Numerical Computation Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-7639-10
Revision A, August 1994

Contents

Figures

Tables

Preface

Purpose

This manual describes the floating-point software and hardware for the SPARC and x86 architectures. It is primarily a reference manual designed to accompany the DevPro 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 itself. See Appendix E, "References" for a brief bibliography on IEEE arithmetic.

This guide describes the floating-point software and hardware that runs under Solaris 2.*x*. The compiler products referred to in this guide are components of SPARCompiler 3.*x* release.

Operating Environment

SPARCompiler 3.0 and 3.0.1 run under SunOS 5.0 or later which is part of the Solaris 2.*x*® operating environment:

SPARCompiler 3.0.1 also runs under SunOS 4.1.1 or later which is part of the Solaris 1® for SPARC operating environment:

ProCompiler 3.0 and 2.0.1 run under SunOS 5.1 or later which is part of the Solaris 2.*x*® for x86 operating environment:

Audience

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

Organization

- Chapter 1, "Introduction" introduces the SPARCompiler floating-point environment and the IEEE Standard for Binary Floating-Point Arithmetic.
- Chapter 2, "IEEE Arithmetic" describes the IEEE arithmetic model, IEEE formats, and underflow.
- Chapter 3, "The Math Libraries" describes `libm` and `libsunmath`, the mathematics libraries.
- Chapter 4, "Exceptions and Signal Handling" describes exceptions and signal handling.
- Chapter 5, "Compile-Debug-Run" describes issues related to compiling, debugging, and running programs doing numerical computations.
- Appendix A, "Examples" contains example programs.
- Appendix B, "SPARC Behavior and Implementation" describes the floating-point hardware options for SPARC workstations.
- Appendix C, "x86 Behavior and Implementation" lists x86 and SPARC compatibilty issues related to the floating-point units used in x86 machines
- Appendix D, "Standards Compliance" discusses standards compliance.
- Appendix E, "References" includes a list of references and related documentation.
- "Glossary" contains a list of definition of terms.

The examples in this manual are in C and FORTRAN, but the concepts apply for either compiler on a SPARC or x86 workstation.

For more information on floating-point arithmetic, one may also refer to a paper on floating-point arithmetic called "*What Every Computer Scientist Should Know About Floating-Point Arithmetic*" by David Goldberg. This paper is located in the same location as the `README` file for any of the languages. The file is in PostScript and is called `floating-point.ps`. The paper is also available through the 3.0.1 Common Tools & Related Materials AnswerBook.

Notational Conventions

This manual uses the following notational conventions:

`Courier font plain face`

Represents what the system prints on the workstation screen, as well as keywords, identifiers, program names, file names, and names of libraries.

`Courier Boldface typewriter font`

Indicates commands that the user should type exactly as printed in the manual.

Italic font

Indicates variables or parameters that should be replaced with an appropriate word or string. It is also used for emphasis.

`demo%`

Represents the system prompt for a nonprivileged user account.

Screen boxes
Enclose a program listing or an interactive session. Any user input is indicated by **`boldface typewriter font`**.

Introduction 1

Sun's floating-point environment on SPARC and x86 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.

This chapter has the following organization:

Solaris for SPARC
Solaris for x86
Floating-Point Environment

This guide will refer to the following platforms:
- Solaris for SPARC
- Solaris for x86

Solaris for SPARC

Solaris 1 for SPARC includes SunOS 4.1.1 or later.

Solaris 2 for SPARC includes SunOS 5.0 or later.

Solaris for x86

Solaris 2 for x86 includes SunOS 5.1 or later.

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 will guide programmers to make full and efficient use of the IEEE mechanisms so that they may write application software effectively.

Many questions about floating-point arithmetic concern elementary operations on numbers. For example, one can ask the following questions:

- 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; or in some very old machines, the computation proceeds, but with garbage.

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

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

IEEE Arithmetic 2

This chapter discusses the IEEE Standard 754, the arithmetic model specified by the IEEE Standard for Binary Floating-Point Arithmetic (*IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985* (IEEE 754)). All SPARC-based and x86-based computers use IEEE arithmetic. All DevPro compiler products support the features of the IEEE arithmetic.

This chapter is organized into the following sections:

IEEE Arithmetic Model

What Is IEEE Arithmetic?

The IEEE Standard 754 specifies:

- Two basic floating-point formats, single and double.

  The IEEE single format has a precision of 24 bits (i.e., 24-bit significands), and 32 bits overall. The IEEE double format has a precision of 53 bits, and 64 bits overall.

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

Any format in the class of IEEE double extended formats has a precision at least of 64 bits, and 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 which 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 $-\infty$ (down); toward $+\infty$ (up); and toward zero (chop).

- Rounding precision, for example, if a system delivers results in double extended format, the user should nonetheless be able to specify that such results are to be rounded to the precision of either basic format, with trailing zeros.

The Standard supports user handling of exceptions, rounding and precision. Consequently the Standard supports interval arithmetic, and diagnosis of anomalies. The IEEE Standard 754 makes it possible to standardize elementary functions like `exp` and `cos`, to create very high-precision arithmetic, and to couple numerical and symbolic algebraic computation.

IEEE Standard 754 floating-point arithmetic offers users greater control over computation than does any other kind of floating-point arithmetic. The IEEE Standard 754 simplifies the task of writing numerically sophisticated, portable programs not only by imposing rigorous requirements on conforming implementations. The Standard also allows 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**
Single	float	REAL *or* REAL*4
Double	double	DOUBLE PRECISION *or* REAL*8
Double Extended	long double	REAL*16 [SPARC *only*]

IEEE Standard 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 format called double extended in Table 2-1 is one of the class of double extended formats defined by the IEEE standard.

The following sections describe in detail each of the three storage formats used for the IEEE floating-point formats.

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`.

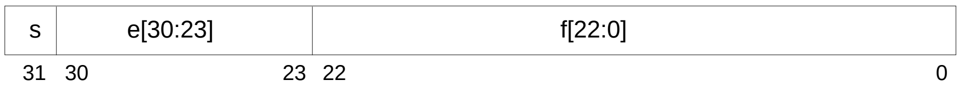

Figure 2-1 Single Storage Format

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*, i.e., the value of the indicated field is irrelevant to the determination of the value 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)

Table 2-2 Values Represented by Bit Patterns in IEEE Single Format *(Continued)*

Single Format Bit Pattern	**Value**
s = 0; e = 255; f = .000 — 00 (all bits in f are zero)	+INF (positive infinity)
s = 1; e = 255; f = .000 — 00 (all bits in f are zero)	–INF (negative infinity)
s = