is represented by the exponent $e_{\min} - 1$ and a zero significand. Since the sign bit can take on two different values, there are two zeros, +0 and -0. If a distinction were made when comparing +0 and -0, simple tests like `if (x = 0)` would have very unpredictable behavior, depending on the sign of `x`. Thus the IEEE standard defines comparison so that +0 = -0, rather than -0 < +0. Although it would be possible always to ignore the sign of zero, the IEEE standard does not do so. When a multiplication or division involves a signed zero, the usual sign rules apply in computing the sign of the answer. Thus 3·(+0) = +0, and +0/-3 = -0. If zero did not have a sign, then the relation $1/(1/x) = x$ would fail to hold when $x = \pm\infty$. The reason is that $1/-\infty$ and $1/+\infty$ both result in 0, and 1/0 results in $+\infty$, the sign information having been lost. One way to restore the identity $1/(1/x) = x$ is to only have one kind of infinity, however that would result in the disastrous consequence of losing the sign of an overflowed quantity.

Another example of the use of signed zero concerns underflow and functions that have a discontinuity at 0, such as log. In IEEE arithmetic, it is natural to define log 0 = $-\infty$ and log x to be a NaN when $x < 0$. Suppose that x represents a small negative number that has underflowed to zero. Thanks to signed zero, x will be negative, so log can return a NaN. However, if there were no signed zero, the log function could not distinguish an underflowed negative number from 0, and would therefore have to return $-\infty$. Another example of a function with a discontinuity at zero is the signum function, which returns the sign of a number.

Probably the most interesting use of signed zero occurs in complex arithmetic. To take a simple example, consider the equation $\sqrt{1/z} = 1/(\sqrt{z})$. This is certainly true when $z \geq 0$. If $z = -1$, the obvious computation gives $\sqrt{1/(-1)} = \sqrt{-1} = i$ and $1/(\sqrt{-1}) = 1/i = -i$. Thus, $\sqrt{1/z} \neq 1/(\sqrt{z})$! The problem can be traced to the fact that square root is multi-valued, and there is no way to select the values so that it is continuous in the entire complex plane. However, square root is continuous if a *branch cut* consisting of all negative real numbers is excluded from consideration. This leaves the problem of what to do for the negative real numbers, which are of the form $-x + i0$, where $x > 0$. Signed zero provides a perfect way to resolve this problem. Numbers of the form $x + i(+0)$ have one sign $(i\sqrt{x})$ and numbers of the form $x + i(-0)$ on the other side of the branch cut have the other sign $(-i\sqrt{x})$. In fact, the natural formulas for computing $\sqrt{\ }$ will give these results.

Back to $\sqrt{1/z} = 1/(\sqrt{z})$. If $z =1 = -1 + i0$, then

$$1/z = 1/(-1 + i0) = [(-1- i0)]/[(-1 + i0)(-1 - i0)] = (-1 -- i0)/((-1)^2 - 0^2) = -1 + i(-0),$$

and so $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$, while $1/(\sqrt{z}) = 1/i = -i$. Thus IEEE arithmetic preserves this identity for all z. Some more sophisticated examples are given by Kahan [1987]. Although distinguishing between +0 and -0 has advantages, it can occasionally be confusing. For example, signed zero destroys the relation

$x = y \Leftrightarrow 1/x = 1/y$, which is false when $x = +0$ and $y = -0$. However, the IEEE committee decided that the advantages of utilizing the sign of zero outweighed the disadvantages.

Denormalized Numbers

Consider normalized floating-point numbers with $\beta = 10$, $p = 3$, and $e_{\min} = -98$. The numbers $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$ appear to be perfectly ordinary floating-point numbers, which are more than a factor of 10 larger than the smallest floating-point number 1.00×10^{-98}. They have a strange property, however: $x \ominus y = 0$ even though $x \neq y$! The reason is that $x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$ is too small to be represented as a normalized number, and so must be flushed to zero. How important is it to preserve the property

$$x = y \Leftrightarrow x - y = 0\ ? \tag{10}$$

It's very easy to imagine writing the code fragment, `if (x ≠ y) then z = 1/(x-y)`, and much later having a program fail due to a spurious division by zero. Tracking down bugs like this is frustrating and time consuming. On a more philosophical level, computer science textbooks often point out that even though it is currently impractical to prove large programs correct, designing programs with the idea of proving them often results in better code. For example, introducing invariants is quite useful, even if they aren't going to be used as part of a proof. Floating-point code is just like any other code: it helps to have provable facts on which to depend. For example, when analyzing formula (6), it was very helpful to know that $x/2 < y < 2x \Rightarrow x \ominus y = x - y$. Similarly, knowing that (10) is true makes writing reliable floating-point code easier. If it is only true for most numbers, it cannot be used to prove anything.

The IEEE standard uses denormalized[1] numbers, which guarantee (10), as well as other useful relations. They are the most controversial part of the standard and probably accounted for the long delay in getting 754 approved. Most high performance hardware that claims to be IEEE compatible does not support denormalized numbers directly, but rather traps when consuming or producing denormals, and leaves it to software to simulate the IEEE standard.[2] The idea behind denormalized numbers goes back to Goldberg [1967] and is very simple. When the exponent is $e_{\min}$, the significand does not have to be normalized, so that when $\beta = 10$, $p = 3$ and $e_{\min} = -98$, 1.00×10^{-98} is no longer the smallest floating-point number, because 0.98×10^{-98} is also a floating-point number.

1. They are called *subnormal* in 854, *denormal* in 754.

2. This is the cause of one of the most troublesome aspects of the standard. Programs that frequently underflow often run noticeably slower on hardware that uses software traps.

There is a small snag when $\beta = 2$ and a hidden bit is being used, since a number with an exponent of $e_{\min}$ will always have a significand greater than or equal to 1.0 because of the implicit leading bit. The solution is similar to that used to represent 0, and is summarized in TABLE D-2. The exponent $e_{\min}$ is used to represent denormals. More formally, if the bits in the significand field are b_1, b_2, …, b_{p-1}, and the value of the exponent is e, then when $e > e_{\min} - 1$, the number being represented is $1.b_1b_2\ldots b_{p-1} \times 2^e$ whereas when $e = e_{\min} - 1$, the number being represented is $0.b_1b_2\ldots b_{p-1} \times 2^{e+1}$. The +1 in the exponent is needed because denormals have an exponent of $e_{\min}$, not $e_{\min} - 1$.

Recall the example of $\beta = 10$, $p = 3$, $e_{\min} = -98$, $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$ presented at the beginning of this section. With denormals, $x - y$ does not flush to zero but is instead represented by the denormalized number $.6 \times 10^{-98}$. This behavior is called gradual *underflow*. It is easy to verify that (10) always holds when using gradual underflow.

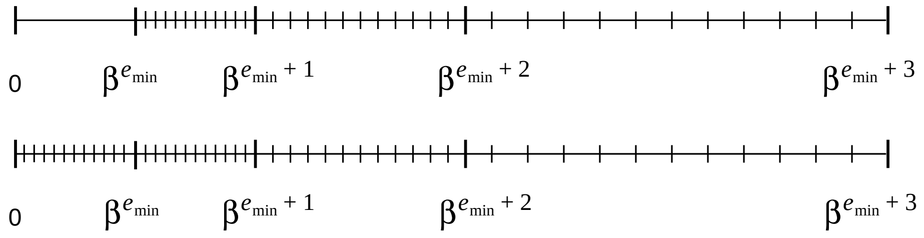


FIGURE D-2 Flush To Zero Compared With Gradual Underflow

FIGURE D-2 illustrates denormalized numbers. The top number line in the figure shows normalized floating-point numbers. Notice the gap between 0 and the smallest normalized number $1.0 \times \beta^{e_{min}}$. If the result of a floating-point calculation falls into this gulf, it is flushed to zero. The bottom number line shows what happens when denormals are added to the set of floating-point numbers. The "gulf" is filled in, and when the result of a calculation is less than $1.0 \times \beta^{e_{min}}$, it is represented by the nearest denormal. When denormalized numbers are added to the number line, the spacing between adjacent floating-point numbers varies in a regular way: adjacent spacings are either the same length or differ by a factor of β. Without denormals, the spacing abruptly changes from $\beta^{-p+1}\beta^{e_{\min}}$ to $\beta^{e_{\min}}$, which is a factor of β^{p-1}, rather than the orderly change by a factor of β. Because of this, many algorithms that can have large relative error for normalized numbers close to the underflow threshold are well-behaved in this range when gradual underflow is used.

Without gradual underflow, the simple expression x - y can have a very large relative error for normalized inputs, as was seen above for $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$. Large relative errors can happen even without cancellation, as the following example shows [Demmel 1984]. Consider dividing two complex numbers, $a + ib$ and $c + id$. The obvious formula

$$\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

suffers from the problem that if either component of the denominator $c + id$ is larger than $\sqrt{\beta}\beta^{e_{max}/2}$, the formula will overflow, even though the final result may be well within range. A better method of computing the quotients is to use Smith's formula:

$$\frac{a + ib}{c + id} = \begin{cases} \dfrac{a + b(d/c)}{c + d(d/c)} + i\dfrac{b - a(d/c)}{c + d(d/c)} & \text{if}(|d| < |c|) \\ \dfrac{b + a(c/d)}{d + c(c/d)} + i\dfrac{-a + b(c/d)}{d + c(c/d)} & \text{if}(|d| \geq |c|) \end{cases} \tag{11}$$

Applying Smith's formula to $(2 \cdot 10^{-98} + i10^{-98})/(4 \cdot 10^{-98} + i(2 \cdot 10^{-98}))$ gives the correct answer of 0.5 with gradual underflow. It yields 0.4 with flush to zero, an error of 100 ulps. It is typical for denormalized numbers to guarantee error bounds for arguments all the way down to $1.0 \text{ x} \beta^{e_{\min}}$.

Exceptions, Flags and Trap Handlers

When an exceptional condition like division by zero or overflow occurs in IEEE arithmetic, the default is to deliver a result and continue. Typical of the default results are NaN for 0/0 and $\sqrt{-1}$, and ∞ for 1/0 and overflow. The preceding sections gave examples where proceeding from an exception with these default values was the reasonable thing to do. When any exception occurs, a status flag is also set. Implementations of the IEEE standard are required to provide users with a way to read and write the status flags. The flags are "sticky" in that once set, they remain set until explicitly cleared. Testing the flags is the only way to distinguish 1/0, which is a genuine infinity from an overflow.

Sometimes continuing execution in the face of exception conditions is not appropriate. The section "Infinity" on page 184 gave the example of $x/(x^2 + 1)$. When $x > \sqrt{\beta}\beta^{e_{\max}/2}$, the denominator is infinite, resulting in a final answer of 0, which is totally wrong. Although for this formula the problem can be solved by rewriting it as $1/(x + x^{-1})$, rewriting may not always solve the problem. The IEEE standard strongly recommends that implementations allow trap handlers to be installed. Then when an exception occurs, the trap handler is called instead of setting the flag. The value returned by the trap handler will be used as the result of the operation. It is the responsibility of the trap handler to either clear or set the status flag; otherwise, the value of the flag is allowed to be undefined.

The IEEE standard divides exceptions into 5 classes: overflow, underflow, division by zero, invalid operation and inexact. There is a separate status flag for each class of exception. The meaning of the first three exceptions is self-evident. Invalid operation covers the situations listed in TABLE D-3, and any comparison that involves a NaN. The default result of an operation that causes an invalid exception is to return a NaN, but the converse is not true. When one of the operands to an operation is a NaN, the result is a NaN but no invalid exception is raised unless the operation also satisfies one of the conditions in TABLE D-3.[1]

TABLE D-4 Exceptions in IEEE 754*

Exception	**Result when traps disabled**	**Argument to trap handler**
overflow	$\pm\infty$ or $\pm x_{\max}$	round($x2^{-\alpha}$)
underflow	0, $\pm 2^{e_{\min}}$ or denormal	round($x2^{\alpha}$)
divide by zero	∞	operands
invalid	NaN	operands
inexact	round(x)	round(x)

*x is the exact result of the operation, $\alpha = 192$ for single precision, 1536 for double, and $x_{\max} = 1.11 \ldots 11 \times 2^{e_{\max}}$.

The inexact exception is raised when the result of a floating-point operation is not exact. In the $\beta = 10$, $p = 3$ system, $3.5 \otimes 4.2 = 14.7$ is exact, but $3.5 \otimes 4.3 = 15.0$ is not exact (since $3.5 \cdot 4.3 = 15.05$), and raises an inexact exception. “Binary to Decimal Conversion” on page 215 discusses an algorithm that uses the inexact exception. A summary of the behavior of all five exceptions is given in TABLE D-4.

There is an implementation issue connected with the fact that the inexact exception is raised so often. If floating-point hardware does not have flags of its own, but instead interrupts the operating system to signal a floating-point exception, the cost of inexact exceptions could be prohibitive. This cost can be avoided by having the status flags maintained by software. The first time an exception is raised, set the software flag for the appropriate class, and tell the floating-point hardware to mask off that class of exceptions. Then all further exceptions will run without interrupting the operating system. When a user resets that status flag, the hardware mask is re-enabled.

1. No invalid exception is raised unless a “trapping” NaN is involved in the operation. See section 6.2 of IEEE Std 754-1985. – Ed.

Trap Handlers

One obvious use for trap handlers is for backward compatibility. Old codes that expect to be aborted when exceptions occur can install a trap handler that aborts the process. This is especially useful for codes with a loop like `do S until (x >= 100)`. Since comparing a NaN to a number with <, ≤, >, ≥, or = (but not ≠) always returns false, this code will go into an infinite loop if `x` ever becomes a NaN.

There is a more interesting use for trap handlers that comes up when computing products such as $\Pi_{i=1}^{n} x_i$ that could potentially overflow. One solution is to use logarithms, and compute $\exp(\Sigma \log x_i)$ instead. The problem with this approach is that it is less accurate, and that it costs more than the simple expression Πx_i, even if there is no overflow. There is another solution using trap handlers called *over/underflow counting* that avoids both of these problems [Sterbenz 1974].

The idea is as follows. There is a global counter initialized to zero. Whenever the partial product $p_k = \Pi_{i=1}^{k} x_i$ overflows for some *k*, the trap handler increments the counter by one and returns the overflowed quantity with the exponent wrapped around. In IEEE 754 single precision, $e_{\max} = 127$, so if $p_k = 1.45 \times 2^{130}$, it will overflow and cause the trap handler to be called, which will wrap the exponent back into range, changing p_k to 1.45×2^{-62} (see below). Similarly, if p_k underflows, the counter would be decremented, and negative exponent would get wrapped around into a positive one. When all the multiplications are done, if the counter is zero then the final product is p_n. If the counter is positive, the product overflowed, if the counter is negative, it underflowed. If none of the partial products are out of range, the trap handler is never called and the computation incurs no extra cost. Even if there are over/underflows, the calculation is more accurate than if it had been computed with logarithms, because each p_k was computed from p_{k-1} using a full precision multiply. Barnett [1987] discusses a formula where the full accuracy of over/underflow counting turned up an error in earlier tables of that formula.

IEEE 754 specifies that when an overflow or underflow trap handler is called, it is passed the wrapped-around result as an argument. The definition of wrapped-around for overflow is that the result is computed as if to infinite precision, then divided by 2α, and then rounded to the relevant precision. For underflow, the result is multiplied by 2α. The exponent α is 192 for single precision and 1536 for double precision. This is why $1.45 \text{ x } 2^{130}$ was transformed into 1.45×2^{-62} in the example above.

Rounding Modes

In the IEEE standard, rounding occurs whenever an operation has a result that is not exact, since (with the exception of binary decimal conversion) each operation is computed exactly and then rounded. By default, rounding means round toward nearest. The standard requires that three other rounding modes be provided, namely round toward 0, round toward +∞, and round toward -∞. When used with the convert to integer operation, round toward -∞ causes the convert to become the floor function, while round toward +∞ is ceiling. The rounding mode affects overflow, because when round toward 0 or round toward -∞ is in effect, an overflow of positive magnitude causes the default result to be the largest representable number, not +∞. Similarly, overflows of negative magnitude will produce the largest negative number when round toward +∞ or round toward 0 is in effect.

One application of rounding modes occurs in interval arithmetic (another is mentioned in "Binary to Decimal Conversion" on page 215). When using interval arithmetic, the sum of two numbers x and y is an interval $[\underline{z}, \bar{z}]$, where $\underline{z}$ is $x \oplus y$ rounded toward -∞, and $\bar{Z}$ is $x \oplus y$ rounded toward +∞. The exact result of the addition is contained within the interval $[\underline{z}, \bar{z}]$. Without rounding modes, interval arithmetic is usually implemented by computing $\underline{z} = (x \oplus y)(1 - \varepsilon)$ and $\bar{z} = (x \oplus y)(1 + \varepsilon)$, where ε is machine epsilon.[1] This results in overestimates for the size of the intervals. Since the result of an operation in interval arithmetic is an interval, in general the input to an operation will also be an interval. If two intervals $[\underline{x}, \bar{x}]$, and $[\underline{y}, \bar{y}]$, are added, the result is $[\underline{z}, \bar{z}]$, where $\underline{z}$ is $\underline{x} \oplus \underline{y}$ with the rounding mode set to round toward -∞, and $\bar{z}$ is $\underline{z} \oplus \underline{y}$ with the rounding mode set to round toward +∞.

When a floating-point calculation is performed using interval arithmetic, the final answer is an interval that contains the exact result of the calculation. This is not very helpful if the interval turns out to be large (as it often does), since the correct answer could be anywhere in that interval. Interval arithmetic makes more sense when used in conjunction with a multiple precision floating-point package. The calculation is first performed with some precision p. If interval arithmetic suggests that the final answer may be inaccurate, the computation is redone with higher and higher precisions until the final interval is a reasonable size.

Flags

The IEEE standard has a number of flags and modes. As discussed above, there is one status flag for each of the five exceptions: underflow, overflow, division by zero, invalid operation and inexact. There are four rounding modes: round toward nearest, round toward +∞, round toward 0, and round toward -∞. It is strongly recommended that there be an enable mode bit for each of the five exceptions. This section gives some simple examples of how these modes and flags can be put to good use. A more sophisticated example is discussed in the section "Binary to Decimal Conversion" on page 215.

1. $\underline{Z}$ may be greater than $\bar{Z}$ if both x and y are negative. – Ed.

Consider writing a subroutine to compute x^n, where n is an integer. When $n > 0$, a simple routine like

```
PositivePower(x,n) {
 while (n is even) {
     x = x*x
     n = n/2
 }
 u = x
 while (true) {
     n = n/2
     if (n==0) return u
     x = x*x
     if (n is odd) u = u*x
 }
```

If n < 0, then a more accurate way to compute x^n is not to call `PositivePower(1/x, -n)` but rather `1/PositivePower(x, -n)`, because the first expression multiplies n quantities each of which have a rounding error from the division (i.e., $1/x$). In the second expression these are exact (i.e., x), and the final division commits just one additional rounding error. Unfortunately, these is a slight snag in this strategy. If `PositivePower(x, -n)` underflows, then either the underflow trap handler will be called, or else the underflow status flag will be set. This is incorrect, because if x^{-n} underflows, then x^n will either overflow or be in range.[1] But since the IEEE standard gives the user access to all the flags, the subroutine can easily correct for this. It simply turns off the overflow and underflow trap enable bits and saves the overflow and underflow status bits. It then computes `1/PositivePower(x, -n)`. If neither the overflow nor underflow status bit is set, it restores them together with the trap enable bits. If one of the status bits is set, it restores the flags and redoes the calculation using `PositivePower(1/x, -n)`, which causes the correct exceptions to occur.

Another example of the use of flags occurs when computing arccos via the formula

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}.$$

If arctan(∞) evaluates to $\pi/2$, then arccos(-1) will correctly evaluate to 2·arctan(∞) = π, because of infinity arithmetic. However, there is a small snag, because the computation of $(1 - x)/(1 + x)$ will cause the divide by zero exception flag to be set, even though arccos(-1) is not exceptional. The solution to this problem is straightforward. Simply save the value of the divide by zero flag before computing arccos, and then restore its old value after the computation.

1. It can be in range because if $x < 1$, $n < 0$ and x^{-n} is just a tiny bit smaller than the underflow threshold $2^{e_{\min}}$, then $x^n \approx 2^{-e_{\min}} < 2^{e_{\max}}$, and so may not overflow, since in all IEEE precisions, $-e_{\min} < e_{\max}$.

Systems Aspects

The design of almost every aspect of a computer system requires knowledge about floating-point. Computer architectures usually have floating-point instructions, compilers must generate those floating-point instructions, and the operating system must decide what to do when exception conditions are raised for those floating-point instructions. Computer system designers rarely get guidance from numerical analysis texts, which are typically aimed at users and writers of software, not at computer designers. As an example of how plausible design decisions can lead to unexpected behavior, consider the following BASIC program.

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
    else print "Not Equal"
```

When compiled and run using Borland's Turbo Basic on an IBM PC, the program prints `Not Equal`! This example will be analyzed in the next section

Incidentally, some people think that the solution to such anomalies is never to compare floating-point numbers for equality, but instead to consider them equal if they are within some error bound E. This is hardly a cure-all because it raises as many questions as it answers. What should the value of E be? If $x < 0$ and $y > 0$ are within E, should they really be considered to be equal, even though they have different signs? Furthermore, the relation defined by this rule, $a \sim b \Leftrightarrow |a - b| < E$, is not an equivalence relation because $a \sim b$ and $b \sim c$ does not imply that $a \sim c$.

Instruction Sets

It is quite common for an algorithm to require a short burst of higher precision in order to produce accurate results. One example occurs in the quadratic formula $(-b \pm \sqrt{b^2 - 4ac})/2a$. As discussed in the section "Proof of Theorem 4" on page 212, when $b^2 \approx 4ac$, rounding error can contaminate up to half the digits in the roots computed with the quadratic formula. By performing the subcalculation of $b^2 - 4ac$ in double precision, half the double precision bits of the root are lost, which means that all the single precision bits are preserved.

The computation of $b^2 - 4ac$ in double precision when each of the quantities a, b, and c are in single precision is easy if there is a multiplication instruction that takes two single precision numbers and produces a double precision result. In order to produce the exactly rounded product of two p-digit numbers, a multiplier needs to generate the entire $2p$ bits of product, although it may throw bits away as it proceeds. Thus, hardware to compute a double precision product from single precision operands will normally be only a little more expensive than a single precision multiplier, and much cheaper than a double precision multiplier. Despite this, modern instruction sets tend to provide only instructions that produce a result of the same precision as the operands.[1]

If an instruction that combines two single precision operands to produce a double precision product was only useful for the quadratic formula, it wouldn't be worth adding to an instruction set. However, this instruction has many other uses. Consider the problem of solving a system of linear equations,

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = \quad b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = \quad b_2$$

$$\cdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = \quad b_n$$

which can be written in matrix form as $Ax = b$, where

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2(1)n} \\ & & \ldots & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}$$

Suppose that a solution $x^{(1)}$ is computed by some method, perhaps Gaussian elimination. There is a simple way to improve the accuracy of the result called *iterative improvement*. First compute

$$\xi = Ax^{(1)} - b \tag{12}$$

and then solve the system

$$Ay = \xi \tag{13}$$

1. This is probably because designers like "orthogonal" instruction sets, where the precisions of a floating-point instruction are independent of the actual operation. Making a special case for multiplication destroys this orthogonality.

Note that if $x^{(1)}$ is an exact solution, then ξ is the zero vector, as is y. In general, the computation of ξ and y will incur rounding error, so $Ay \approx \xi \approx Ax^{(1)} - b = A(x^{(1)} - x)$, where x is the (unknown) true solution. Then $y \approx x^{(1)} - x$, so an improved estimate for the solution is

$$x^{(2)} = x^{(1)} - y \tag{14}$$

The three steps (12), (13), and (14) can be repeated, replacing $x^{(1)}$ with $x^{(2)}$, and $x^{(2)}$ with $x^{(3)}$. This argument that $x^{(i+1)}$ is more accurate than $x^{(i)}$ is only informal. For more information, see [Golub and Van Loan 1989].

When performing iterative improvement, ξ is a vector whose elements are the difference of nearby inexact floating-point numbers, and so can suffer from catastrophic cancellation. Thus iterative improvement is not very useful unless $\xi = Ax^{(1)} - b$ is computed in double precision. Once again, this is a case of computing the product of two single precision numbers (A and $x^{(1)}$), where the full double precision result is needed.

To summarize, instructions that multiply two floating-point numbers and return a product with twice the precision of the operands make a useful addition to a floating-point instruction set. Some of the implications of this for compilers are discussed in the next section.

Languages and Compilers

The interaction of compilers and floating-point is discussed in Farnum [1988], and much of the discussion in this section is taken from that paper.

Ambiguity

Ideally, a language definition should define the semantics of the language precisely enough to prove statements about programs. While this is usually true for the integer part of a language, language definitions often have a large grey area when it comes to floating-point. Perhaps this is due to the fact that many language designers believe that nothing can be proven about floating-point, since it entails rounding error. If so, the previous sections have demonstrated the fallacy in this reasoning. This section discusses some common grey areas in language definitions, including suggestions about how to deal with them.

Remarkably enough, some languages don't clearly specify that if `x` is a floating-point variable (with say a value of `3.0/10.0`), then every occurrence of (say) `10.0*x` must have the same value. For example Ada, which is based on Brown's model, seems to imply that floating-point arithmetic only has to satisfy Brown's axioms, and thus expressions can have one of many possible values. Thinking about floating-point in this fuzzy way stands in sharp contrast to the IEEE model, where the result of each floating-point operation is precisely defined. In the IEEE model, we can prove that `(3.0/10.0)*10.0` evaluates to `3` (Theorem 7). In Brown's model, we cannot.

Another ambiguity in most language definitions concerns what happens on overflow, underflow and other exceptions. The IEEE standard precisely specifies the behavior of exceptions, and so languages that use the standard as a model can avoid any ambiguity on this point.

Another grey area concerns the interpretation of parentheses. Due to roundoff errors, the associative laws of algebra do not necessarily hold for floating-point numbers. For example, the expression `(x+y)+z` has a totally different answer than `x+(y+z)` when $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ (it is 1 in the former case, 0 in the latter). The importance of preserving parentheses cannot be overemphasized. The algorithms presented in theorems 3, 4 and 6 all depend on it. For example, in Theorem 6, the formula $x_h = mx - (mx - x)$ would reduce to $x_h = x$ if it weren't for parentheses, thereby destroying the entire algorithm. A language definition that does not require parentheses to be honored is useless for floating-point calculations.

Subexpression evaluation is imprecisely defined in many languages. Suppose that `ds` is double precision, but `x` and `y` are single precision. Then in the expression `ds + x*y` is the product performed in single or double precision? Another example: in `x + m/n` where `m` and `n` are integers, is the division an integer operation or a floating-point one? There are two ways to deal with this problem, neither of which is completely satisfactory. The first is to require that all variables in an expression have the same type. This is the simplest solution, but has some drawbacks. First of all, languages like Pascal that have subrange types allow mixing subrange variables with integer variables, so it is somewhat bizarre to prohibit mixing single and double precision variables. Another problem concerns constants. In the expression `0.1*x`, most languages interpret 0.1 to be a single precision constant. Now suppose the programmer decides to change the declaration of all the floating-point variables from single to double precision. If 0.1 is still treated as a single precision constant, then there will be a compile time error. The programmer will have to hunt down and change every floating-point constant.

The second approach is to allow mixed expressions, in which case rules for subexpression evaluation must be provided. There are a number of guiding examples. The original definition of C required that every floating-point expression be computed in double precision [Kernighan and Ritchie 1978]. This leads to anomalies like the example at the beginning of this section. The expression `3.0/7.0` is computed in double precision, but if `q` is a single-precision variable, the quotient is rounded to single precision for storage. Since 3/7 is a repeating binary fraction, its computed value in double precision is different from its stored value in single precision. Thus the comparison $q = 3/7$ fails. This suggests that computing every expression in the highest precision available is not a good rule.

Another guiding example is inner products. If the inner product has thousands of terms, the rounding error in the sum can become substantial. One way to reduce this rounding error is to accumulate the sums in double precision (this will be discussed in more detail in the section "Optimizers" on page 201). If `d` is a double precision variable, and `x[]` and `y[]` are single precision arrays, then the inner product loop will look like `d = d + x[i]*y[i]`. If the multiplication is done in single precision, than much of the advantage of double precision accumulation is lost, because the product is truncated to single precision just before being added to a double precision variable.

A rule that covers both of the previous two examples is to compute an expression in the highest precision of any variable that occurs in that expression. Then `q = 3.0/7.0` will be computed entirely in single precision[1] and will have the boolean value true, whereas `d = d + x[i]*y[i]` will be computed in double precision, gaining the full advantage of double precision accumulation. However, this rule is too simplistic to cover all cases cleanly. If `dx` and `dy` are double precision variables, the expression `y = x + single(dx-dy)` contains a double precision variable, but performing the sum in double precision would be pointless, because both operands are single precision, as is the result.

A more sophisticated subexpression evaluation rule is as follows. First assign each operation a tentative precision, which is the maximum of the precisions of its operands. This assignment has to be carried out from the leaves to the root of the expression tree. Then perform a second pass from the root to the leaves. In this pass, assign to each operation the maximum of the tentative precision and the precision expected by the parent. In the case of `q = 3.0/7.0`, every leaf is single precision, so all the operations are done in single precision. In the case of `d = d + x[i]*y[i]`, the tentative precision of the multiply operation is single precision, but in the second pass it gets promoted to double precision, because its parent operation expects a double precision operand. And in `y = x + single(dx-dy)`, the addition is done in single precision. Farnum [1988] presents evidence that this algorithm in not difficult to implement.

The disadvantage of this rule is that the evaluation of a subexpression depends on the expression in which it is embedded. This can have some annoying consequences. For example, suppose you are debugging a program and want to know the value of a subexpression. You cannot simply type the subexpression to the debugger and ask it to be evaluated, because the value of the subexpression in the program depends on the expression it is embedded in. A final comment on subexpressions: since converting decimal constants to binary is an operation, the evaluation rule also affects the interpretation of decimal constants. This is especially important for constants like `0.1` which are not exactly representable in binary.

1. This assumes the common convention that `3.0` is a single-precision constant, while `3.0D0` is a double precision constant.

Another potential grey area occurs when a language includes exponentiation as one of its built-in operations. Unlike the basic arithmetic operations, the value of exponentiation is not always obvious [Kahan and Coonen 1982]. If `**` is the exponentiation operator, then `(-3)**3` certainly has the value -27. However, `(-3.0)**3.0` is problematical. If the `**` operator checks for integer powers, it would compute `(-3.0)**3.0` as $-3.0^3 = -27$. On the other hand, if the formula $x^y = e^{y \log x}$ is used to define `**` for real arguments, then depending on the log function, the result could be a NaN (using the natural definition of $\log(x)$ = `NaN` when $x < 0$). If the FORTRAN `CLOG` function is used however, then the answer will be -27, because the ANSI FORTRAN standard defines `CLOG(-3.0)` to be $i\pi + \log 3$ [ANSI 1978]. The programming language Ada avoids this problem by only defining exponentiation for integer powers, while ANSI FORTRAN prohibits raising a negative number to a real power.

In fact, the FORTRAN standard says that

> Any arithmetic operation whose result is not mathematically defined is prohibited...

Unfortunately, with the introduction of $\pm\infty$ by the IEEE standard, the meaning of *not mathematically defined* is no longer totally clear cut. One definition might be to use the method shown in section "Infinity" on page 184. For example, to determine the value of a^b, consider non-constant analytic functions f and g with the property that $f(x) \rightarrow a$ and $g(x) \rightarrow b$ as $x \rightarrow 0$. If $f(x)^{g(x)}$ always approaches the same limit, then this should be the value of a^b. This definition would set $2^\infty = \infty$ which seems quite reasonable. In the case of 1.0^∞, when $f(x) = 1$ and $g(x) = 1/x$ the limit approaches 1, but when $f(x) = 1 - x$ and $g(x) = 1/x$ the limit is e^{-1}. So 1.0^∞, should be a NaN. In the case of 0^0, $f(x)^{g(x)} = e^{g(x) \log f(x)}$. Since f and g are analytic and take on the value 0 at 0, $f(x) = a_1x^1 + a_2x^2 + \ldots$ and $g(x) = b_1x^1 + b_2x^2 + \ldots$. Thus $\lim_{x \rightarrow 0} g(x) \log f(x) = \lim_{x \rightarrow 0} x \log(x(a_1 + a_2x + \ldots)) = \lim_{x \rightarrow 0} x \log(a_1x) = 0$. So $f(x)^{g(x)} \rightarrow e^0 = 1$ for all f and g, which means that $0^0 = 1$.[1] [2] Using this definition would unambiguously define the exponential function for all arguments, and in particular would define `(-3.0)**3.0` to be -27.

The IEEE Standard

The section "The IEEE Standard" on page 176," discussed many of the features of the IEEE standard. However, the IEEE standard says nothing about how these features are to be accessed from a programming language. Thus, there is usually a mismatch between floating-point hardware that supports the standard and programming languages like C, Pascal or FORTRAN. Some of the IEEE capabilities

1. The conclusion that $0^0 = 1$ depends on the restriction that f be nonconstant. If this restriction is removed, then letting f be the identically 0 function gives 0 as a possible value for $\lim_{x \rightarrow 0} f(x)^{g(x)}$, and so 0^0 would have to be defined to be a NaN.

2. In the case of 0^0, plausibility arguments can be made, but the convincing argument is found in "Concrete Mathematics" by Graham, Knuth and Patashnik, and argues that $0^0 = 1$ for the binomial theorem to work. – Ed.

can be accessed through a library of subroutine calls. For example the IEEE standard requires that square root be exactly rounded, and the square root function is often implemented directly in hardware. This functionality is easily accessed via a library square root routine. However, other aspects of the standard are not so easily implemented as subroutines. For example, most computer languages specify at most two floating-point types, while the IEEE standard has four different precisions (although the recommended configurations are single plus single-extended or single, double, and double-extended). Infinity provides another example. Constants to represent $\pm\infty$ could be supplied by a subroutine. But that might make them unusable in places that require constant expressions, such as the initializer of a constant variable.

A more subtle situation is manipulating the state associated with a computation, where the state consists of the rounding modes, trap enable bits, trap handlers and exception flags. One approach is to provide subroutines for reading and writing the state. In addition, a single call that can atomically set a new value and return the old value is often useful. As the examples in the section "Flags" on page 192 show, a very common pattern of modifying IEEE state is to change it only within the scope of a block or subroutine. Thus the burden is on the programmer to find each exit from the block, and make sure the state is restored. Language support for setting the state precisely in the scope of a block would be very useful here. Modula-3 is one language that implements this idea for trap handlers [Nelson 1991].

There are a number of minor points that need to be considered when implementing the IEEE standard in a language. Since $x - x = +0$ for all x,[1] $(+0) - (+0) = +0$. However, $-(+0) = -0$, thus $-x$ should not be defined as $0 - x$. The introduction of NaNs can be confusing, because a NaN is never equal to any other number (including another NaN), so $x = x$ is no longer always true. In fact, the expression $x \neq x$ is the simplest way to test for a NaN if the IEEE recommended function `Isnan` is not provided. Furthermore, NaNs are unordered with respect to all other numbers, so $x \leq y$ cannot be defined as *not* $x > y$. Since the introduction of NaNs causes floating-point numbers to become partially ordered, a `compare` function that returns one of <, =, >, or *unordered* can make it easier for the programmer to deal with comparisons.

Although the IEEE standard defines the basic floating-point operations to return a NaN if any operand is a NaN, this might not always be the best definition for compound operations. For example when computing the appropriate scale factor to use in plotting a graph, the maximum of a set of values must be computed. In this case it makes sense for the max operation to simply ignore NaNs.

Finally, rounding can be a problem. The IEEE standard defines rounding very precisely, and it depends on the current value of the rounding modes. This sometimes conflicts with the definition of implicit rounding in type conversions or the explicit `round` function in languages. This means that programs which wish to

1. Unless the rounding mode is round toward $-\infty$, in which case $x - x = -0$.

use IEEE rounding can't use the natural language primitives, and conversely the language primitives will be inefficient to implement on the ever increasing number of IEEE machines.

Optimizers

Compiler texts tend to ignore the subject of floating-point. For example Aho et al. [1986] mentions replacing `x/2.0` with `x*0.5`, leading the reader to assume that `x/10.0` should be replaced by `0.1*x`. However, these two expressions do not have the same semantics on a binary machine, because 0.1 cannot be represented exactly in binary. This textbook also suggests replacing `x*y-x*z` by `x*(y-z)`, even though we have seen that these two expressions can have quite different values when $y \approx z$. Although it does qualify the statement that any algebraic identity can be used when optimizing code by noting that optimizers should not violate the language definition, it leaves the impression that floating-point semantics are not very important. Whether or not the language standard specifies that parenthesis must be honored, `(x+y)+z` can have a totally different answer than `x+(y+z)`, as discussed above. There is a problem closely related to preserving parentheses that is illustrated by the following code:

```
eps = 1;
do eps = 0.5*eps; while (eps + 1 > 1);
```

This is designed to give an estimate for machine epsilon. If an optimizing compiler notices that $eps + 1 > 1 \Leftrightarrow eps > 0$, the program will be changed completely. Instead of computing the smallest number x such that $1 \oplus x$ is still greater than x ($x \approx e \approx \beta^{-p}$), it will compute the largest number x for which $x/2$ is rounded to 0 ($x \approx \beta^{e_{\min}}$). Avoiding this kind of "optimization" is so important that it is worth presenting one more very useful algorithm that is totally ruined by it.

Many problems, such as numerical integration and the numerical solution of differential equations involve computing sums with many terms. Because each addition can potentially introduce an error as large as .5 ulp, a sum involving thousands of terms can have quite a bit of rounding error. A simple way to correct for this is to store the partial summand in a double precision variable and to perform each addition using double precision. If the calculation is being done in single precision, performing the sum in double precision is easy on most computer systems. However, if the calculation is already being done in double precision, doubling the precision is not so simple. One method that is sometimes advocated is to sort the numbers and add them from smallest to largest. However, there is a much more efficient method which dramatically improves the accuracy of sums, namely

Theorem 8 (Kahan Summation Formula)

Suppose that $\Sigma_{j=1}^{N} x_j$ *is computed using the following algorithm*

```
S = X[1];
C = 0;
for j = 2 to N {
    Y = X[j] - C;
    T = S + Y;
    C = (T - S) - Y;
    S = T;
}
```

Then the computed sum S is equal to $\Sigma x_j(1 + \delta_j) + O(N\varepsilon^2)\Sigma|x_j|$, *where* $(|\delta_j| \le 2\varepsilon)$.

Using the naive formula Σx_j, the computed sum is equal to $\Sigma x_j(1 + \delta_j)$ where $|\delta_j| < (n - j)e$. Comparing this with the error in the Kahan summation formula shows a dramatic improvement. Each summand is perturbed by only $2e$, instead of perturbations as large as ne in the simple formula. Details are in, "Errors In Summation" on page 216.

An optimizer that believed floating-point arithmetic obeyed the laws of algebra would conclude that $C = [T\text{-}S] - Y = [(S+Y)\text{-}S] - Y = 0$, rendering the algorithm completely useless. These examples can be summarized by saying that optimizers should be extremely cautious when applying algebraic identities that hold for the mathematical real numbers to expressions involving floating-point variables.

Another way that optimizers can change the semantics of floating-point code involves constants. In the expression `1.0E-40*x`, there is an implicit decimal to binary conversion operation that converts the decimal number to a binary constant. Because this constant cannot be represented exactly in binary, the inexact exception should be raised. In addition, the underflow flag should to be set if the expression is evaluated in single precision. Since the constant is inexact, its exact conversion to binary depends on the current value of the IEEE rounding modes. Thus an optimizer that converts `1.0E-40` to binary at compile time would be changing the semantics of the program. However, constants like 27.5 which are exactly representable in the smallest available precision can be safely converted at compile time, since they are always exact, cannot raise any exception, and are unaffected by the rounding modes. Constants that are intended to be converted at compile time should be done with a constant declaration, such as `const pi = 3.14159265`.

Common subexpression elimination is another example of an optimization that can change floating-point semantics, as illustrated by the following code

```
C = A*B;
RndMode = Up
D = A*B;
```

Although `A*B` can appear to be a common subexpression, it is not because the rounding mode is different at the two evaluation sites. Three final examples: $x = x$ cannot be replaced by the boolean constant `true`, because it fails when x is a NaN; $-x = 0 - x$ fails for $x = +0$; and $x < y$ is not the opposite of $x \geq y$, because NaNs are neither greater than nor less than ordinary floating-point numbers.

Despite these examples, there are useful optimizations that can be done on floating-point code. First of all, there are algebraic identities that are valid for floating-point numbers. Some examples in IEEE arithmetic are $x + y = y + x$, $2 \times x = x + x$, $1 \times x = x$, and $0.5 \times x = x/2$. However, even these simple identities can fail on a few machines such as CDC and Cray supercomputers. Instruction scheduling and in-line procedure substitution are two other potentially useful optimizations.[1]

As a final example, consider the expression `dx = x*y`, where `x` and `y` are single precision variables, and `dx` is double precision. On machines that have an instruction that multiplies two single precision numbers to produce a double precision number, `dx = x*y` can get mapped to that instruction, rather than compiled to a series of instructions that convert the operands to double and then perform a double to double precision multiply.

Some compiler writers view restrictions which prohibit converting $(x + y) + z$ to $x + (y + z)$ as irrelevant, of interest only to programmers who use unportable tricks. Perhaps they have in mind that floating-point numbers model real numbers and should obey the same laws that real numbers do. The problem with real number semantics is that they are extremely expensive to implement. Every time two n bit numbers are multiplied, the product will have $2n$ bits. Every time two n bit numbers with widely spaced exponents are added, the number of bits in the sum is n + the space between the exponents. The sum could have up to $(e^{max} - e^{min})$ + n bits, or roughly $2 \cdot e^{max}$ + n bits. An algorithm that involves thousands of operations (such as solving a linear system) will soon be operating on numbers with many significant bits, and be hopelessly slow. The implementation of library functions such as sin and cos is even more difficult, because the value of these transcendental functions aren't rational numbers. Exact integer arithmetic is often provided by lisp systems and is handy for some problems. However, exact floating-point arithmetic is rarely useful.

1. The VMS math libraries on the VAX use a weak form of in-line procedure substitution, in that they use the inexpensive jump to subroutine call rather than the slower `CALLS` and `CALLG` instructions.

The fact is that there are useful algorithms (like the Kahan summation formula) that exploit the fact that $(x + y) + z \neq x + (y + z)$, and work whenever the bound

$$a \oplus b = (a + b)(1 + \delta)$$

holds (as well as similar bounds for –, × and /). Since these bounds hold for almost all commercial hardware, it would be foolish for numerical programmers to ignore such algorithms, and it would be irresponsible for compiler writers to destroy these algorithms by pretending that floating-point variables have real number semantics.

Exception Handling

The topics discussed up to now have primarily concerned systems implications of accuracy and precision. Trap handlers also raise some interesting systems issues. The IEEE standard strongly recommends that users be able to specify a trap handler for each of the five classes of exceptions, and the section "Trap Handlers" on page 191, gave some applications of user defined trap handlers. In the case of invalid operation and division by zero exceptions, the handler should be provided with the operands, otherwise, with the exactly rounded result. Depending on the programming language being used, the trap handler might be able to access other variables in the program as well. For all exceptions, the trap handler must be able to identify what operation was being performed and the precision of its destination.

The IEEE standard assumes that operations are conceptually serial and that when an interrupt occurs, it is possible to identify the operation and its operands. On machines which have pipelining or multiple arithmetic units, when an exception occurs, it may not be enough to simply have the trap handler examine the program counter. Hardware support for identifying exactly which operation trapped may be necessary.

Another problem is illustrated by the following program fragment.

```
x = y*z;
z = x*w;
a = b + c;
d = a/x;
```

Suppose the second multiply raises an exception, and the trap handler wants to use the value of `a`. On hardware that can do an add and multiply in parallel, an optimizer would probably move the addition operation ahead of the second multiply, so that the add can proceed in parallel with the first multiply. Thus when the second multiply traps, `a = b + c` has already been executed, potentially changing the result of `a`. It would not be reasonable for a compiler to avoid this kind of optimization, because every floating-point operation can potentially trap, and thus virtually all instruction scheduling optimizations would be eliminated. This problem can be avoided by prohibiting trap handlers from accessing any variables of the program directly. Instead, the handler can be given the operands or result as an argument.

But there are still problems. In the fragment

```
x = y*z;
z = a + b;
```

the two instructions might well be executed in parallel. If the multiply traps, its argument `z` could already have been overwritten by the addition, especially since addition is usually faster than multiply. Computer systems that support the IEEE standard must provide some way to save the value of `z`, either in hardware or by having the compiler avoid such a situation in the first place.

W. Kahan has proposed using *presubstitution* instead of trap handlers to avoid these problems. In this method, the user specifies an exception and the value he wants to be used as the result when the exception occurs. As an example, suppose that in code for computing (sin $x)/x$, the user decides that $x = 0$ is so rare that it would improve performance to avoid a test for $x = 0$, and instead handle this case when a 0/0 trap occurs. Using IEEE trap handlers, the user would write a handler that returns a value of 1 and install it before computing sin x/x. Using presubstitution, the user would specify that when an invalid operation occurs, the value 1 should be used. Kahan calls this presubstitution, because the value to be used must be specified before the exception occurs. When using trap handlers, the value to be returned can be computed when the trap occurs.

The advantage of presubstitution is that it has a straightforward hardware implementation.[1] As soon as the type of exception has been determined, it can be used to index a table which contains the desired result of the operation. Although presubstitution has some attractive attributes, the widespread acceptance of the IEEE standard makes it unlikely to be widely implemented by hardware manufacturers.

The Details

A number of claims have been made in this paper concerning properties of floating-point arithmetic. We now proceed to show that floating-point is not black magic, but rather is a straightforward subject whose claims can be verified mathematically. This section is divided into three parts. The first part presents an introduction to error

1. The difficulty with presubstitution is that it requires either direct hardware implementation, or continuable floating-point traps if implemented in software. – Ed.

analysis, and provides the details for the section "Rounding Error" on page 162. The second part explores binary to decimal conversion, filling in some gaps from the section "The IEEE Standard" on page 176. The third part discusses the Kahan summation formula, which was used as an example in the section "Systems Aspects" on page 194.

Rounding Error

In the discussion of rounding error, it was stated that a single guard digit is enough to guarantee that addition and subtraction will always be accurate (Theorem 2). We now proceed to verify this fact. Theorem 2 has two parts, one for subtraction and one for addition. The part for subtraction is

Theorem 9

If x and y are positive floating-point numbers in a format with parameters β *and p, and if subtraction is done with p + 1 digits (i.e. one guard digit), then the relative rounding error in the result is less than*

$$\left(\frac{\beta}{2}+1\right)\beta^{-p} = \left(1+\frac{2}{\beta}\right)e \le 2e.$$

Proof

Interchange x and y if necessary so that $x > y$. It is also harmless to scale x and y so that x is represented by $x_0.x_1 \ldots x_{p-1} \times \beta^0$. If y is represented as $y_0.y_1 \ldots y_{p-1}$, then the difference is exact. If y is represented as $0.y_1 \ldots y_p$, then the guard digit ensures that the computed difference will be the exact difference rounded to a floating-point number, so the rounding error is at most e. In general, let $y = 0.0 \ldots 0y_{k+1} \ldots y_{k+p}$ and $\bar{y}$ be y truncated to $p + 1$ digits. Then

$$y - \bar{y} < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \ldots + \beta^{-p-k}). \tag{15}$$

From the definition of guard digit, the computed value of $x - y$ is $x - \bar{y}$ rounded to be a floating-point number, that is, $(x - \bar{y}) + \delta$, where the rounding error δ satisfies

$$|\delta| \le (\beta/2)\beta^{-p}. \tag{16}$$

The exact difference is $x - y$, so the error is $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y + \delta$. There are three cases. If $x - y \geq 1$ then the relative error is bounded by

$$\frac{y - \bar{y} + \delta}{1} \leq \beta^{-p}\left[(\beta - 1)(\beta^{-1} + \ldots + \beta^{-k}) + \beta/2\right] < \beta^{-p}(1 + \beta/2). \tag{17}$$

Secondly, if $x - \bar{y} < 1$, then $\delta = 0$. Since the smallest that $x - y$ can be is

$$1.0 - 0.\left(\overbrace{0\ldots0}^{k}\right)\left(\overbrace{\rho\ldots\rho}^{p}\right) > (\beta - 1)(\beta^{-1} + \ldots + \beta^{-k}), \text{ where } \rho = \beta - 1,$$

in this case the relative error is bounded by

$$\frac{y - \bar{y} + \delta}{(\beta - 1)(\beta^{-1} + \ldots + \beta^{-k})} < \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \ldots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \ldots + \beta^{-k})} = \beta^{-p}. \tag{18}$$

The final case is when $x - y < 1$ but $x - \bar{y} \geq 1$. The only way this could happen is if $x - \bar{y} = 1$, in which case $\delta = 0$. But if $\delta = 0$, then (18) applies, so that again the relative error is bounded by $\beta^{-p} < \beta^{-p}(1 + \beta/2)$. ▮

When $\beta = 2$, the bound is exactly $2e$, and this bound is achieved for $x = 1 + 2^{2-p}$ and $y = 2^{1-p} - 2^{1-2p}$ in the limit as $p \to \infty$. When adding numbers of the same sign, a guard digit is not necessary to achieve good accuracy, as the following result shows.

Theorem 10

If $x \geq 0$ and $y \geq 0$, then the relative error in computing $x + y$ is at most 2ε, even if no guard digits are used.

Proof

The algorithm for addition with k guard digits is similar to that for subtraction. If $x \geq y$, shift y right until the radix points of x and y are aligned. Discard any digits shifted past the $p + k$ position. Compute the sum of these two $p + k$ digit numbers exactly. Then round to p digits.

We will verify the theorem when no guard digits are used; the general case is similar. There is no loss of generality in assuming that $x \geq y \geq 0$ and that x is scaled to be of the form $d.dd\ldots d \times \beta^0$. First, assume there is no carry out. Then the digits shifted off the end of y have a value less than β^{-p+1}, and the sum is at least 1, so the relative error is less than $\beta^{-p+1}/1 = 2e$. If there is a carry out, then the error from shifting must be added to the rounding error of

$$\frac{1}{2}\beta^{-p+2}.$$

The sum is at least β, so the relative error is less than

$$\left(\beta^{-p+1} + \frac{1}{2}\beta^{-p+2}\right)/\beta = (1 + \beta/2)\beta^{-p} \leq 2\varepsilon. ∎$$

It is obvious that combining these two theorems gives Theorem 2. Theorem 2 gives the relative error for performing one operation. Comparing the rounding error of $x^2 - y^2$ and $(x + y)\ (x - y)$ requires knowing the relative error of multiple operations. The relative error of $x \ominus y$ is $\delta_1 = [(x \ominus y) - (x - y)]\ /\ (x - y)$, which satisfies $|\delta_1| \leq 2e$. Or to write it another way

$$x \ominus y = (x - y)\ (1 + \delta_1), \quad |\delta_1| \leq 2e \tag{19}$$

Similarly

$$x \oplus y = (x + y)\ (1 + \delta_2), \quad |\delta_2| \leq 2e \tag{20}$$

Assuming that multiplication is performed by computing the exact product and then rounding, the relative error is at most .5 ulp, so

$$u \otimes v = uv\ (1 + \delta_3), \qquad |\delta_3| \leq e \tag{21}$$

for any floating-point numbers u and v. Putting these three equations together (letting $u = x \ominus y$ and $v = x \oplus y$) gives

$$(x \ominus y) \otimes (x \oplus y) = (x - y)\ (1 + \delta_1)\ (x + y)\ (1 + \delta_2)\ (1 + \delta_3) \tag{22}$$

So the relative error incurred when computing $(x - y)\ (x + y)$ is

$$\frac{(x-y) \ominus (x+y) - (x^2 - y^2)}{(x^2 - y^2)} = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) - 1 \tag{23}$$

This relative error is equal to $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3 + \delta_1\delta_2\delta_3$, which is bounded by $5\varepsilon + 8\varepsilon^2$. In other words, the maximum relative error is about 5 rounding errors (since e is a small number, e^2 is almost negligible).

A similar analysis of $(x \otimes x) \ominus (y \otimes y)$ cannot result in a small value for the relative error, because when two nearby values of x and y are plugged into x^2 - y^2, the relative error will usually be quite large. Another way to see this is to try and duplicate the analysis that worked on $(x \ominus y) \otimes (x \oplus y)$, yielding

$$\begin{aligned}(x \otimes x) \ominus (y \otimes y) &= [x^2(1 + \delta_1) - y^2(1 + \delta_2)]\,(1 + \delta_3) \\ &= ((x^2 - y^2)\,(1 + \delta_1) + (\delta_1 - \delta_2)y^2)\,(1 + \delta_3)\end{aligned}$$

When x and y are nearby, the error term $(\delta_1 - \delta_2)y^2$ can be as large as the result $x^2 - y^2$. These computations formally justify our claim that $(x - y)\,(x + y)$ is more accurate than $x^2 - y^2$.

We next turn to an analysis of the formula for the area of a triangle. In order to estimate the maximum error that can occur when computing with (7), the following fact will be needed.

Theorem 11

If subtraction is performed with a guard digit, and $y/2 \le x \le 2y$, then $x - y$ is computed exactly.

Proof

Note that if x and y have the same exponent, then certainly $x \ominus y$ is exact. Otherwise, from the condition of the theorem, the exponents can differ by at most 1. Scale and interchange x and y if necessary so that $0 \le y \le x$, and x is represented as $x_0.x_1 \ldots x_{p-1}$ and y as $0.y_1 \ldots y_p$. Then the algorithm for computing $x \ominus y$ will compute $x - y$ exactly and round to a floating-point number. If the difference is of the form $0.d_1 \ldots d_p$, the difference will already be p digits long, and no rounding is necessary. Since $x \le 2y$, $x - y \le y$, and since y is of the form $0.d_1 \ldots d_p$, so is $x - y$. ∎

When $\beta > 2$, the hypothesis of Theorem 11 cannot be replaced by $y/\beta \le x \le \beta y$; the stronger condition $y/2 \le x \le 2y$ is still necessary. The analysis of the error in $(x - y)\,(x + y)$, immediately following the proof of Theorem 10, used the fact that the relative error in the basic operations of addition and subtraction is small (namely equations (19) and (20)). This is the most common kind of error analysis. However, analyzing formula (7) requires something more, namely Theorem 11, as the following proof will show.

Theorem 12

If subtraction uses a guard digit, and if a,b and c are the sides of a triangle ($a \ge b \ge c$), then the relative error in computing $(a + (b + c))(c - (a - b))(c + (a - b))(a +(b - c))$ is at most 16ε, *provided $e < .005$.*

Proof

Let's examine the factors one by one. From Theorem 10, $b \oplus c = (b + c)(1 + \delta_1)$, where δ_1 is the relative error, and $|\delta_1| \le 2\varepsilon$. Then the value of the first factor is

$$(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1))(1 + \delta_2),$$

and thus

$$\begin{aligned}(a + b + c)(1 - 2\varepsilon)^2 &\le [a + (b + c)(1 - 2\varepsilon)] \cdot (1-2\varepsilon)\\ &\le a \oplus (b \oplus c)\\ &\le [a + (b + c)(1 + 2\varepsilon)](1 + 2\varepsilon)\\ &\le (a + b + c)(1 + 2\varepsilon)^2\end{aligned}$$

This means that there is an η_1 so that

$$(a \oplus (b \oplus c)) = (a + b + c)(1 + \eta_1)^2, \ |\eta_1| \le 2\varepsilon. \tag{24}$$

The next term involves the potentially catastrophic subtraction of c and $a \ominus b$, because $a \ominus b$ may have rounding error. Because a, b and c are the sides of a triangle, $a \le b + c$, and combining this with the ordering $c \le b \le a$ gives $a \le b + c \le 2b \le 2a$. So $a - b$ satisfies the conditions of Theorem 11. This means that $a - b = a \ominus b$ is exact, hence $c \ominus (a - b)$ is a harmless subtraction which can be estimated from Theorem 9 to be

$$(c \ominus (a \ominus b)) = (c - (a - b))(1 + \eta_2), \ |\eta_2| \le 2\varepsilon \tag{25}$$

The third term is the sum of two exact positive quantities, so

$$(c \oplus (a \ominus b)) = (c + (a - b))(1 + \eta_3), \ |\eta_3| \le 2\varepsilon \tag{26}$$

Finally, the last term is

$$(a \oplus (b \ominus c)) = (a + (b - c))(1 + \eta_4)^2, \ |\eta_4| \le 2\varepsilon, \tag{27}$$

using both Theorem 9 and Theorem 10. If multiplication is assumed to be exactly rounded, so that $x \otimes y = xy(1 + \zeta)$ with $|\zeta| \le \varepsilon$, then combining (24), (25), (26) and (27) gives

$$\begin{aligned}&(a \oplus (b \oplus c))(c \ominus (a \ominus b))(c \oplus (a \ominus b))(a \oplus (b \ominus c))\\ &\quad \le (a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))E\end{aligned}$$

where

$$E = (1 + \eta_1)^2 (1 + \eta_2) (1 + \eta_3) (1 + \eta_4)^2 (1 + \zeta_1)(1 + \zeta_2) (1 + \zeta_3)$$

An upper bound for E is $(1 + 2\varepsilon)^6(1 + \varepsilon)^3$, which expands out to $1 + 15\varepsilon + O(\varepsilon^2)$. Some writers simply ignore the $O(e^2)$ term, but it is easy to account for it. Writing $(1 + 2\varepsilon)^6(1 + \varepsilon)^3 = 1 + 15\varepsilon + \varepsilon R(\varepsilon)$, $R(\varepsilon)$ is a polynomial in e with positive coefficients, so it is an increasing function of ε. Since $R(.005) = .505$, $R(\varepsilon) < 1$ for all $\varepsilon < .005$, and hence $E \le (1 + 2\varepsilon)^6(1 + \varepsilon)^3 < 1 + 16\varepsilon$. To get a lower bound on E, note that $1 - 15\varepsilon - \varepsilon R(\varepsilon) < E$, and so when $\varepsilon < .005$, $1 - 16\varepsilon < (1 - 2\varepsilon)^6(1 - \varepsilon)^3$. Combining these two bounds yields $1 - 16\varepsilon < E < 1 + 16\varepsilon$. Thus the relative error is at most 16ε. ∎

Theorem 12 certainly shows that there is no catastrophic cancellation in formula (7). So although it is not necessary to show formula (7) is numerically stable, it is satisfying to have a bound for the entire formula, which is what Theorem 3 of "Cancellation" on page 168 gives.

Proof of Theorem 3

Let

$$q = (a + (b + c)) (c - (a - b)) (c + (a - b)) (a + (b - c))$$

and

$$Q = (a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c)).$$

Then, Theorem 12 shows that $Q = q(1 + \delta)$, with $\delta \le 16\varepsilon$. It is easy to check that

$$1 - 0.52|\delta| \le \sqrt{1 - |\delta|} \le \sqrt{1 + |\delta|} \le 1 + 0.52|\delta| \qquad (28)$$

provided $\delta \le .04/(.52)^2 \approx .15$, and since $|\delta| \le 16\varepsilon \le 16(.005) = .08$, δ does satisfy the condition. Thus

$$\sqrt{Q} = \sqrt{q(1 + \delta)} = \sqrt{q}(1 + \delta_1),$$

with $|\delta_1| \le .52|\delta| \le 8.5\varepsilon$. If square roots are computed to within .5 ulp, then the error when computing $\sqrt{Q}$ is $(1 + \delta_1)(1 + \delta_2)$, with $|\delta_2| \le \varepsilon$. If $\beta = 2$, then there is no further error committed when dividing by 4. Otherwise, one more factor $1 + \delta_3$ with $|\delta_3| \le \varepsilon$ is necessary for the division, and using the method in the proof of Theorem 12, the final error bound of $(1 + \delta_1) (1 + \delta_2) (1 + \delta_3)$ is dominated by $1 + \delta_4$, with $|\delta_4| \le 11\varepsilon$. ∎

To make the heuristic explanation immediately following the statement of Theorem 4 precise, the next theorem describes just how closely $\mu(x)$ approximates a constant.

Theorem 13

If $\mu(x) = \ln(1 + x)/x$, *then for* $0 \le x \le \frac{3}{4}$, $\frac{1}{2} \le \mu(x) \le 1$ *and the derivative satisfies* $|\mu'(x)| \le \frac{1}{2}$.

Proof

Note that $\mu(x) = 1 - x/2 + x^2/3 - \ldots$ is an alternating series with decreasing terms, so for $x \le 1$, $\mu(x) \ge 1 - x/2 \ge 1/2$. It is even easier to see that because the series for μ is alternating, $\mu(x) \le 1$. The Taylor series of $\mu'(x)$ is also alternating, and if $x \le \frac{3}{4}$ has decreasing terms, so $-\frac{1}{2} \le \mu'(x) \le -\frac{1}{2} + 2x/3$, or $-\frac{1}{2} \le \mu'(x) \le 0$, thus $|\mu'(x)| \le \frac{1}{2}$. ∎

Proof of Theorem 4

Since the Taylor series for ln

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

is an alternating series, $0 < x - \ln(1 + x) < x^2/2$, the relative error incurred when approximating $\ln(1 + x)$ by x is bounded by $x/2$. If $1 \oplus x = 1$, then $|x| < \varepsilon$, so the relative error is bounded by $\varepsilon/2$.

When $1 \oplus x \ne 1$, define $\hat{x}$ via $1 \oplus x = 1 + \hat{x}$. Then since $0 \le x < 1$, $(1 \oplus x) \ominus 1 = \hat{x}$. If division and logarithms are computed to within $\frac{1}{2}$ ulp, then the computed value of the expression $\ln(1 + x)/((1 + x) - 1)$ is

$$\frac{\ln(1 \oplus x)}{(1 \oplus x) \ominus 1}(1 + \delta_1)(1 + \delta_2) = \frac{\ln(1 + \hat{x})}{\hat{x}}(1 + \delta_1)(1 + \delta_2) = \mu(\hat{x})(1 + \delta_1)(1 + \delta_2) \tag{29}$$

where $|\delta_1| \le \varepsilon$ and $|\delta_2| \le \varepsilon$. To estimate $\mu(\hat{x})$, use the mean value theorem, which says that

$$\mu(\hat{x}) - \mu(x) = (\hat{x} - x)\mu'(\xi) \tag{30}$$

for some ξ between x and $\hat{x}$. From the definition of $\hat{x}$, it follows that $|\hat{x} - x| \leq \varepsilon$, and combining this with Theorem 13 gives $|\mu(\hat{x}) - \mu(x)| \leq \varepsilon/2$, or $|\mu(\hat{x})/\mu(x) - 1| \leq \varepsilon/(2|\mu(x)|) \leq \varepsilon$ which means that $\mu(\hat{x}) = \mu(x)\ (1 + \delta_3)$, with $|\delta_3| \leq \varepsilon$. Finally, multiplying by x introduces a final δ_4, so the computed value of

$$x \cdot \ln(1 \oplus x)/((1 \oplus x) \ominus 1)$$

is

$$\frac{x\ \ln(1+x)}{(1+x)-1}(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4), \quad |\delta_i| \leq \varepsilon$$

It is easy to check that if $\varepsilon < 0.1$, then

$$(1 + \delta_1)\ (1 + \delta_2)\ (1 + \delta_3)\ (1 + \delta_4) = 1 + \delta,$$

with $|\delta| \leq 5\varepsilon$. ▮

An interesting example of error analysis using formulas (19), (20), and (21) occurs in the quadratic formula $(-b \pm \sqrt{b^2 - 4ac})/2a$. The section "Cancellation" on page 168, explained how rewriting the equation will eliminate the potential cancellation caused by the ± operation. But there is another potential cancellation that can occur when computing $d = b_2 - 4ac$. This one cannot be eliminated by a simple rearrangement of the formula. Roughly speaking, when $b^2 \approx 4ac$, rounding error can contaminate up to half the digits in the roots computed with the quadratic formula. Here is an informal proof (another approach to estimating the error in the quadratic formula appears in Kahan [1972]).

If $b^2 \approx 4ac$, rounding error can contaminate up to half the digits in the roots computed with the quadratic formula $(-b \pm \sqrt{b^2 - 4ac})/2a$.

Proof: Write $(b \otimes b) \ominus (4a \otimes c) = (b^2(1 + \delta_1) - 4ac(1 + \delta_2))\ (1 + \delta_3)$, where $|\delta_i| \leq \varepsilon$. [1] Using $d = b^2 - 4ac$, this can be rewritten as $(d(1 + \delta_1) - 4ac(\delta_2 - \delta_1))\ (1 + \delta_3)$. To get an estimate for the size of this error, ignore second order terms in δ_i, in which case the absolute error is $d(\delta_1 + \delta_3) - 4ac\delta_4$, where $|\delta_4| = |\delta_1 - \delta_2| \leq 2\varepsilon$. Since $d \ll 4ac$, the first term $d(\delta_1 + \delta_3)$ can be ignored. To estimate the second term, use the fact that $ax^2 + bx + c = a(x - r_1)\ (x - r_2)$, so $ar_1r_2 = c$. Since $b^2 \approx 4ac$, then $r_1 \approx r_2$, so the second error term is $4ac\delta_4 \approx 4a^2r_1\delta_4^2$. Thus the computed value of $\sqrt{d}$ is

$$\sqrt{d + 4a^2r_1^2\delta_4}\,.$$

The inequality

$$p - q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q, \quad p \geq q > 0$$

1. In this informal proof, assume that $\beta = 2$ so that multiplication by 4 is exact and doesn't require a δ_i.

shows that

$$\sqrt{d + 4a^2 r_1^2 \delta_4} = \sqrt{d} + E,$$

where

$$|E| \le \sqrt{4a^2 r_1^2 |\delta_4|},$$

so the absolute error in $\sqrt{d}/2a$ is about $r_1\sqrt{\delta_4}$. Since $\delta_4 \approx \beta^{-p}$, $\sqrt{\delta_4} \approx \beta^{-p/2}$, and thus the absolute error of $r_1\sqrt{\delta_4}$ destroys the bottom half of the bits of the roots $r_1 \approx r_2$. In other words, since the calculation of the roots involves computing with $(\sqrt{d})/(2a)$, and this expression does not have meaningful bits in the position corresponding to the lower order half of r_i, then the lower order bits of r_i cannot be meaningful. ▮

Finally, we turn to the proof of Theorem 6. It is based on the following fact, which is proven in the section "Theorem 14 and Theorem 8" on page 221.

Theorem 14

Let $0 < k < p$, and set $m = \beta^k + 1$, and assume that floating-point operations are exactly rounded. Then $(m \otimes x) \ominus (m \otimes x \ominus x)$ is exactly equal to x rounded to $p - k$ significant digits. More precisely, x is rounded by taking the significand of x, imagining a radix point just left of the k least significant digits and rounding to an integer.

Proof of Theorem 6

By Theorem 14, x_h is x rounded to $p - k = \lfloor p/2 \rfloor$ places. If there is no carry out, then certainly x_h can be represented with $\lfloor p/2 \rfloor$ significant digits. Suppose there is a carry-out. If $x = x_0.x_1 \ldots x_{p-1} \times \beta^e$, then rounding adds 1 to x_{p-k-1}, and the only way there can be a carry-out is if $x_{p-k-1} = \beta - 1$, but then the low order digit of x_h is $1 + x_{p-k-1} = 0$, and so again x_h is representable in $\lfloor p/2 \rfloor$ digits.

To deal with x_l, scale x to be an integer satisfying $\beta^{p-1} \le x \le \beta^p - 1$. Let $x = \bar{x}_h + \bar{x}_l$ where $\bar{x}_h$ is the $p - k$ high order digits of x, and $\bar{x}_l$ is the k low order digits. There are three cases to consider. If $\bar{x}_l < (\beta/2)\beta^{k-1}$, then rounding x to $p - k$ places is the same as chopping and $x_h = \bar{x}_h$, and $x_l = \bar{x}_l$. Since $\bar{x}_l$ has at most k digits, if p is even, then $\bar{x}_l$ has at most $k = \lceil p/2 \rceil = \lfloor p/2 \rfloor$ digits. Otherwise, $\beta = 2$ and $\bar{x}_1 < 2^{k-1}$ is representable with $k - 1 \le \lfloor p/2 \rfloor$ significant bits. The second case is when $\bar{x} > (\beta/2)\beta^{k-1}$, and then computing x_h involves rounding up, so $x_h = \bar{x}_h + \beta^k$, and $x_1 = x - x_h = x - \bar{x}_h - \beta^k = \bar{x}_l - \beta^k$. Once again, $\bar{x}_l$ has at most k digits, so is representable with $\lfloor p/2 \rfloor$ digits. Finally, if $\bar{x}_l = (\beta/2)\beta^{k-1}$, then $x_h = \bar{x}_h$ or $\bar{x}_h + \beta^k$ depending on whether there is a round up. So x_l is either $(\beta/2)\beta^{k-1}$ or $(\beta/2)\beta^{k-1} - \beta^k = -\beta^k/2$, both of which are represented with 1 digit. ▮

Theorem 6 gives a way to express the product of two working precision numbers exactly as a sum. There is a companion formula for expressing a sum exactly. If $|x| \geq |y|$ then $x + y = (x \oplus y) + (x \ominus (x \oplus y)) \oplus y$ [Dekker 1971; Knuth 1981, Theorem C in section 4.2.2]. However, when using exactly rounded operations, this formula is only true for $\beta = 2$, and not for $\beta = 10$ as the example $x = .99998$, $y = .99997$ shows.

Binary to Decimal Conversion

Since single precision has $p = 24$, and $2^{24} < 10^8$, you might expect that converting a binary number to 8 decimal digits would be sufficient to recover the original binary number. However, this is not the case.

Theorem 15

When a binary IEEE single precision number is converted to the closest eight digit decimal number, it is not always possible to uniquely recover the binary number from the decimal one. However, if nine decimal digits are used, then converting the decimal number to the closest binary number will recover the original floating-point number.

Proof

Binary single precision numbers lying in the half open interval $[10^3, 2^{10}) = [1000, 1024)$ have 10 bits to the left of the binary point, and 14 bits to the right of the binary point. Thus there are $(2^{10} - 10^3)2^{14} = 393{,}216$ different binary numbers in that interval. If decimal numbers are represented with 8 digits, then there are $(2^{10} - 10^3)10^4 = 240{,}000$ decimal numbers in the same interval. There is no way that 240,000 decimal numbers could represent 393,216 different binary numbers. So 8 decimal digits are not enough to uniquely represent each single precision binary number.

To show that 9 digits are sufficient, it is enough to show that the spacing between binary numbers is always greater than the spacing between decimal numbers. This will ensure that for each decimal number N, the interval

$$[N - \frac{1}{2}\text{ulp}, N + \frac{1}{2}\text{ulp}]$$

contains at most one binary number. Thus each binary number rounds to a unique decimal number which in turn rounds to a unique binary number.

To show that the spacing between binary numbers is always greater than the spacing between decimal numbers, consider an interval $[10^n, 10^{n+1}]$. On this interval, the spacing between consecutive decimal numbers is $10^{(n+1)-9}$. On $[10^n, 2^m]$, where m is the smallest integer so that $10^n < 2^m$, the spacing of binary numbers is 2^{m-24}, and the spacing gets larger further on in the interval. Thus it is enough to check that $10^{(n+1)-9} < 2^{m-24}$. But in fact, since $10^n < 2^m$, then $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^m 2^{-24}$. ∎

The same argument applied to double precision shows that 17 decimal digits are required to recover a double precision number.

Binary-decimal conversion also provides another example of the use of flags. Recall from the section "Precision" on page 178, that to recover a binary number from its decimal expansion, the decimal to binary conversion must be computed exactly. That conversion is performed by multiplying the quantities N and $10^{|P|}$ (which are both exact if $p < 13$) in single-extended precision and then rounding this to single precision (or dividing if $p < 0$; both cases are similar). Of course the computation of $N \cdot 10^{|P|}$ cannot be exact; it is the combined operation round$(N \cdot 10^{|P|})$ that must be exact, where the rounding is from single-extended to single precision. To see why it might fail to be exact, take the simple case of $\beta = 10$, $p = 2$ for single, and $p = 3$ for single-extended. If the product is to be 12.51, then this would be rounded to 12.5 as part of the single-extended multiply operation. Rounding to single precision would give 12. But that answer is not correct, because rounding the product to single precision should give 13. The error is due to double rounding.

By using the IEEE flags, double rounding can be avoided as follows. Save the current value of the inexact flag, and then reset it. Set the rounding mode to round-to-zero. Then perform the multiplication $N \cdot 10^{|P|}$. Store the new value of the inexact flag in `ixflag`, and restore the rounding mode and inexact flag. If `ixflag` is 0, then $N \cdot 10^{|P|}$ is exact, so round$(N \cdot 10^{|P|})$ will be correct down to the last bit. If `ixflag` is 1, then some digits were truncated, since round-to-zero always truncates. The significand of the product will look like $1.b_1 \ldots b_{22} b_{23} \ldots b_{31}$. A double rounding error may occur if $b_{23} \ldots b_{31} = 10 \ldots 0$. A simple way to account for both cases is to perform a logical `OR` of `ixflag` with b_{31}. Then round$(N \cdot 10^{|P|})$ will be computed correctly in all cases.

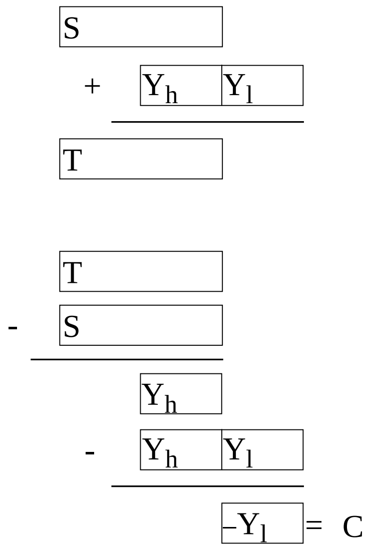
Errors In Summation

The section "Optimizers" on page 201, mentioned the problem of accurately computing very long sums. The simplest approach to improving accuracy is to double the precision. To get a rough estimate of how much doubling the precision improves the accuracy of a sum, let $s_1 = x_1$, $s_2 = s_1 \oplus x_2 \ldots$, $s_i = s_i - 1 \oplus x_i$. Then $s_i = (1 + \delta_i)(s_{i-1} + x_i)$, where $|\delta_i| \le \varepsilon$, and ignoring second order terms in δ_i gives

$$s_n = \sum_{j=1}^{n} x_j\left(1 + \sum_{k=j}^{n} \delta_k\right) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j\left(\sum_{k=j}^{n} \delta_k\right) \tag{31}$$

The first equality of (31) shows that the computed value of Σx_j is the same as if an exact summation was performed on perturbed values of x_j. The first term x_1 is perturbed by $n\varepsilon$, the last term x_n by only ε. The second equality in (31) shows that error term is bounded by $n\varepsilon\Sigma|x_j|$. Doubling the precision has the effect of squaring ε. If the sum is being done in an IEEE double precision format, $1/\varepsilon \approx 10^{16}$, so that $n\varepsilon \ll 1$ for any reasonable value of n. Thus, doubling the precision takes the maximum perturbation of $n\varepsilon$ and changes it to $n\varepsilon^2 \ll \varepsilon$. Thus the 2ε error bound for the Kahan summation formula (Theorem 8) is not as good as using double precision, even though it is much better than single precision.

For an intuitive explanation of why the Kahan summation formula works, consider the following diagram of the procedure.



Each time a summand is added, there is a correction factor C which will be applied on the next loop. So first subtract the correction C computed in the previous loop from X_j, giving the corrected summand Y. Then add this summand to the running sum S. The low order bits of Y (namely Y_l) are lost in the sum. Next compute the high order bits of Y by computing $T - S$. When Y is subtracted from this, the low order bits of Y will be recovered. These are the bits that were lost in the first sum in the diagram. They become the correction factor for the next loop. A formal proof of Theorem 8, taken from Knuth [1981] page 572, appears in the section "Theorem 14 and Theorem 8" on page 221."

Summary

It is not uncommon for computer system designers to neglect the parts of a system related to floating-point. This is probably due to the fact that floating-point is given very little (if any) attention in the computer science curriculum. This in turn has caused the apparently widespread belief that floating-point is not a quantifiable subject, and so there is little point in fussing over the details of hardware and software that deal with it.

This paper has demonstrated that it is possible to reason rigorously about floating-point. For example, floating-point algorithms involving cancellation can be proven to have small relative errors if the underlying hardware has a guard digit, and there is an efficient algorithm for binary-decimal conversion that can be proven to be invertible, provided that extended precision is supported. The task of constructing reliable floating-point software is made much easier when the underlying computer system is supportive of floating-point. In addition to the two examples just mentioned (guard digits and extended precision), the section "Systems Aspects" on page 194 of this paper has examples ranging from instruction set design to compiler optimization illustrating how to better support floating-point.

The increasing acceptance of the IEEE floating-point standard means that codes that utilize features of the standard are becoming ever more portable. The section "The IEEE Standard" on page 176, gave numerous examples illustrating how the features of the IEEE standard can be used in writing practical floating-point codes.

Acknowledgments

This article was inspired by a course given by W. Kahan at Sun Microsystems from May through July of 1988, which was very ably organized by David Hough of Sun. My hope is to enable others to learn about the interaction of floating-point and computer systems without having to get up in time to attend 8:00 a.m. lectures. Thanks are due to Kahan and many of my colleagues at Xerox PARC (especially John Gilbert) for reading drafts of this paper and providing many useful comments. Reviews from Paul Hilfinger and an anonymous referee also helped improve the presentation.

References

Aho, Alfred V., Sethi, R., and Ullman J. D. 1986. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

ANSI 1978. *American National Standard Programming Language FORTRAN*, ANSI Standard X3.9-1978, American National Standards Institute, New York, NY.

Barnett, David 1987. *A Portable Floating-Point Environment*, unpublished manuscript.

Brown, W. S. 1981. *A Simple but Realistic Model of Floating-Point Computation*, ACM Trans. on Math. Software 7(4), pp. 445-480.

Cody, W. J et. al. 1984. *A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic*, IEEE Micro 4(4), pp. 86-100.

Cody, W. J. 1988. *Floating-Point Standards — Theory and Practice*, in “Reliability in Computing: the role of interval methods in scientific computing”, ed. by Ramon E. Moore, pp. 99-107, Academic Press, Boston, MA.

Coonen, Jerome 1984. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, PhD Thesis, Univ. of California, Berkeley.

Dekker, T. J. 1971. *A Floating-Point Technique for Extending the Available Precision*, Numer. Math. 18(3), pp. 224-242.

Demmel, James 1984. *Underflow and the Reliability of Numerical Software*, SIAM J. Sci. Stat. Comput. 5(4), pp. 887-919.

Farnum, Charles 1988. *Compiler Support for Floating-point Computation*, Software-Practice and Experience, 18(7), pp. 701-709.

Forsythe, G. E. and Moler, C. B. 1967. *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Goldberg, I. Bennett 1967. *27 Bits Are Not Enough for 8-Digit Accuracy*, Comm. of the ACM. 10(2), pp 105-106.

Goldberg, David 1990. *Computer Arithmetic*, in “Computer Architecture: A Quantitative Approach”, by David Patterson and John L. Hennessy, Appendix A, Morgan Kaufmann, Los Altos, CA.

Golub, Gene H. and Van Loan, Charles F. 1989. *Matrix Computations*, 2nd edition, The Johns Hopkins University Press, Baltimore Maryland.

Graham, Ronald L., Knuth, Donald E. and Patashnik, Oren. 1989. *Concrete Mathematics,* Addison-Wesley, Reading, MA, p.162.

Hewlett Packard 1982. *HP-15C Advanced Functions Handbook*.

IEEE 1987. *IEEE Standard 754-1985 for Binary Floating-point Arithmetic*, IEEE, (1985). Reprinted in SIGPLAN 22(2) pp. 9-25.

Kahan, W. 1972. *A Survey Of Error Analysis*, in Information Processing 71, Vol 2, pp. 1214 - 1239 (Ljubljana, Yugoslavia), North Holland, Amsterdam.

Kahan, W. 1986. *Calculating Area and Angle of a Needle-like Triangle*, unpublished manuscript.

Kahan, W. 1987. *Branch Cuts for Complex Elementary Functions*, in "The State of the Art in Numerical Analysis", ed. by M.J.D. Powell and A. Iserles (Univ of Birmingham, England), Chapter 7, Oxford University Press, New York.

Kahan, W. 1988. Unpublished lectures given at Sun Microsystems, Mountain View, CA.

Kahan, W. and Coonen, Jerome T. 1982. *The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments*, in "The Relationship Between Numerical Computation And Programming Languages", ed. by J. K. Reid, pp. 103-115, North-Holland, Amsterdam.

Kahan, W. and LeBlanc, E. 1985. *Anomalies in the IBM Acrith Package*, Proc. 7th IEEE Symposium on Computer Arithmetic (Urbana, Illinois), pp. 322-331.

Kernighan, Brian W. and Ritchie, Dennis M. 1978. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.

Kirchner, R. and Kulisch, U. 1987. *Arithmetic for Vector Processors*, Proc. 8th IEEE Symposium on Computer Arithmetic (Como, Italy), pp. 256-269.

Knuth, Donald E., 1981. *The Art of Computer Programming, Volume II*, Second Edition, Addison-Wesley, Reading, MA.

Kulisch, U. W., and Miranker, W. L. 1986. *The Arithmetic of the Digital Computer: A New Approach*, SIAM Review 28(1), pp 1-36.

Matula, D. W. and Kornerup, P. 1985. *Finite Precision Rational Arithmetic: Slash Number Systems*, IEEE Trans. on Comput. C-34(1), pp 3-18.

Nelson, G. 1991. *Systems Programming With Modula-3*, Prentice-Hall, Englewood Cliffs, NJ.

Reiser, John F. and Knuth, Donald E. 1975. *Evading the Drift in Floating-point Addition*, Information Processing Letters 3(3), pp 84-87.

Sterbenz, Pat H. 1974. *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ.

Swartzlander, Earl E. and Alexopoulos, Aristides G. 1975. *The Sign/Logarithm Number System*, IEEE Trans. Comput. C-24(12), pp. 1238-1242.

Walther, J. S., 1971. *A unified algorithm for elementary functions*, Proceedings of the AFIP Spring Joint Computer Conf. 38, pp. 379-385.

Theorem 14 and Theorem 8

This section contains two of the more technical proofs that were omitted from the text.

Theorem 14

Let $0 < k < p$, and set $m = \beta^k + 1$, and assume that floating-point operations are exactly rounded. Then $(m \otimes x) \ominus (m \otimes x \ominus x)$ is exactly equal to x rounded to $p - k$ significant digits. More precisely, x is rounded by taking the significand of x, imagining a radix point just left of the k least significant digits, and rounding to an integer.

Proof

The proof breaks up into two cases, depending on whether or not the computation of $mx = \beta^k x + x$ has a carry-out or not.

Assume there is no carry out. It is harmless to scale x so that it is an integer. Then the computation of $mx = x + \beta^k x$ looks like this:

```
aa...aabb...bb
+aa...aabb...bb
zz...zzbb...bb
```

where x has been partitioned into two parts. The low order k digits are marked `b` and the high order $p - k$ digits are marked `a`. To compute $m \otimes x$ from mx involves rounding off the low order k digits (the ones marked with `b`) so

$$m \otimes x = mx - x \bmod(\beta^k) + r\beta^k \tag{32}$$

The value of r is 1 if `.bb...b` is greater than $\frac{1}{2}$ and 0 otherwise. More precisely

$$r = 1 \text{ if } \texttt{a.bb...b} \text{ rounds to } a + 1,\ r = 0 \text{ otherwise.} \tag{33}$$

Next compute $m \otimes x - x = mx - x \bmod(\beta^k) + r\beta^k - x = \beta^k(x + r) - x \bmod(\beta^k)$. The picture below shows the computation of $m \otimes x - x$ rounded, that is, $(m \otimes x) \ominus x$. The top line is $\beta^k(x + r)$, where `B` is the digit that results from adding `r` to the lowest order digit `b`.

```
aa...aabb...bB00...00
-bb...bb
zz...      zzZ00...00
```

If `.bb...b` $< \frac{1}{2}$ then $r = 0$, subtracting causes a borrow from the digit marked `B`, but the difference is rounded up, and so the net effect is that the rounded difference equals the top line, which is $\beta^k x$. If `.bb...b` $> \frac{1}{2}$ then $r = 1$, and 1 is subtracted from `B` because of the borrow, so the result is $\beta^k x$. Finally consider the case `.bb...b` $= \frac{1}{2}$. If $r = 0$ then `B` is even, `Z` is odd, and the difference is rounded up, giving $\beta^k x$. Similarly when $r = 1$, `B` is odd, `Z` is even, the difference is rounded down, so again the difference is $\beta^k x$. To summarize

$$(m \otimes x) \ominus x = \beta^k x \tag{34}$$

Combining equations (32) and (34) gives $(m \otimes x) - (m \otimes x \ominus x) = x - x \bmod(\beta^k) + \rho \cdot \beta^k$. The result of performing this computation is

```
r00...00
          + aa...aabb...bb
          -          bb...bb
             aa...aA00...00
```

The rule for computing r, equation (33), is the same as the rule for rounding `a...ab...b` to $p - k$ places. Thus computing $mx - (mx - x)$ in floating-point arithmetic precision is exactly equal to rounding x to $p - k$ places, in the case when $x + \beta^k x$ does not carry out.

When $x + \beta^k x$ does carry out, then $mx = \beta^k x + x$ looks like this:

```
aa...aabb...bb
+aa...aabb...bb
zz...zZbb...bb
```

Thus, $m \otimes x = mx - x \bmod(\beta^k) + w\beta^k$, where $w = -Z$ if $Z < \beta/2$, but the exact value of w is unimportant. Next, $m \otimes x - x = \beta^k x - x \bmod(\beta^k) + w\beta^k$. In a picture

```
aa...aabb...bb00...00
-   bb... bb
+   w
 zz    ... zZbb ...bb
```
[1]

1. This is the sum if adding w does not generate carry out. Additional argument is needed for the special case where adding w does generate carry out. – Ed.

Rounding gives $(m \otimes x) \ominus x = \beta^k x + w\beta^k - r\beta^k$, where $r = 1$ if `.bb...b` $> \frac{1}{2}$ or if `.bb...b` $= \frac{1}{2}$ and $b_0 = 1$.[1] Finally,

$$(m \otimes x) - (m \otimes x \ominus x) = mx - x \bmod(\beta^k) + w\beta^k - (\beta^k x + w\beta^k - r\beta^k)$$
$$= x - x \bmod(\beta^k) + r\beta^k.$$

And once again, $r = 1$ exactly when rounding `a...ab...b` to $p - k$ places involves rounding up. Thus Theorem 14 is proven in all cases. ▮

Theorem 8 (Kahan Summation Formula)

Suppose that $\Sigma_{j=1}^{N} x_j$ is computed using the following algorithm

```
S = X [1];
C = 0;
for j = 2 to N {
Y = X [j] - C;
   T = S + Y;
   C = (T - S) - Y;
   S = T;
}
```

Then the computed sum S is equal to $S = \Sigma x_j (1 + \delta_j) + O(N\varepsilon^2) \Sigma |x_j|$, *where* $|\delta_j| \le 2\varepsilon$.

Proof

First recall how the error estimate for the simple formula Σx_i went. Introduce $s_1 = x_1$, $s_i = (1 + \delta_i)(s_{i-1} + x_i)$. Then the computed sum is s_n, which is a sum of terms, each of which is an x_i multiplied by an expression involving δ_j's. The exact coefficient of x_1 is $(1 + \delta_2)(1 + \delta_3) \dots (1 + \delta_n)$, and so by renumbering, the coefficient of x_2 must be $(1 + \delta_3)(1 + \delta_4) \dots (1 + \delta n)$, and so on. The proof of Theorem 8 runs along exactly the same lines, only the coefficient of x_1 is more complicated. In detail $s_0 = c_0 = 0$ and

$$y_k = x_k \ominus c_{k-1} = (x_k - c_{k-1})(1 + \eta_k)$$

$$s_k = s_{k-1} \oplus\approx y_k = (s_{k-1} + y_k)(1 + \sigma_k)$$

$$c_k = (s_k \ominus s_{k-1}) \ominus y_k = [(s_k - s_{k-1})(1 + \gamma_k) - y_k](1 + \delta_k)$$

where all the Greek letters are bounded by ε. Although the coefficient of x_1 in s_k is the ultimate expression of interest, in turns out to be easier to compute the coefficient of x_1 in $s_k - c_k$ and c_k.

1. Rounding gives $\beta^k x + w\beta^k - r\beta^k$ only if $(\beta^k x + w\beta^k)$ keeps the form of $\beta^k x$. – Ed.

When $k = 1$,

$c_1 = (s_1(1 + \gamma_1) - y_1)\ (1 + d_1)$

$= y_1((1 + s_1)\ (1 + \gamma_1) - 1)\ (1 + d_1)$

$= x_1(s_1 + \gamma_1 + s_1 g_1)\ (1 + d_1)\ (1 + h_1)$

$s_1 - c_1 = x_1[(1 + s_1) - (s_1 + g_1 + s_1 g_1)\ (1 + d_1)](1 + h_1)$

$= x_1[1 - g_1 - s_1 d_1 - s_1 g_1 - d_1 g_1 - s_1 g_1 d_1](1 + h_1)$

Calling the coefficients of x_1 in these expressions C_k and S_k respectively, then

$C_1 = 2\varepsilon + O(\varepsilon^2)$

$S_1 = + \eta_1 - \gamma_1 + 4\varepsilon^2 + O(\varepsilon^3)$

To get the general formula for S_k and C_k, expand the definitions of s_k and c_k, ignoring all terms involving x_i with $i > 1$ to get

$s_k = (s_{k-1} + y_k)(1 + \sigma_k)$

$= [s_{k-1} + (x_k - c_{k-1})\ (1 + \eta_k)](1 + \sigma_k)$

$= [(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}](1+\sigma_k)$

$c_k = [\{s_k - s_{k-1}\}(1 + \gamma_k) - y_k](1 + \delta_k)$

$= [\{((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1 + \sigma_k) - s_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k)$

$= [\{(s_{k-1} - c_{k-1})\sigma_k - \eta_k c_{k-1}(1 + \sigma_k) - c_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k)$

$= [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))](1 + \delta_k)$,

$s_k - c_k = ((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})\ (1 + \sigma_k)$

$- [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k)](1 + \delta_k)$

$= (s_{k-1} - c_{k-1})((1 + \sigma_k) - \sigma_k(1 + \gamma_k)(1 + \delta_k))$

$+ c_{k-1}(-\eta_k(1 + \sigma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\ (1 + \delta_k))$

$= (s_{-1} - c_{k-1})\ (1 - \sigma_k(\gamma_k + \delta_k + \gamma_k\delta_k))$

$+ c_{k-1} - [\eta_k + \gamma_k + \eta_k(\gamma_k + \sigma_k\gamma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\delta_k]$

Since S_k and C_k are only being computed up to order ε^2, these formulas can be simplified to

$C_k = (\sigma_k + O(\varepsilon^2))S_{k-1} + (-\gamma_k + O(\varepsilon^2))C_{k-1}$

$S_k = ((1 + 2\varepsilon^2 + O(\varepsilon^3))S_{k-1} + (2\varepsilon + O(\varepsilon^2))C_{k-1}$

Using these formulas gives

$$C_2 = \sigma_2 + O(\varepsilon^2)$$

$$S_2 = 1 + \eta_1 - \gamma_1 + 10\varepsilon^2 + O(\varepsilon^3)$$

and in general it is easy to check by induction that

$$C_k = \sigma_k + O(\varepsilon^2)$$

$$S_k = 1 + \eta_1 - \gamma_1 + (4_k{+}2)\varepsilon^2 + O(\varepsilon^3)$$

Finally, what is wanted is the coefficient of x_1 in s_k. To get this value, let $x_{n+1} = 0$, let all the Greek letters with subscripts of $n + 1$ equal 0, and compute s_{n+1}. Then $s_{n+1} = s_n - c_n$, and the coefficient of x_1 in s_n is less than the coefficient in s_{n+1}, which is $S_n = 1 + \eta_1 - \gamma_1 + (4n + 2)\varepsilon^2 = (1 + 2\varepsilon + O(n\varepsilon^2))$. ∎

Differences Among IEEE 754 Implementations

Note – This section is not part of the published paper. It has been added to clarify certain points and correct possible misconceptions about the IEEE standard that the reader might infer from the paper. This material was not written by David Goldberg, but it appears here with his permission.

The preceding paper has shown that floating-point arithmetic must be implemented carefully, since programmers may depend on its properties for the correctness and accuracy of their programs. In particular, the IEEE standard requires a careful implementation, and it is possible to write useful programs that work correctly and deliver accurate results only on systems that conform to the standard. The reader might be tempted to conclude that such programs should be portable to all IEEE systems. Indeed, portable software would be easier to write if the remark "When a program is moved between two machines and both support IEEE arithmetic, then if any intermediate result differs, it must be because of software bugs, not from differences in arithmetic," were true.

Unfortunately, the IEEE standard does not guarantee that the same program will deliver identical results on all conforming systems. Most programs will actually produce different results on different systems for a variety of reasons. For one, most programs involve the conversion of numbers between decimal and binary formats, and the IEEE standard does not completely specify the accuracy with which such conversions must be performed. For another, many programs use elementary functions supplied by a system library, and the standard doesn't specify these functions at all. Of course, most programmers know that these features lie beyond the scope of the IEEE standard.

Many programmers may not realize that even a program that uses only the numeric formats and operations prescribed by the IEEE standard can compute different results on different systems. In fact, the authors of the standard intended to allow different implementations to obtain different results. Their intent is evident in the definition of the term *destination* in the IEEE 754 standard: "A destination may be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's format and the operands' values." (IEEE 754-1985, p. 7) In other words, the IEEE standard requires that each result be rounded correctly to the precision of the destination into which it will be placed, but the standard does not require that the precision of that destination be determined by a user's program. Thus, different systems may deliver their results to destinations with different precisions, causing the same program to produce different results (sometimes dramatically so), even though those systems all conform to the standard.

Several of the examples in the preceding paper depend on some knowledge of the way floating-point arithmetic is rounded. In order to rely on examples such as these, a programmer must be able to predict how a program will be interpreted, and in particular, on an IEEE system, what the precision of the destination of each arithmetic operation may be. Alas, the loophole in the IEEE standard's definition of *destination* undermines the programmer's ability to know how a program will be interpreted. Consequently, several of the examples given above, when implemented as apparently portable programs in a high-level language, may not work correctly on IEEE systems that normally deliver results to destinations with a different precision than the programmer expects. Other examples may work, but proving that they work may lie beyond the average programmer's ability.

In this section, we classify existing implementations of IEEE 754 arithmetic based on the precisions of the destination formats they normally use. We then review some examples from the paper to show that delivering results in a wider precision than a program expects can cause it to compute wrong results even though it is provably correct when the expected precision is used. We also revisit one of the proofs in the paper to illustrate the intellectual effort required to cope with unexpected precision even when it doesn't invalidate our programs. These examples show that despite all that the IEEE standard prescribes, the differences it allows among different implementations can prevent us from writing portable, efficient numerical software whose behavior we can accurately predict. To develop such software, then, we must first create programming languages and environments that limit the variability the IEEE standard permits and allow programmers to express the floating-point semantics upon which their programs depend.

Current IEEE 754 Implementations

Current implementations of IEEE 754 arithmetic can be divided into two groups distinguished by the degree to which they support different floating-point formats in hardware. *Extended-based* systems, exemplified by the Intel x86 family of processors, provide full support for an extended double precision format but only partial support for single and double precision: they provide instructions to load or store data in single and double precision, converting it on-the-fly to or from the extended double format, and they provide special modes (not the default) in which the results of arithmetic operations are rounded to single or double precision even though they are kept in registers in extended double format. (Motorola 68000 series processors round results to both the precision and range of the single or double formats in these modes. Intel x86 and compatible processors round results to the precision of the single or double formats but retain the same range as the extended double format.) *Single/double* systems, including most RISC processors, provide full support for single and double precision formats but no support for an IEEE-compliant extended double precision format. (The IBM POWER architecture provides only partial support for single precision, but for the purpose of this section, we classify it as a single/double system.)

To see how a computation might behave differently on an extended-based system than on a single/double system, consider a C version of the example from "Systems Aspects" on page 194:

```
int main() {
    double  q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

Here the constants 3.0 and 7.0 are interpreted as double precision floating-point numbers, and the expression 3.0/7.0 inherits the `double` data type. On a single/double system, the expression will be evaluated in double precision since that is the most efficient format to use. Thus, `q` will be assigned the value 3.0/7.0 rounded correctly to double precision. In the next line, the expression 3.0/7.0 will again be evaluated in double precision, and of course the result will be equal to the value just assigned to `q`, so the program will print "Equal" as expected.

On an extended-based system, even though the expression 3.0/7.0 has type `double`, the quotient will be computed in a register in extended double format, and thus in the default mode, it will be rounded to extended double precision. When the resulting value is assigned to the variable `q`, however, it may then be stored in memory, and since `q` is declared `double`, the value will be rounded to double precision. In the next line, the expression 3.0/7.0 may again be evaluated in extended precision yielding a result that differs from the double precision value stored in `q`, causing the program to print "Not equal". Of course, other outcomes are possible, too: the compiler could decide to store and thus round the value of the expression 3.0/7.0 in the second line before comparing it with `q`, or it could keep `q` in a register in extended precision without storing it. An optimizing compiler might evaluate the expression 3.0/7.0 at compile time, perhaps in double precision or perhaps in extended double precision. (With one x86 compiler, the program prints "Equal" when compiled with optimization and "Not Equal" when compiled for debugging.) Finally, some compilers for extended-based systems automatically change the rounding precision mode to cause operations producing results in registers to round those results to single or double precision, albeit possibly with a wider range. Thus, on these systems, we can't predict the behavior of the program simply by reading its source code and applying a basic understanding of IEEE 754 arithmetic. Neither can we accuse the hardware or the compiler of failing to provide an IEEE 754 compliant environment; the hardware has delivered a correctly rounded result to each destination, as it is required to do, and the compiler has assigned some intermediate results to destinations that are beyond the user's control, as it is allowed to do.

Pitfalls in Computations on Extended-Based Systems

Conventional wisdom maintains that extended-based systems must produce results that are at least as accurate, if not more accurate than those delivered on single/double systems, since the former always provide at least as much precision and often more than the latter. Trivial examples such as the C program above as well as more subtle programs based on the examples discussed below show that this wisdom is naive at best: some apparently portable programs, which are indeed portable across single/double systems, deliver incorrect results on extended-based systems precisely because the compiler and hardware conspire to occasionally provide more precision than the program expects.

Current programming languages make it difficult for a program to specify the precision it expects. As the section "Languages and Compilers" on "Languages and Compilers" on page 196 mentions, many programming languages don't specify that each occurrence of an expression like `10.0*x` in the same context should evaluate to the same value. Some languages, such as Ada, were influenced in this respect by variations among different arithmetics prior to the IEEE standard. More recently, languages like ANSI C have been influenced by standard-conforming extended-based systems. In fact, the ANSI C standard explicitly allows a compiler to evaluate a floating-point expression to a precision wider than that normally associated with its type. As a result, the value of the expression `10.0*x` may vary in ways that depend on a variety of factors: whether the expression is immediately assigned to a variable or appears as a subexpression in a larger expression; whether the expression participates in a comparison; whether the expression is passed as an argument to a function, and if so, whether the argument is passed by value or by reference; the current precision mode; the level of optimization at which the program was compiled; the precision mode and expression evaluation method used by the compiler when the program was compiled; and so on.

Language standards are not entirely to blame for the vagaries of expression evaluation. Extended-based systems run most efficiently when expressions are evaluated in extended precision registers whenever possible, yet values that must be stored are stored in the narrowest precision required. Constraining a language to require that `10.0*x` evaluate to the same value everywhere would impose a performance penalty on those systems. Unfortunately, allowing those systems to evaluate `10.0*x` differently in syntactically equivalent contexts imposes a penalty of its own on programmers of accurate numerical software by preventing them from relying on the syntax of their programs to express their intended semantics.

Do real programs depend on the assumption that a given expression always evaluates to the same value? Recall the algorithm presented in Theorem 4 for computing ln(1 + x), written here in Fortran:

```
real function log1p(x)
real x
if (1.0 + x .eq. 1.0) then
   log1p = x
else
   log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

On an extended-based system, a compiler may evaluate the expression `1.0 + x` in the third line in extended precision and compare the result with `1.0`. When the same expression is passed to the log function in the sixth line, however, the compiler may store its value in memory, rounding it to single precision. Thus, if `x` is not so small that `1.0 + x` rounds to `1.0` in extended precision but small enough that `1.0 + x` rounds to `1.0` in single precision, then the value returned by `log1p(x)` will be zero instead of `x`, and the relative error will be one—rather larger than 5ε. Similarly, suppose the rest of the expression in the sixth line, including the reoccurrence of the subexpression `1.0 + x`, is evaluated in extended precision. In that case, if `x` is small but not quite small enough that `1.0 + x` rounds to `1.0` in single precision, then the value returned by `log1p(x)` can exceed the correct value by nearly as much as `x`, and again the relative error can approach one. For a concrete example, take x to be $2^{-24} + 2^{-47}$, so x is the smallest single precision number such that 1.0 + x rounds up to the next larger number, $1 + 2^{-23}$. Then log(1.0 + x) is approximately 2^{-23}. Because the denominator in the expression in the sixth line is evaluated in extended precision, it is computed exactly and delivers x, so log1p(x) returns approximately 2^{-23}, which is nearly twice as large as the exact value. (This actually happens with at least one compiler. When the preceding code is compiled by the Sun WorkShop Compilers 4.2.1 Fortran 77 compiler for x86 systems using the -O optimization flag, the generated code computes 1.0 + x exactly as described. As a result, the function delivers zero for log1p(1.0e-10) and 1.19209E-07 for log1p(5.97e-8).)

For the algorithm of Theorem 4 to work correctly, the expression 1.0 + x must be evaluated the same way each time it appears; the algorithm can fail on extended-based systems only when 1.0 + x is evaluated to extended double precision in one instance and to single or double precision in another. Of course, since log is a generic intrinsic function in Fortran, a compiler could evaluate the expression 1.0 + x in extended precision throughout, computing its logarithm in the same precision, but evidently we cannot assume that the compiler will do so. (One can also imagine a similar example involving a user-defined function. In that case, a compiler could still keep the argument in extended precision even though the function returns a single precision result, but few if any existing Fortran compilers do this, either.) We might therefore attempt to ensure that 1.0 + x is evaluated consistently by assigning it to a variable. Unfortunately, if we declare that variable real, we may still be foiled by a compiler that substitutes a value kept in a register in extended precision for one appearance of the variable and a value stored in memory in single precision for another. Instead, we would need to declare the variable with a type that corresponds to the extended precision format. Standard FORTRAN 77 does not provide a way to do this, and while Fortran 95 offers the SELECTED_REAL_KIND mechanism for describing various formats, it does not explicitly require implementations that evaluate expressions in extended precision to allow variables to be declared with that precision. In short, there is no portable way to write this program in standard Fortran that is guaranteed to prevent the expression 1.0 + x from being evaluated in a way that invalidates our proof.

There are other examples that can malfunction on extended-based systems even when each subexpression is stored and thus rounded to the same precision. The cause is *double-rounding*. In the default precision mode, an extended-based system will initially round each result to extended double precision. If that result is then stored to double precision, it is rounded again. The combination of these two roundings can yield a value that is different than what would have been obtained by rounding the first result correctly to double precision. This can happen when the result as rounded to extended double precision is a "halfway case", i.e., it lies exactly halfway between two double precision numbers, so the second rounding is determined by the round-ties-to-even rule. If this second rounding rounds in the same direction as the first, the net rounding error will exceed half a unit in the last place. (Note, though, that double-rounding only affects double precision computations. One can prove that the sum, difference, product, or quotient of two *p*-bit numbers, or the square root of a *p*-bit number, rounded first to *q* bits and then to *p* bits gives the same value as if the result were rounded just once to *p* bits provided $q \geq 2p + 2$. Thus, extended double precision is wide enough that single precision computations don't suffer double-rounding.)

Some algorithms that depend on correct rounding can fail with double-rounding. In fact, even some algorithms that don't require correct rounding and work correctly on a variety of machines that don't conform to IEEE 754 can fail with double-rounding. The most useful of these are the portable algorithms for performing simulated multiple precision arithmetic mentioned in the section "Theorem 5" on page 173. For example, the procedure described in Theorem 6 for splitting a floating-point number into high and low parts doesn't work correctly in double-rounding arithmetic: try to split the double precision number $2^{52} + 3 \times 2^{26} - 1$ into two parts each with at most 26 bits. When each operation is rounded correctly to double precision, the high order part is $2^{52} + 2^{27}$ and the low order part is $2^{26} - 1$, but when each operation is rounded first to extended double precision and then to double precision, the procedure produces a high order part of $2^{52} + 2^{28}$ and a low order part of $-2^{26} - 1$. The latter number occupies 27 bits, so its square can't be computed exactly in double precision. Of course, it would still be possible to compute the square of this number in extended double precision, but the resulting algorithm would no longer be portable to single/double systems. Also, later steps in the multiple precision multiplication algorithm assume that all partial products have been computed in double precision. Handling a mixture of double and extended double variables correctly would make the implementation significantly more expensive.

Likewise, portable algorithms for adding multiple precision numbers represented as arrays of double precision numbers can fail in double-rounding arithmetic. These algorithms typically rely on a technique similar to Kahan's summation formula. As the informal explanation of the summation formula given the section "Errors In Summation" on page 216 suggests, if `s` and `y` are floating-point variables with `|s|` ≥ `|y|` and we compute:

```
t = s + y;
e = (s - t) + y;
```

then in most arithmetics, `e` recovers exactly the roundoff error that occurred in computing `t`. This technique doesn't work in double-rounded arithmetic, however: if `s` $= 2^{52} + 1$ and `y` $= 1/2 - 2^{-54}$, then `s + y` rounds first to $2^{52} + 3/2$ in extended double precision, and this value rounds to $2^{52} + 2$ in double precision by the round-ties-to-even rule; thus the net rounding error in computing `t` is $1/2 + 2^{-54}$, which is not representable exactly in double precision and so can't be computed exactly by the expression shown above. Here again, it would be possible to recover the roundoff error by computing the sum in extended double precision, but then a program would have to do extra work to reduce the final outputs back to double precision, and double-rounding could afflict this process, too. For this reason, although portable programs for simulating multiple precision arithmetic by these methods work correctly and efficiently on a wide variety of machines, they do not work as advertised on extended-based systems.

Finally, some algorithms that at first sight appear to depend on correct rounding may in fact work correctly with double-rounding. In these cases, the cost of coping with double-rounding lies not in the implementation but in the verification that the algorithm works as advertised. To illustrate, we prove the following variant of Theorem 7:

Theorem 7′

If m and n are integers representable in IEEE 754 double precision with $|m| < 2^{52}$ *and n has the special form* $n = 2^i + 2^j$, *then* $(m \oslash n) \otimes n = m$, *provided both floating-point operations are either rounded correctly to double precision or rounded first to extended double precision and then to double precision.*

Proof

Assume without loss that $m > 0$. Let $q = m \oslash n$. Scaling by powers of two, we can consider an equivalent setting in which $2^{52} \le m < 2^{53}$ and likewise for q, so that both m and q are integers whose least significant bits occupy the units place (i.e., ulp(m) = ulp(q) = 1). Before scaling, we assumed $m < 2^{52}$, so after scaling, m is an even integer. Also, because the scaled values of m and q satisfy $m/2 < q < 2m$, the corresponding value of n must have one of two forms depending on which of m or q is larger: if $q < m$, then evidently $1 < n < 2$, and since n is a sum of two powers of two, $n = 1 + 2^{-k}$ for some k; similarly, if $q > m$, then $1/2 < n < 1$, so $n = 1/2 + 2^{-(k+1)}$. (As n is the sum of two powers of two, the closest possible value of n to one is $n = 1 + 2^{-52}$. Because $m/(1 + 2^{-52})$ is no larger than the next smaller double precision number less than m, we can’t have $q = m$.)

Let e denote the rounding error in computing q, so that $q = m/n + e$, and the computed value $q \otimes n$ will be the (once or twice) rounded value of $m + ne$. Consider first the case in which each floating-point operation is rounded correctly to double precision. In this case, $|e| < 1/2$. If n has the form $1/2 + 2^{-(k+1)}$, then $ne = nq - m$ is an integer multiple of $2^{-(k+1)}$ and $|ne| < 1/4 + 2^{-(k+2)}$. This implies that $|ne| \le 1/4$. Recall that the difference between m and the next larger representable number is 1 and the difference between m and the next smaller representable number is either 1 if $m > 2^{52}$ or 1/2 if $m = 2^{52}$. Thus, as $|ne| \le 1/4$, $m + ne$ will round to m. (Even if $m = 2^{52}$ and $ne = -1/4$, the product will round to m by the round-ties-to-even rule.) Similarly, if n has the form $1 + 2^{-k}$, then ne is an integer multiple of 2^{-k} and $|ne| < 1/2 + 2^{-(k+1)}$; this implies $|ne| \le 1/2$. We can’t have $m = 2^{52}$ in this case because m is strictly greater than q, so m differs from its nearest representable neighbors by ±1. Thus, as $|ne| \leq 1/2$, again $m + ne$ will round to m. (Even if $|ne| = 1/2$, the product will round to m by the round-ties-to-even rule because m is even.) This completes the proof for correctly rounded arithmetic.

In double-rounding arithmetic, it may still happen that q is the correctly rounded quotient (even though it was actually rounded twice), so $|e| < 1/2$ as above. In this case, we can appeal to the arguments of the previous paragraph provided we consider the fact that $q \otimes n$ will be rounded twice. To account for this, note that the IEEE standard requires that an extended double format carry at least 64 significant bits, so that the numbers $m \pm 1/2$ and $m \pm 1/4$ are exactly representable in extended double precision. Thus, if n has the form $1/2 + 2^{-(k+1)}$, so that $|ne| \leq 1/4$, then rounding $m + ne$ to extended double precision must produce a result that differs from m by at most 1/4, and as noted above, this value will round to m in double precision. Similarly, if n has the form $1 + 2^{-k}$, so that $|ne| \leq 1/2$, then rounding $m + ne$ to extended double precision must produce a result that differs from m by at most 1/2, and this value will round to m in double precision. (Recall that $m > 2^{52}$ in this case.)

Finally, we are left to consider cases in which q is not the correctly rounded quotient due to double-rounding. In these cases, we have $|e| < 1/2 + 2^{-(d+1)}$ in the worst case, where d is the number of extra bits in the extended double format. (All existing extended-based systems support an extended double format with exactly 64 significant bits; for this format, $d = 64 - 53 = 11$.) Because double-rounding only produces an incorrectly rounded result when the second rounding is determined by the round-ties-to-even rule, q must be an even integer. Thus if n has the form $1/2 + 2^{-(k+1)}$, then $ne = nq - m$ is an integer multiple of 2^{-k}, and

$$|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}.$$

If $k \leq d$, this implies $|ne| \leq 1/4$. If $k > d$, we have $|ne| \leq 1/4 + 2^{-(d+2)}$. In either case, the first rounding of the product will deliver a result that differs from m by at most 1/4, and by previous arguments, the second rounding will round to m. Similarly, if n has the form $1 + 2^{-k}$, then ne is an integer multiple of $2^{-(k-1)}$, and

$$|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}.$$

If $k \leq d$, this implies $|ne| \leq 1/2$. If $k > d$, we have $|ne| \leq 1/2 + 2^{-(d+1)}$. In either case, the first rounding of the product will deliver a result that differs from m by at most 1/2, and again by previous arguments, the second rounding will round to m. ∎

The preceding proof shows that the product can incur double-rounding only if the quotient does, and even then, it rounds to the correct result. The proof also shows that extending our reasoning to include the possibility of double-rounding can be challenging even for a program with only two floating-point operations. For a more complicated program, it may be impossible to systematically account for the effects of double-rounding, not to mention more general combinations of double and extended double precision computations.

Programming Language Support for Extended Precision

The preceding examples should not be taken to suggest that extended precision *per se* is harmful. Many programs can benefit from extended precision when the programmer is able to use it selectively. Unfortunately, current programming languages do not provide sufficient means for a programmer to specify when and how extended precision should be used. To indicate what support is needed, we consider the ways in which we might want to manage the use of extended precision.

In a portable program that uses double precision as its nominal working precision, there are five ways we might want to control the use of a wider precision:

1. Compile to produce the fastest code, using extended precision where possible on extended-based systems. Clearly most numerical software does not require more of the arithmetic than that the relative error in each operation is bounded by the "machine epsilon". When data in memory are stored in double precision, the machine epsilon is usually taken to be the largest relative roundoff error in that precision, since the input data are (rightly or wrongly) assumed to have been rounded when they were entered and the results will likewise be rounded when they are stored. Thus, while computing some of the intermediate results in extended precision may yield a more accurate result, extended precision is not essential. In this case, we might prefer that the compiler use extended precision only when it will not appreciably slow the program and use double precision otherwise.

2. Use a format wider than double if it is reasonably fast and wide enough, otherwise resort to something else. Some computations can be performed more easily when extended precision is available, but they can also be carried out in double precision with only somewhat greater effort. Consider computing the Euclidean norm of a vector of double precision numbers. By computing the squares of the elements and accumulating their sum in an IEEE 754 extended double format with its wider exponent range, we can trivially avoid premature underflow or overflow for vectors of practical lengths. On extended-based systems, this is the fastest way to compute the norm. On single/double systems, an extended double format would have to be emulated in software (if one were supported at all), and such emulation would be much slower than simply using double precision, testing the exception flags to determine whether underflow or overflow occurred, and if so, repeating the computation with explicit scaling. Note that to support this use of extended precision, a language must provide both an indication of the widest available format that is reasonably fast, so that a

program can choose which method to use, and environmental parameters that indicate the precision and range of each format, so that the program can verify that the widest fast format is wide enough (e.g., that it has wider range than double).

3. Use a format wider than double even if it has to be emulated in software. For more complicated programs than the Euclidean norm example, the programmer may simply wish to avoid the need to write two versions of the program and instead rely on extended precision even if it is slow. Again, the language must provide environmental parameters so that the program can determine the range and precision of the widest available format.

4. Don't use a wider precision; round results correctly to the precision of the double format, albeit possibly with extended range. For programs that are most easily written to depend on correctly rounded double precision arithmetic, including some of the examples mentioned above, a language must provide a way for the programmer to indicate that extended precision must not be used, even though intermediate results may be computed in registers with a wider exponent range than double. (Intermediate results computed in this way can still incur double-rounding if they underflow when stored to memory: if the result of an arithmetic operation is rounded first to 53 significant bits, then rounded again to fewer significant bits when it must be denormalized, the final result may differ from what would have been obtained by rounding just once to a denormalized number. Of course, this form of double-rounding is highly unlikely to affect any practical program adversely.)

5. Round results correctly to both the precision and range of the double format. This strict enforcement of double precision would be most useful for programs that test either numerical software or the arithmetic itself near the limits of both the range and precision of the double format. Such careful test programs tend to be difficult to write in a portable way; they become even more difficult (and error prone) when they must employ dummy subroutines and other tricks to force results to be rounded to a particular format. Thus, a programmer using an extended-based system to develop robust software that must be portable to all IEEE 754 implementations would quickly come to appreciate being able to emulate the arithmetic of single/double systems without extraordinary effort.

No current language supports all five of these options. In fact, few languages have attempted to give the programmer the ability to control the use of extended precision at all. One notable exception is the ISO/IEC 9899:1999 Programming Languages - C standard, the latest revision to the C language, which is now in the final stages of standardization.

The C99 standard allows an implementation to evaluate expressions in a format wider than that normally associated with their type, but the C99 standard recommends using one of only three expression evaluation methods. The three recommended methods are characterized by the extent to which expressions are "promoted" to wider formats, and the implementation is encouraged to identify which method it uses by defining the preprocessor macro FLT_EVAL_METHOD: if FLT_EVAL_METHOD is 0, each expression is evaluated in a format that corresponds to its type; if FLT_EVAL_METHOD is 1, float expressions are promoted to the format that corresponds to double; and if FLT_EVAL_METHOD is 2, float and double expressions are promoted to the format that corresponds to long double. (An implementation is allowed to set FLT_EVAL_METHOD to –1 to indicate that the expression evaluation method is indeterminable.) The C99 standard also requires that the <math.h> header file define the types float_t and double_t, which are at least as wide as float and double, respectively, and are intended to match the types used to evaluate float and double expressions. For example, if FLT_EVAL_METHOD is 2, both float_t and double_t are long double. Finally, the C99 standard requires that the <float.h> header file define preprocessor macros that specify the range and precision of the formats corresponding to each floating-point type.

The combination of features required or recommended by the C99 standard supports some of the five options listed above but not all. For example, if an implementation maps the long double type to an extended double format and defines FLT_EVAL_METHOD to be 2, the programmer can reasonably assume that extended precision is relatively fast, so programs like the Euclidean norm example can simply use intermediate variables of type long double (or double_t). On the other hand, the same implementation must keep anonymous expressions in extended precision even when they are stored in memory (e.g., when the compiler must spill floating-point registers), and it must store the results of expressions assigned to variables declared double to convert them to double precision even if they could have been kept in registers. Thus, neither the double nor the double_t type can be compiled to produce the fastest code on current extended-based hardware.

Likewise, the C99 standard provides solutions to some of the problems illustrated by the examples in this section but not all. A C99 standard version of the log1p function is guaranteed to work correctly if the expression 1.0 + x is assigned to a variable (of any type) and that variable used throughout. A portable, efficient C99 standard program for splitting a double precision number into high and low parts, however, is more difficult: how can we split at the correct position and avoid double-rounding if we cannot guarantee that double expressions are rounded correctly to double precision? One solution is to use the double_t type to perform the splitting in double precision on single/double systems and in extended precision on extended-based systems, so that in either case the arithmetic will be correctly rounded. Theorem 14 says that we can split at any bit position provided we know the precision of the underlying arithmetic, and the FLT_EVAL_METHOD and environmental parameter macros should give us this information.

The following fragment shows one possible implementation:

```
#include <math.h>
#include <float.h>

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif

...
    double   x, xh, xl;
    double_t m;

    m = scalbn(1.0, PWR2) + 1.0;  // 2**PWR2 + 1
    xh = (m * x) - ((m * x) - x);
    xl = x - xh;
```

Of course, to find this solution, the programmer must know that `double` expressions may be evaluated in extended precision, that the ensuing double-rounding problem can cause the algorithm to malfunction, and that extended precision may be used instead according to Theorem 14. A more obvious solution is simply to specify that each expression be rounded correctly to double precision. On extended-based systems, this merely requires changing the rounding precision mode, but unfortunately, the C99 standard does not provide a portable way to do this. (Early drafts of the Floating-Point C Edits, the working document that specified the changes to be made to the C90 standard to support floating-point, recommended that implementations on systems with rounding precision modes provide `fegetprec` and `fesetprec` functions to get and set the rounding precision, analogous to the `fegetround` and `fesetround` functions that get and set the rounding direction. This recommendation was removed before the changes were made to the C99 standard.)

Coincidentally, the C99 standard's approach to supporting portability among systems with different integer arithmetic capabilities suggests a better way to support different floating-point architectures. Each C99 standard implementation supplies an `<stdint.h>` header file that defines those integer types the implementation supports, named according to their sizes and efficiency: for example, `int32_t` is an integer type exactly 32 bits wide, `int_fast16_t` is the implementation's fastest integer type at least 16 bits wide, and `intmax_t` is the widest integer type supported. One can imagine a similar scheme for floating-point types: for example, `float53_t` could name a floating-point type with exactly 53 bit precision but possibly wider range, `float_fast24_t` could name the implementation's fastest type with at least 24 bit precision, and `floatmax_t` could name the widest reasonably fast type supported. The fast types could allow compilers on extended-based systems to generate the fastest possible code subject only to the constraint that the values of named variables must not appear to change as a result of register spilling. The exact width types would cause compilers on extended-based systems to set the rounding precision mode to round to the specified precision, allowing wider range subject to the same constraint. Finally, `double_t` could name a type with both the precision and range of the IEEE 754 double format, providing strict double evaluation. Together with environmental parameter macros named accordingly, such a scheme would readily support all five options described above and allow programmers to indicate easily and unambiguously the floating-point semantics their programs require.

Must language support for extended precision be so complicated? On single/double systems, four of the five options listed above coincide, and there is no need to differentiate fast and exact width types. Extended-based systems, however, pose difficult choices: they support neither pure double precision nor pure extended precision computation as efficiently as a mixture of the two, and different programs call for different mixtures. Moreover, the choice of when to use extended precision should not be left to compiler writers, who are often tempted by benchmarks (and sometimes told outright by numerical analysts) to regard floating-point arithmetic as "inherently inexact" and therefore neither deserving nor capable of the predictability of integer arithmetic. Instead, the choice must be presented to programmers, and they will require languages capable of expressing their selection.

Conclusion

The foregoing remarks are not intended to disparage extended-based systems but to expose several fallacies, the first being that all IEEE 754 systems must deliver identical results for the same program. We have focused on differences between extended-based systems and single/double systems, but there are further differences among systems within each of these families. For example, some single/double systems provide a single instruction to multiply two numbers and add a third with just one final rounding. This operation, called a *fused multiply-add*, can cause the same program to produce different results across different single/double systems, and, like extended precision, it can even cause the same program to produce different results on the same system depending on whether and when it is used. (A fused multiply-add can also foil the splitting process of Theorem 6, although it can be used in a non-portable way to perform multiple precision multiplication without the need for splitting.) Even though the IEEE standard didn't anticipate such an operation, it nevertheless conforms: the intermediate product is delivered to a "destination" beyond the user's control that is wide enough to hold it exactly, and the final sum is rounded correctly to fit its single or double precision destination.

The idea that IEEE 754 prescribes precisely the result a given program must deliver is nonetheless appealing. Many programmers like to believe that they can understand the behavior of a program and prove that it will work correctly without reference to the compiler that compiles it or the computer that runs it. In many ways, supporting this belief is a worthwhile goal for the designers of computer systems and programming languages. Unfortunately, when it comes to floating-point arithmetic, the goal is virtually impossible to achieve. The authors of the IEEE standards knew that, and they didn't attempt to achieve it. As a result, despite nearly universal conformance to (most of) the IEEE 754 standard throughout the computer industry, programmers of portable software must continue to cope with unpredictable floating-point arithmetic.

If programmers are to exploit the features of IEEE 754, they will need programming languages that make floating-point arithmetic predictable. The C99 standard improves predictability to some degree at the expense of requiring programmers to write multiple versions of their programs, one for each `FLT_EVAL_METHOD`. Whether future languages will choose instead to allow programmers to write a single program with syntax that unambiguously expresses the extent to which it depends on IEEE 754 semantics remains to be seen. Existing extended-based systems threaten that prospect by tempting us to assume that the compiler and the hardware can know better than the programmer how a computation should be performed on a given system. That assumption is the second fallacy: the accuracy required in a computed result depends not on the machine that produces it but only on the conclusions that will be drawn from it, and of the programmer, the compiler, and the hardware, at best only the programmer can know what those conclusions may be.

APPENDIX **E**

Standards Compliance

The compilers, header files, and libraries in the Forte Developer compilers products for the Solaris environment support multiple standards: System V Interface Definition (SVID), Edition 3, X/Open and ANSI C. Accordingly, the mathematical library `libm` and related files have been modified so that C programs comply with the standards. Users' programs are usually not affected, because the differences primarily involve exception handling.

SVID History

To understand the differences between exception handling according to SVID and the point of view represented by the IEEE Standard, it is necessary to review the circumstances under which both developed. Many of the ideas in SVID trace their origins to the early days of UNIX, when it was first implemented on mainframe computers. These early environments have in common that rational floating-point operations `+`, `-`, `*` and `/` are atomic machine instructions, while `sqrt`, conversion to integral value in floating-point format, and elementary transcendental functions are subroutines composed of many atomic machine instructions.

Because these environments treat floating-point exceptions in varied ways, uniformity could only be imposed by checking arguments and results in software before and after each atomic floating-point instruction. Because this has too great an impact on performance, SVID does not specify the effect of floating-point exceptions such as division by zero or overflow.

Operations implemented by subroutines are slow compared to single atomic floating-point instructions; extra error checking of arguments and results has little performance impact; so such checking is required by the SVID. When exceptions are detected, default results are specified, `errno` is set to `EDOM` for improper operands, or `ERANGE` for results that overflow or underflow, and the function `matherr()` is called with a record containing details of the exception. This costs little on the machines for which UNIX was originally developed, but the value is correspondingly small because the far more common exceptions in the basic operations `+`, `-`, `*` and `/` are completely unspecified.

IEEE 754 History

The IEEE Standard explicitly states that compatibility with previous implementations was not a goal. Instead, an exception handling scheme was developed with efficiency and users' requirements in mind. This scheme is uniform across the simple rational operations (`+`, `-`, `*` and `/`), and more complicated operations such as remainder, square root, and conversion between formats. Although the Standard does not specify transcendental functions, the framers of the Standard anticipated that the same exception handling scheme would be applied to elementary transcendental functions in conforming systems.

Elements of IEEE exception handling include suitable default results and interruption of computation only when requested in advance.

SVID Future Directions

The current SVID, ("Edition 3" or "SVR4"), identifies certain directions for future development. One of these is compatibility with the IEEE Standard. In particular a future version of the SVID replaces references to `HUGE`, intended to be a large finite number, with `HUGE_VAL`, which is infinity on IEEE systems. `HUGE_VAL`, for instance, is returned as the result of floating-point overflows. The values returned by `libm` functions for input arguments that raise exceptions are those in the IEEE column in TABLE E-1. In addition, `errno` no longer needs to be set.

SVID Implementation

The following `libm` functions provide operand or result checking corresponding to SVID. The `sqrt` function is the only function that does not conform to SVID when called from a C program that uses the `libm` in-line expansion templates through `-xlibmil`, because this causes the hardware instruction for square root, `fsqrt[sd]`, to be used in place of a function call.

TABLE E-1 Exceptional Cases and `libm` Functions

Function	errno	error message	SVID	X/Open	IEEE
`acos(|x|>1)`	EDOM	DOMAIN	0.0	0.0	NaN
`acosh(x<1)`	EDOM	DOMAIN	NaN	NaN	NaN
`asin(|x|>1)`	EDOM	DOMAIN	0.0	0.0	NaN
`atan2((+-0,+-0)`	EDOM	DOMAIN	0.0	0.0	+-0.0,+-pi
`atanh(|x|>1)`	EDOM	DOMAIN	NaN	NaN	NaN
`atanh(+-1)`	EDOM/ERANGE	SING	+-HUGE (EDOM)	+-HUGE_VAL (ERANGE)	+-infinity
`cosh overflow`	ERANGE	-	HUGE	HUGE_VAL	infinity
`exp overflow`	ERANGE	-	HUGE	HUGE_VAL	infinity
`exp underflow`	ERANGE	-	0.0	0.0	0.0
`fmod(x,0)`	EDOM	DOMAIN	x	NaN	NaN
`gamma(0 or -integer)`	EDOM	SING	HUGE	HUGE_VAL	infinity
`gamma overflow`	ERANGE	-	HUGE	HUGE_VAL	infinity
`hypot overflow`	ERANGE	-	HUGE	HUGE_VAL	infinity
`j0(|x| > X_TLOSS)`	ERANGE	TLOSS	0.0	0.0	correct answer
`j1(|x| > X_TLOSS)`	ERANGE	TLOSS	0.0	0.0	correct answer
`jn(|x| > X_TLOSS)`	ERANGE	TLOSS	0.0	0.0	correct answer
`lgamma(0 or -integer)`	EDOM	SING	HUGE	HUGE_VAL	infinity
`lgamma overflow`	ERANGE	-	HUGE	HUGE_VAL	infinity
`log(0)`	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity

TABLE E-1 Exceptional Cases and `libm` Functions *(Continued)*

Function	errno	error message	SVID	X/Open	IEEE
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log1p(-1)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log1p(x<-1)	EDOM	DOMAIN	NaN	NaN	NaN
pow(0,0)	EDOM	DOMAIN	0.0	1.0 (no error)	1.0 (no error)
pow(NaN,0)	EDOM	DOMAIN	NaN	NaN	1.0 (no error)
pow(0,neg)	EDOM	DOMAIN	0.0	-HUGE_VAL	+-infinity
pow(neg, non-integer)	EDOM	DOMAIN	0.0	NaN	NaN
pow overflow	ERANGE	-	+-HUGE	+-HUGE_VAL	+-infinity
pow underflow	ERANGE	-	+-0.0	+-0.0	+-0.0
remainder(x,0)	EDOM	DOMAIN	NaN	NaN	NaN
scalb overflow	ERANGE	-	+-HUGE_VAL	+-HUGE_VAL	+-infinity
scalb underflow	ERANGE	-	+-0.0	+-0.0	+-0.0
sinh overflow	ERANGE	-	+-HUGE	+-HUGE_VAL	+-infinity
sqrt(x<0)	EDOM	DOMAIN	0.0	NaN	NaN
y0(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
y0(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y0(x > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	correct answer
y1(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
y1(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y1(x > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	correct answer
yn(n,0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
yn(n,x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
yn(n, x> X_TLOSS)	ERANGE	TLOSS	0.0	0.0	correct answer

General Notes on Exceptional Cases and `libm` Functions

TABLE E-1 lists all the `libm` functions affected by the standards. The value `X_TLOSS` is defined in `<values.h>`. SVID requires `<math.h>` to define `HUGE` as `MAXFLOAT`, which is approximately 3.4e+38. `HUGE_VAL` is defined as infinity in `libc. errno` is a global variable accessible to C and C++ programs.

`<errno.h>` defines 120 or so possible values for `errno`; the two used by the math library are `EDOM` for domain errors and `ERANGE` for range errors. See `intro`(3) and `perror`(3).

- The ANSI C compiler switches `-Xt`, `-Xa`, `-Xc`, `-Xs`, among other things, control the level of standards compliance that is enforced by the compiler. Refer to `cc`(1) for a description of these switches.
- As far as `libm` is concerned, `-Xt` and `-Xa` cause SVID and X/Open behavior, respectively. `-Xc` corresponds to strict ANSI C behavior.

  An additional switch `-<x>libmieee`, when specified, returns values in the spirit of IEEE 754. The default behavior for `libm` and `libsunmath` is to be SVID-compliant on Solaris 2.6, Solaris 7, and Solaris 8.
- For strict ANSI C (`-Xc`), `errno` is set always, `matherr`() is not called, and the X/Open value is returned.
- For SVID (`-Xt` or `-Xs`), the function `matherr`() is called with information about the exception. This includes the value that is the default SVID return value.

  A user-supplied `matherr`() could alter the return value; see `matherr`(3m). If there is no user-supplied `matherr()`, `libm` sets `errno`, possibly prints a message to standard error, and returns the value listed in the SVID column of TABLE E-1.
- For X/Open (`-Xa`), the behavior is the same as for the SVID, in that `matherr`() is invoked and `errno` set accordingly. However, no error message is written to standard error, and the X/Open return values are the same as IEEE return values in many cases.
- For the purposes of `libm` exception handling, `-Xs` behaves the same as `-Xt`. That is, programs compiled with `-Xs` use the SVID compliant versions of the `libm` functions listed in TABLE E-1.
- For efficiency, programs compiled with inline hardware floating-point do not do the extra checking required to set `EDOM` or call `matherr()`if `sqrt` encounters a negative argument. `NaN` is returned for the function value in situations where `EDOM` might otherwise be set.

  Thus, C programs that replace `sqrt()` function calls with `fsqrt[sd]` instructions conform to the IEEE Floating-Point Standard, but may no longer conform to the error handling requirements of the System V Interface Definition.

Notes on `libm`

SVID specifies two floating-point exceptions, `PLOSS` (partial loss of significance) and `TLOSS` (total loss of significance). Unlike `sqrt(-1)`, these have no inherent mathematical meaning, and unlike `exp(+-10000)`, these do not reflect inherent limitations of a floating-point storage format.

`PLOSS` and `TLOSS` reflect instead limitations of particular algorithms for `fmod` and for trigonometric functions that suffer abrupt declines in accuracy at definite boundaries.

Like most IEEE implementations, the `libm` algorithms do not suffer such abrupt declines, and so do not signal `PLOSS`. To satisfy the dictates of SVID compliance, the Bessel functions do signal `TLOSS` for large input arguments, although accurate results can be safely calculated.

The implementations of `sin`, `cos`, and `tan` treat the essential singularity at infinity like other essential singularities by returning a `NaN` and setting `EDOM` for infinite arguments.

Likewise SVID specifies that `fmod(x,y)` is be zero if `x/y` overflows, but the `libm` implementation of `fmod`, derived from the IEEE remainder function, does not compute `x/y` explicitly and hence always delivers an exact result.

LIA-1 Conformance

In this section, LIA-1 refers to ISO/IEC 10967-1:1994 Information Technology - Language Independent Arithmetic - Part 1: Integer and floating-point arithmetic.

The C and Fortran 95 compilers (`cc` and `f95`) contained in the Forte Developer compilers release conform to LIA-1 in the following senses (paragraph letters correspond to those in LIA-1 section 8):

a. TYPES (LIA 5.1): The LIA-1 conformant types are C int and Fortran `INTEGER`. Other types may conform as well, but they are not specified here. Further specifications for specific languages await language bindings to LIA-1 from the cognizant language standards organizations.

b. PARAMETERS (LIA 5.1):

```
#include <values.h> /* defines MAXINT */

#define TRUE 1

#define FALSE 0

#define BOUNDED TRUE
```

```
#define MODULO TRUE
#define MAXINT 2147483647
#define MININT -2147483648
                 logical bounded, modulo
                 integer maxint, minint
                 parameter (bounded = .TRUE.)
                 parameter (modulo = .TRUE.)
                 parameter (maxint = 2147483647)
                 parameter (minint = -2147483648)
```

d. DIV/REM/MOD (LIA 5.1.3):

C / and %, and Fortran / and mod(), provide `DIVtI(x,y)` and `REMtI(x,y)`. Also, `modaI(x,y)` is available with this code:

```
int modaI(int x, int y) {
                 int t = x % y;
                 if (y < 0 && t > 0)
                 t -= y;
                 else if (y > 0 && t < 0)
                 t += y;
                     return t;
                 }
```

or this:

```
                 integer function modaI(x, y)
                 integer x, y, t
                 t = mod(x, y)
                 if (y .lt. 0 .and. t .gt. 0) t = t - y
                 if (y .gt. 0 .and. t .lt. 0) t = t + y
                 modaI = t
                 return
                 end
```

i. NOTATION (LIA 5.1.3): The following table shows the notation by which the LIA integer operations may be realized.

TABLE E-2 LIA-1 Conformance - Notation

LIA	C	Fortran if different
`addI(x,y)`	`x+y`	
`subI(x,y)`	`x-y`	
`mulI(x,y)`	`x*y`	
`divtI(x,y)`	`x/y`	
`remtI(x,y)`	`x%y`	`mod(x,y)`
`modaI(x,y)`	`see above`	
`negI(x)`	`-x`	
`absI(x)`	`#include <stdlib.h>` `abs(x)`	`abs(x)`
`signI(x)`	`#define signI(x) (x > 0 ? 1 : (x < 0 ? -1 : 0))`	see below
`eqI(x,y)`	`x==y`	`x.eq.y`
`neqI(x,y)`	`x!=y`	`x.ne.y`
`lssI(x,y)`	`x<y`	`x.lt.y`
`leqI(x,y)`	`x<=y`	`x.le.y`
`gtrI(x,y)`	`x>y`	`x.gt.y`
`geqI(x,y)`	`x>=y`	`x.ge.y`

The following code shows the Fortran notation for `signI(x)`.

```
integer function signi(x)
integer x, t
if (x .gt. 0) t=1
if (x .lt. 0) t=-1
if (x .eq. 0) t=0
return
end
```

j. EXPRESSION EVALUATION: By default, when no optimization is specified, expressions are evaluated in `int` (C) or `INTEGER` (Fortran) precision. Parentheses are respected. The order of evaluation of associative unparenthesized expressions such as a + b + c or a * b * c is not specified.

k. METHOD OF OBTAINING PARAMETERS: Include the definitions above in the source code.

n. NOTIFICATION: Integer exceptions are x/0 and x%0 or mod(x,0). By default, these exceptions generate `SIGFPE`. When no signal handler is specified for `SIGFPE`, the process terminates and dumps memory.

o. SELECTION MECHANISM: `signal`(3) or `signal`(3F) may be used to enable user exception handling for `SIGFPE`.

References

The following manual provides more information about SPARC floating-point hardware:

SPARC Architecture Manual, Version 9, PTR Prentice Hall, New Jersey, 1994.

The remaining references are organized by chapter. Information on obtaining Standards documents and test programs is included at the end.

Chapter 2: “IEEE Arithmetic”

Cody et al., “A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic,” *IEEE Computer*, August 1984.

Coonen, J.T., “An Implementation Guide to a Proposed Standard for Floating Point Arithmetic”, *Computer*, Vol. 13, No. 1, Jan. 1980, pp 68-79.

Demmel, J., “Underflow and the Reliability of Numerical Software”, SIAM J. *Scientific Statistical Computing*, Volume 5 (1984), 887-919.

Hough, D., “Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic”, *Computer*, Vol. 13, No. 1, Jan. 1980, pp 70-74.

Kahan, W., and Coonen, J.T., “The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments”, published in *The Relationship between Numerical Computation and Programming Languages*, Reid, J.K., (editor), North-Holland Publishing Company, 1982.

Kahan, W., “Implementation of Algorithms”, *Computer Science Technical Report No. 20*, University of California, Berkeley CA, 1973. Available from National Technical Information Service, NTIS Document No. AD–769 124 (339 pages), 1-703-487-4650 (ordinary orders) or 1-800-336-4700 (rush orders.)

Karpinski, R., “Paranoia: a Floating-Point Benchmark”, *Byte*, February 1985.

Knuth, D.E., *The Art of Computer Programming, Vol.2: Semi-Numerical Algorithms*, Addison-Wesley, Reading, Mass, 1969, p 195.

Linnainmaa, S., “Combatting the effects of Underflow and Overflow in Determining Real Roots of Polynomials”, SIGNUM Newsletter 16, (1981), 11-16.

Rump, S.M., “How Reliable are Results of Computers?”, translation of “Wie zuverlassig sind die Ergebnisse unserer Rechenanlagen?”, *Jahrbuch Uberblicke Mathematik 1983*, pp 163-168, C Bibliographisches Institut AG 1984.

Sterbenz, P, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974. (Out of print; most university libraries have copies.)

Stevenson, D. et al., Cody, W., Hough, D. Coonen, J., various papers proposing and analyzing a draft standard for binary floating-point arithmetic, *IEEE Computer*, March 1981.

The Proposed IEEE Floating-Point Standard, special issue of the ACM *SIGNUM Newsletter*, October 1979.

Chapter 3: “The Math Libraries”

Cody, William J. and Waite, William, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1980.

Coonen, J.T., *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, PhD Dissertation, University of California, Berkeley, 1984.

Tang, Peter Ping Tak, *Some Software Implementations of the Functions Sin and Cos*, Technical Report ANL-90/3, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1990.

Tang, Peter Ping Tak, *Table-driven Implementations of the Exponential Function EXPM1 in IEEE Floating-Point Arithmetic*, Preprint MCS-P125-0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February, 1990.

Tang, Peter Ping Tak, *Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic*, ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989, pp 144-157 communication, July 18, 1988.

Tang, Peter Ping Tak, *Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic*, preprint MCS-P55-0289, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1989 (to appear in ACM Trans. on Math. Soft.)

Park, Stephen K. and Miller, Keith W., “Random Number Generators: Good Ones Are Hard To Find”, *Communications of the ACM*, Vol. 31, No. 10, October 1988, pp 1192 - 1201.

Chapter 4: “Exceptions and Signal Handling”

Coonen, J.T, “Underflow and the Denormalized Numbers”, *Computer*, 14, No. 3, March 1981, pp 75-87.

Demmel, J., and X. Li, “Faster Numerical Algorithms via Exception Handling”, *IEEE Trans. Comput.* Vol. 48, No. 8, August 1994, pp 983-992.

Kahan, W., “A Survey of Error Analysis”, *Information Processing 71*, North-Holland, Amsterdam, 1972, pp 1214-1239.

Appendix B: “SPARC Behavior and Implementation”

The following manufacturer’s documentation provides more information about the floating-point hardware and main processor chips. It is organized by system architecture.

Texas Instruments, *SN74ACT8800 Family, 32-Bit CMOS Processor Building Blocks: Date Manual*, 1st edition, Texas Instruments Incorporated, 1988.

Weitek, *WTL 3170 Floating Point Coprocessor: Preliminary Data*, 1988, published by Weitek Corporation, 1060 E. Arques Avenue, Sunnyvale, CA 94086.

Weitek, *WTL 1164/WTL 1165 64-bit IEEE Floating Point Multiplier/Divider and ALU: Preliminary Data*, 1986, published by Weitek Corporation, 1060 E. Arques Avenue, Sunnyvale, CA 94086.

Standards

American National Standard for Information Systems – Programming Language C (ANSI C), Document no. X3.159-1989, American National Standards Institute, 1430 Broadway, New York, NY 10018.

IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985 (IEEE 754), published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1985.

IEEE Standard Glossary of Mathematics of Computing Terminology, ANSI/IEEE Std 1084-1986, published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1986.

IEEE Standard Portable Operating System Interface for Computer Environments (POSIX), IEEE Std 1003.1-1988, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017.

System V Application Binary Interface (ABI), AT&T (1-800-432-6600), 1989.

SPARC System V ABI Supplement (SPARC ABI), AT&T (1-800-432-6600), 1990.

System V Interface Definition, 3rd edition, (SVID89, or SVID Issue 3), Volumes I–IV, Part number 320-135, AT&T (1-800-432-6600), 1989.

X/OPEN Portability Guide, Set of 7 Volumes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1989.

Test Programs

A number of test programs for floating-point arithmetic and math libraries are available from Netlib in the ucbtest package. These programs include versions of `Paranoia`, Z. Alex Liu's Berkeley Elementary Function test program, the IEEE test vectors, and programs based on number-theoretic methods developed by Prof. W. Kahan that generate hard test cases for correctly rounded multiplication, division, and square root.

ucbtest is located at `http://www.netlib.org/fp/ucbtest.tgz`.

Glossary

This glossary describes computer floating-point arithmetic terms. It also describes terms and acronyms associated with parallel processing.

This symbol, "||", appended to a term designates it as associated with parallel processing.

accuracy	A measure of how well one number approximates another. For example, the accuracy of a computed result often reflects the extent to which errors in the computation cause it to differ from the mathematically exact result. Accuracy can be expressed in terms of significant digits (e.g., "The result is accurate to six digits") or more generally in terms of the preservation of relevant mathematical properties (e.g., "The result has the correct algebraic sign").
array processing||	A number of processors working simultaneously, each handling one element of the array, so that a single operation can apply to all elements of the array in parallel.
associativity||	See *cache*, *direct* `mapped` *cache*, *fully associative cache*, *set associative cache*.
asynchronous control||	Computer control behavior where a specific operation is begun upon receipt of an indication (signal) that a particular event has occurred. Asynchronous control relies on synchronization mechanisms called locks to coordinate processors. See also *mutual exclusion*, *mutex lock*, *semaphore lock*, *single-lock strategy*, *spin lock*.
barrier||	A synchronization mechanism for coordinating tasks even when data accesses are not involved. A barrier is analogous to a gate. Processors or threads operating in parallel reach the gate at different times, but none can pass through until all processors reach the gate. For example, suppose at the end of each day, all bank tellers are required to tally the amount of money that was deposited, and the amount that was withdrawn. These totals are then reported to the bank vice president, who must check the grand totals to verify debits equal credits. The tellers operate at their own speeds; that is, they finish totaling their transactions at different times. The barrier mechanism prevents

	tellers from leaving for home before the grand total is checked. If debits do not equal credits, all tellers must return to their desks to find the error. The barrier is removed after the vice president obtains a satisfactory grand total.
biased exponent	The sum of the base-2 exponent and a constant (bias) chosen to make the stored exponent's range non-negative. For example, the exponent of 2^{-100} is stored in IEEE single precision format as (-100) + (single precision bias of 127) = 27.
binade	The interval between any two consecutive powers of two.
blocked state$^{||}$	A thread is waiting for a resource or data; such as, return data from a pending disk read, or waiting for another thread to unlock a resource.
bound threads$^{||}$	For Solaris threads, a thread permanently assigned to a particular LWP is called a bound thread. Bound threads can be scheduled on a real-time basis in strict priority with respect to all other active threads in the system, not only within a process. An LWP is an entity that can be scheduled with the same default scheduling priority as any UNIX process.
cache$^{||}$	Small, fast, hardware-controlled memory that acts as a buffer between a processor and main memory. Cache contains a copy of the most recently used memory locations—addresses and contents—of instructions and data. Every address reference goes first to cache. If the desired instruction or data is not in cache, a cache miss occurs. The contents are fetched across the bus from main memory into the CPU register specified in the instruction being executed and a copy is also written to cache. It is likely that the same location will be used again soon, and, if so, the address is found in cache, resulting in a cache hit. If a write to that address occurs, the hardware not only writes to cache, but can also generate a write-through to main memory.
	See also *associativity*, *circuit switching*, *direct mapped cache*, *fully associative cache*, *MBus*, *packet switching*, *set associative cache*, *write-back*, *write-through*, *XDBus*.
cache locality$^{||}$	A program does not access all of its code or data at once with equal probability. Having recently accessed information in cache increases the probability of finding information locally without having to access memory. The principle of locality states that programs access a relatively small portion of their address space at any instant of time. There are two different types of locality: temporal and spatial.
	Temporal locality (locality in time) is the tendency to reuse recently accessed items. For example, most programs contain loops, so that instructions and data are likely to be accessed repeatedly. Temporal locality retains recently accessed items closer to the processor in cache rather than requiring a memory access. See also *cache*, *competitive-caching*, *false sharing*, *write-invalidate*, *write-update*.
	Spatial locality (locality in space) is the tendency to reference items whose addresses are close to other recently accessed items. For example, accesses to elements of an array or record show a natural spatial locality. Caching takes

	advantage of spatial locality by moving blocks (multiple contiguous words) from memory into cache and closer to the processor. See also *cache*, *competitive-caching, false sharing, write-invalidate, write-update*.
chaining	A hardware feature of some pipeline architectures that allows the result of an operation to be used immediately as an operand for a second operation, simultaneously with the writing of the result to its destination register. The total cycle time of two chained operations is less than the sum of the stand-alone cycle times for the instructions. For example, the TI 8847 supports chaining of consecutive `fadd`, `fsub`, and `fmul` (of the same precision). Chained `faddd/fmuld` requires 12 cycles, while consecutive unchained `faddd/fmuld` requires 17 cycles.
circuit switching||	A mechanism for caches to communicate with each other as well as with main memory. A dedicated connection (circuit) is established between caches or between cache and main memory. While a circuit is in place no other traffic can travel over the bus.
coherence||	In systems with multiple caches, the mechanism that ensures that all processors see the same image of memory at all times.
common exceptions	The three floating point exceptions overflow, invalid, and division are collectively referred to as the common exceptions for the purposes of `ieee_flags`(3m) and `ieee_handler`(3m). They are called common exceptions because they are commonly trapped as errors.
competitive-caching||	Competitive-caching maintains cache coherence by using a hybrid of write-invalidate and write-update. Competitive-caching uses a counter to age shared data. Shared data is purged from cache based on a least-recently-used (LRU) algorithm. This can cause shared data to become private data again, thus eliminating the need for the cache coherency protocol to access memory (via backplane bandwidth) to keep multiple copies synchronized. See also *cache, cache locality, false sharing, write-invalidate, write-update*.
concurrency||	The execution of two or more active threads or processes in parallel. On a uniprocessor apparent concurrence is accomplished by rapidly switching between threads. On a multiprocessor system true parallel execution can be achieved. See also *asynchronous control, multiprocessor system, thread*.
concurrent processes||	Processes that execute in parallel in multiple processors or asynchronously on a single processor. Concurrent processes can interact with each other, and one process can suspend execution pending receipt of information from another process or the occurrence of an external event. See also *process, sequential processes*.
condition variable||	For Solaris threads, a condition variable enables threads to atomically block until a condition is satisfied. The condition is tested under the protection of a mutex lock. When the condition is false, a thread blocks on a condition variable

and atomically releases the mutex waiting for the condition to change. When another thread changes the condition, it can signal the associated condition variable to cause one or more waiting threads to wake up, reacquire the mutex, and re-evaluate the condition. Condition variables can be used to synchronize threads in this process and other processes if the variable is allocated in memory that is writable and shared among the cooperating processes and have been initialized for this behavior.

context switch In multitasking operating systems, such as the SunOS™ operating system, processes run for a fixed time quantum. At the end of the time quantum, the CPU receives a signal from the timer, interrupts the currently running process, and prepares to run a new process. The CPU saves the registers for the old process, and then loads the registers for the new process. Switching from the old process state to the new is known as a context switch. Time spent switching contexts is system overhead; the time required depends on the number of registers, and on whether there are special instructions to save the registers associated with a process.

control flow model$^{||}$ The von Neumann model of a computer. This model specifies flow of control; that is, which instruction is executed at each step of a program. All Sun workstations are instances of the von Neumann model. See also *data flow model, demand-driven dataflow*.

critical region$^{||}$ An indivisible section of code that can only be executed by one thread at a time and is not interruptible by other threads; such as, code that accesses a shared variable. See also *mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock*.

critical resource$^{||}$ A resource that can only be in use by at most one thread at any given time. Where several asynchronous threads are required to coordinate their access to a critical resource, they do so by synchronization mechanisms. See also *mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock*.

data flow model$^{||}$ This computer model specifies what happens to data, and ignores instruction order. That is, computations move forward by nature of availability of data values instead of the availability of instructions. See also *control flow model, demand-driven dataflow*.

data race$^{||}$ In multithreading, a situation where two or more threads simultaneously access a shared resource. The results are indeterminate depending on the order in which the threads accessed the resource. This situation, called a data race, can produce different results when a program is run repeatedly with the same input. See also *mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock*.

deadlock$^{||}$ A situation that can arise when two (or more) separately active processes compete for resources. Suppose that process P requires resources X and Y and requests their use in that order at the same time that process Q requires resources Y and X and asks for them in that order. If process P has acquired

	resource X and simultaneously process Q has acquired resource Y, then neither process can proceed—each process requires a resource that has been allocated to the other process.
default result	The value that is delivered as the result of a floating-point operation that caused an exception.
demand-driven dataflow$^{||}$	A task is enabled for execution by a processor when its results are required by another task that is also enabled; such as, a graph reduction model. A graph reduction program consists of reducible expressions that are replaced by their computed values as the computation progresses through time. Most of the time, the reductions are done in parallel—nothing prevents parallel reductions except the availability of data from previous reductions. See also *control flow model, data flow model*.
denormalized number	Older nomenclature for subnormal number.
direct mapped cache$^{||}$	A direct mapped cache is a one-way set associative cache. That is, each cache entry holds one block and forms a *single* set with *one* element. See also *cache, cache locality, false sharing, fully associative cache, set associative cache, write-invalidate, write-update*.
distributed memory architecture$^{||}$	A combination of local memory and processors at each node of the interconnect network topology. Each processor can directly access only a portion of the total memory of the system. Message passing is used to communicate between any two processors, and there is no global, shared memory. Therefore, when a data structure must be shared, the program issues send/receive messages to the process that owns that structure. See also *interprocess communication, message passing*.
double precision	Using two words to represent a number in order to keep or increase precision. On SPARC workstations, double precision is the 64-bit IEEE double precision.
exception	An arithmetic exception arises when an attempted atomic arithmetic operation has no result that is acceptable universally. The meanings of atomic and acceptable vary with time and place.
exponent	The component of a floating-point number that signifies the integer power to which the base is raised in determining the value of the represented number.
false sharing$^{||}$	A condition that occurs in cache when two unrelated data accessed independently by two threads reside in the same block. This block can end up 'ping-ponging' between caches for no valid reason. Recognizing such a case and rearranging the data structure to eliminate the false sharing greatly increases cache performance. See also *cache, cache locality*.

floating-point number system A system for representing a subset of real numbers in which the spacing between representable numbers is not a fixed, absolute constant. Such a system is characterized by a base, a sign, a significand, and an exponent (usually biased). The value of the number is the signed product of its significand and the base raised to the power of the unbiased exponent.

fully associative cache|| A fully associative cache with *m* entries is an *m*-way set associative cache. That is, it has a *single* set with *m* blocks. A cache entry can reside in any of the *m* blocks within that set. See also *cache, cache locality, direct mapped cache, false sharing, set associative cache, write-invalidate, write-updat*e.

gradual underflow When a floating-point operation underflows, return a subnormal number instead of 0. This method of handling underflow minimizes the loss of accuracy in floating-point calculations on small numbers.

hidden bits Extra bits used by hardware to ensure correct rounding, not accessible by software. For example, IEEE double precision operations use three hidden bits to compute a 56-bit result that is then rounded to 53 bits.

IEEE Standard 754 The standard for binary floating-point arithmetic developed by the Institute of Electrical and Electronics Engineers, published in 1985.

in-line template A fragment of assembly language code that is substituted for the function call it defines, during the inlining pass of Forte Developer compilers. Used (for example) by the math library in in-line template files (`libm.il`) in order to access hardware implementations of trigonometric functions and other elementary functions from C programs.

interconnection network topology|| Interconnection topology describes how the processors are connected. All networks consist of switches whose links go to processor-memory nodes and to other switches. There are four generic forms of topology: star, ring, bus, and fully-connected network. Star topology consists of a single hub processor with the other processors directly connected to the single hub, the non-hub processors are not directly connected to each other. In ring topology all processors are on a ring and communication is generally in one direction around the ring. Bus topology is noncyclic, with all nodes connected; consequently, traffic travels in both directions, and some form of arbitration is needed to determine which processor can use the bus at any particular time. In a fully-connected (crossbar) network, every processor has a bidirectional link to every other processor.

Commercially-available parallel processors use multistage network topologies. A multistage network topology is characterized by 2-dimensional grid, and boolean n-cube.

interprocess communication	Message passing among active processes. See also *circuit switching*, *distributed memory architecture*, *MBus*, *message passing*, *packet switching*, *shared memory*, *XDBus*.
IPC	See *interprocess communication*.
light-weight process	Solaris threads are implemented as a user-level library, using the kernel's threads of control, that are called light-weight processes (LWPs). In the Solaris environment, a process is a collection of LWPs that share memory. Each LWP has the scheduling priority of a UNIX process and shares the resources of that process. LWPs coordinate their access to the shared memory by using synchronization mechanisms such as locks. An LWP can be thought of as a virtual CPU that executes code or system calls. The threads library schedules threads on a pool of LWPs in the process, in much the same way as the kernel schedules LWPs on a pool of processors. Each LWP is independently dispatched by the kernel, performs independent system calls, incurs independent page faults, and runs in parallel on a multiprocessor system. The LWPs are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority.
lock	A mechanism for enforcing a policy for serializing access to shared data. A thread or process uses a particular lock in order to gain access to shared memory protected by that lock. The locking and unlocking of data is voluntary in the sense that only the programmer knows what must be locked. See also *data race, mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock*.
LWP	See *light-weight process*.
MBus	MBus is a bus specification for a processor/memory/IO interconnect. It is licensed by SPARC International to several silicon vendors who produce interoperating CPU modules, IO interfaces and memory controllers. MBus is a circuit-switched protocol combining read requests and response on a single bus. MBus level I defines uniprocessor signals; MBus level II defines multiprocessor extensions for the write-invalidate cache coherence mechanism.
memory	A medium that can retain information for subsequent retrieval. The term is most frequently used for referring to a computer's internal storage that can be directly addressed by machine instructions. See also *cache*, *distributed memory, shared memory.*
message passing	In the distributed memory architecture, a mechanism for processes to communicate with each other. There is no shared data structure in which they deposit messages. Message passing allows a process to send data to another process and for the intended recipient to synchronize with the arrival of the data.
MIMD	See *Multiple Instruction Multiple Data, shared memory.*

mt-safe In the Solaris environment, function calls inside libraries are either mt-safe or not mt-safe; mt-safe code is also called “re-entrant” code. That is, several threads can simultaneously call a given function in a module and it is up to the function code to handle this. The assumption is that data shared between threads is only accessed by module functions. If mutable global data is available to clients of a module, appropriate locks must also be made visible in the interface. Furthermore, the module function cannot be made re-entrant unless the clients are assumed to use the locks consistently and at appropriate times. See also *single-lock strategy*.

Multiple Instruction Multiple Data System model where many processors can be simultaneously executing different instructions on different data. Furthermore, these processors operate in a largely autonomous manner as if they are separate computers. They have no central controller, and they typically do not operate in lock-step fashion. Most real world banks run this way. Tellers do not consult with one another, nor do they perform each step of every transaction at the same time. Instead, they work on their own, until a data access conflict occurs. Processing of transactions occurs without concern for timing or customer order. But customers A and B must be explicitly prevented from simultaneously accessing the joint AB account balance. MIMD relies on synchronization mechanisms called locks to coordinate access to shared resources. See also *mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock*.

multiple read single write In a concurrent environment, the first process to access data for writing has exclusive access to it, making concurrent write access or simultaneous read and write access impossible. However, the data can be read by multiple readers.

multiprocessor See *multiprocessor system*.

multiprocessor bus In a shared memory multiprocessor machine each CPU and cache module are connected together via a bus that also includes memory and IO connections. The bus enforces a cache coherency protocol. See also *cache, coherence, Mbus, XDBus*.

multiprocessor system A system in which more than one processor can be active at any given time. While the processors are actively executing separate processes, they run completely asynchronously. However, synchronization between processors is essential when they access critical system resources or critical regions of system code. See also *critical region, critical resource, multithreading, uniprocessor system*.

multitasking In a uniprocessor system, a large number of threads appear to be running in parallel. This is accomplished by rapidly switching between threads.

multithreading||	Applications that can have more than one thread or processor active at one time. Multithreaded applications can run in both uniprocessor systems and multiprocessor systems. See also *bound thread, mt-safe, single-lock strategy, thread, unbound thread, uniprocessor*.
mutex lock||	Synchronization variable to implement the mutual exclusion mechanism. See also *condition variable, mutual exclusion*.
mutual exclusion||	In a concurrent environment, the ability of a thread to update a critical resource without accesses from competing threads. See also *critical region, critical resource*.
NaN	Stands for Not a Number. A symbolic entity that is encoded in floating-point format.
normal number	In IEEE arithmetic, a number with a biased exponent that is neither zero nor maximal (all 1’s), representing a subset of the normal range of real numbers with a bounded small relative error.
packet switching||	In the shared memory architecture, a mechanism for caches to communicate with each other as well as with main memory. In packet switching, traffic is divided into small segments called packets that are multiplexed onto the bus. A packet carries identification that enables cache and memory hardware to determine whether the packet is destined for it or to send the packet on to its ultimate destination. Packet switching allows bus traffic to be multiplexed and unordered (not sequenced) packets to be put on the bus. The unordered packets are reassembled at the destination (cache or main memory). See also *cache, shared memory*.
paradigm||	A model of the world that is used to formulate a computer solution to a problem. Paradigms provide a context in which to understand and solve a real-world problem. Because a paradigm is a model, it abstracts the details of the problem from the reality, and in doing so, makes the problem easier to solve. Like all abstractions, however, the model can be inaccurate because it only approximates the real world. See also *Multiple Instruction Multiple Data, Single Instruction Multiple Data, Single Instruction Single Data, Single Program Multiple Data*.
parallel processing||	In a multiprocessor system, true parallel execution is achieved where a large number of threads or processes can be active at one time. See also *concurrence, multiprocessor system, multithreading, uniprocesso*r.
parallelism||	See *concurrent processes, multithreading*.
pipeline||	If the total function applied to the data can be divided into distinct processing phases, different portions of data can flow along from phase to phase; such as a compiler with phases for lexical analysis, parsing, type checking, code generation and so on. As soon as the first program or module has passed the

	lexical analysis phase, it can be passed on to the parsing phase while the lexical analyzer starts on the second program or module. See also *array processing, vector processing*.
pipelining	A hardware feature where operations are reduced to multiple stages, each of which takes (typically) one cycle to complete. The pipeline is filled when new operations can be issued each cycle. If there are no dependencies among instructions in the pipe, new results can be delivered each cycle. Chaining implies pipelining of dependent instructions. If dependent instructions cannot be chained, when the hardware does not support chaining of those particular instructions, then the pipeline stalls.
precision	A quantitative measure of the density of representable numbers. For example, in a binary floating point format that has a precision of 53 significant bits, there are 2^{53} representable numbers between any two adjacent powers of two (within the range of normal numbers). Do not confuse precision with accuracy, which expresses how closely one number approximates another.
process$^{\|\|}$	A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources.
quiet NaN	A NaN (not a number) that propagates through almost every arithmetic operation without raising new exceptions.
radix	The base number of any system of numbers. For example, 2 is the radix of a binary system, and 10 is the radix of the decimal system of numeration. SPARC workstations use radix-2 arithmetic; IEEE Std 754 is a radix-2 arithmetic standard.
round	Inexact results must be rounded up or down to obtain representable values. When a result is rounded up, it is increased to the next representable value. When rounded down, it is reduced to the preceding representable value.
roundoff error	The error introduced when a real number is rounded to a machine-representable number. Most floating-point calculations incur roundoff error. For any one floating-point operation, IEEE Std 754 specifies that the result shall not incur more than one rounding error.
semaphore lock$^{\|\|}$	Synchronization mechanism for controlling access to critical resources by cooperating asynchronous threads. See also *semaphore*.
semaphore$^{\|\|}$	A special-purpose data type introduced by E. W. Dijkstra that coordinates access to a particular resource or set of shared resources. A semaphore has an integer value (that cannot become negative) with two operations allowed on it. The signal (`V` or `up`) operation increases the value by one, and in general indicates that a resource has become free. The wait (`P` or `down`) operation decreases the value by one, when that can be done without the value going negative, and in general indicates that a free resource is about to start being used. See also *semaphore lock*.

sequential processes||	Processes that execute in such a manner that one must finish before the next begins. See also *concurrent processes, process*.
set associative cache||	In a set associative cache, there are a fixed number of locations (at least two) where each block can be placed. A set associative cache with *n* locations for a block is called an *n*-way set associative cache. An *n*-way set associative cache consists of more than one set, each of which consists of *n* blocks. A block can be placed in any location (element) of that set. Increasing the associativity level (number of blocks in a set) increases the cache hit rate. See also *cache, cache locality, false sharing, write-invalidate, write-update*.
shared memory architecture||	In a bus-connected multiprocessor system, processes or threads communicate through a global memory shared by all processors. This shared data segment is placed in the address space of the cooperating processes between their private data and stack segments. Subsequent tasks spawned by `fork()` copy all but the shared data segment in their address space. Shared memory requires program language extensions and library routines to support the model.
signaling NaN	A NaN (not a number) that raises the invalid operation exception whenever it appears as an operand.
significand	The component of a floating-point number that is multiplied by a signed power of the base to determine the value of the number. In a normalized number, the significand consists of a single nonzero digit to the left of the radix point and a fraction to the right.
SIMD||	See *Single Instruction Multiple Data*.
Single Instruction Multiple Data||	System model where there are many processing elements, but they are designed to execute the same instruction at the same time; that is, one program counter is used to sequence through a single copy of the program. SIMD is especially useful for solving problems that have lots of data that needs to be updated on a wholesale basis; such as numerical calculations that are regular. Many scientific and engineering applications (such as, image processing, particle simulation, and finite element methods) naturally fall into the SIMD paradigm. See also *array processing, pipeline, vector processing*.
Single Instruction Single Data||	The conventional uniprocessor model, with a single processor fetching and executing a sequence of instructions that operate on the data items specified within them. This is the original von Neumann model of the operation of a computer.
single precision	Using one computer word to represent a number.

Single Program Multiple Data|| A form of asynchronous parallelism where simultaneous processing of different data occurs without lock-step coordination. In SPMD, processors can execute different instructions at the same time; such as, different branches of an `if-then-else` statement.

single-lock strategy|| In the single-lock strategy, a thread acquires a single, application-wide mutex lock whenever any thread in the application is running and releases the lock before the thread blocks. The single-lock strategy requires cooperation from all modules and libraries in the system to synchronize on the single lock. Because only one thread can be accessing shared data at any given time, each thread has a consistent view of memory. This strategy is quite effective in a uniprocessor, provided shared memory is put into a consistent state before the lock is released and that the lock is released often enough to allow other threads to run. Furthermore, in uniprocessor systems, concurrency is diminished if the lock is not dropped during most I/O operations. The single-lock strategy cannot be applied in a multiprocessor system.

SISD|| See *Single Instruction Single Data*.

snooping|| The most popular protocol for maintaining cache coherency is called snooping. Cache controllers monitor or snoop on the bus to determine whether or not the cache contains a copy of a shared block.

For reads, multiple copies can reside in the cache of different processors, but because the processors need the most recent copy, all processors must get new values after a write. See also *cache, competitive-caching, false sharing, write-invalidate, write-update*.

For writes, a processor must have exclusive access to write to cache. Writes to unshared blocks do not cause bus traffic. The consequence of a write to shared data is either to invalidate all other copies or to update the shared copies with the value being written. See also *cache, competitive-caching, false sharing, write-invalidate, write-update*.

spin lock|| Threads use a spin lock to test a lock variable over and over until some other task releases the lock. That is, the waiting thread spins on the lock until the lock is cleared. Then, the waiting thread sets the lock while inside the critical region. After work in the critical region is complete, the thread clears the spin lock so another thread can enter the critical region. The difference between a spin lock and a mutex is that an attempt to get a mutex held by someone else will block and release the LWP; a spin lock does not release the LWP. See also *mutex lock*.

SPMD|| See *Single Program Multiple Data*.

stderr Standard Error is the Unix file pointer to standard error output. This file is opened when a program is started.

Store 0 Flushing the underflowed result of an arithmetic operation to zero.

subnormal number	In IEEE arithmetic, a nonzero floating point number with a biased exponent of zero. The subnormal numbers are those between zero and the smallest normal number.
thread$^{\Vert}$	A flow of control within a single UNIX process address space. Solaris threads provide a light-weight form of concurrent task, allowing multiple threads of control in a common user-address space, with minimal scheduling and communication overhead. Threads share the same address space, file descriptors (when one thread opens a file, the other threads can read it), data structures, and operating system state. A thread has a program counter and a stack to keep track of local variables and return addresses. Threads interact through the use of shared data and thread synchronization operations. See also *bound thread, light-weight processes, multithreading, unbound thread*.
topology$^{\Vert}$	See *interconnection network topology*.
two's complement	The radix complement of a binary numeral, formed by subtracting each digit from 1, then adding 1 to the least significant digit and executing any required carries. For example, the two's complement of 1101 is 0011.
ulp	Stands for unit in last place. In binary formats, the least significant bit of the significand, bit 0, is the unit in the last place.
ulp(x)	Stands for `ulp` of x truncated in working format.
unbound threads$^{\Vert}$	For Solaris threads, threads scheduled onto a pool of LWPs are called unbound threads. The threads library invokes and assigns LWPs to execute runnable threads. If the thread becomes blocked on a synchronization mechanism (such as a mutex lock) the state of the thread is saved in process memory. The threads library then assigns another thread to the LWP. See also *bound thread, multithreading, thread*.
underflow	A condition that occurs when the result of a floating-point arithmetic operation is so small that it cannot be represented as a normal number in the destination floating-point format with only normal roundoff.
uniprocessor system$^{\Vert}$	A uniprocessor system has only one processor active at any given time. This single processor can run multithreaded applications as well as the conventional single instruction single data model. See also *multithreading, single instruction single data, single-lock strategy*.
vector processing$^{\Vert}$	Processing of sequences of data in a uniform manner, a common occurrence in manipulation of matrices (whose elements are vectors) or other arrays of data. This orderly progression of data can capitalize on the use of pipeline processing. See also *array processing, pipeline*.
word	An ordered set of characters that are stored, addressed, transmitted and operated on as a single entity within a given computer. In the context of SPARC workstations, a word is 32 bits.

wrapped number	In IEEE arithmetic, a number created from a value that otherwise overflows or underflows by adding a fixed offset to its exponent to position the wrapped value in the normal number range. Wrapped results are not currently produced on SPARC workstations.
write-back$^{\|\|}$	Write policy for maintaining coherency between cache and main memory. Write-back (also called copy back or store in) writes only to the block in local cache. Writes occur at the speed of cache memory. The modified cache block is written to main memory only when the corresponding memory address is referenced by another processor. The processor can write within a cache block multiple times and writes it to main memory only when referenced. Because every write does not go to memory, write-back reduces demands on bus bandwidth. See also *cache, coherence, write-through*.
write-invalidate$^{\|\|}$	Maintains cache coherence by reading from local caches until a write occurs. To change the value of a variable the writing processor first invalidates all copies in other caches. The writing processor is then free to update its local copy until another processor asks for the variable. The writing processor issues an invalidation signal over the bus and all caches check to see if they have a copy; if so, they must invalidate the block containing the word. This scheme allows multiple readers, but only a single writer. Write-invalidate use the bus only on the first write to invalidate the other copies; subsequent local writes do not result in bus traffic, thus reducing demands on bus bandwidth. See also *cache, cache locality, coherence, false sharing, write-update*.
write-through$^{\|\|}$	Write policy for maintaining coherency between cache and main memory. Write-through (also called store through) writes to main memory as well as to the block in local cache. Write-through has the advantage that main memory has the most current copy of the data. See also *cache, coherence, write-back*.
write-update$^{\|\|}$	Write-update, also known as write-broadcast, maintains cache coherence by immediately updating all copies of a shared variable in all caches. This is a form of write-through because all writes go over the bus to update copies of shared data. Write-update has the advantage of making new values appear in cache sooner, which can reduce latency. See also *cache, cache locality, coherence, false sharing, write-invalidate*.
XDBus$^{\|\|}$	The XDBus specification uses low-impedance GTL (Gunning Transceiver Logic) transceiver signalling to drive longer backplanes at higher clock rates. XDBus supports a large number of CPUs with multiple interleaved memory banks for increased throughput. XDBus uses a packet switched protocol with split requests and responses for more efficient bus utilization. XDBus also defines an interleaving scheme so that one, two or four separate bus data paths can be used as a single backplane for increased throughput. XDBus supports write-invalidate, write-update and competitive-caching coherency schemes, and has several congestion control mechanisms. See also *cache, coherence, competitive-caching, write-invalidate, write-update*.

Index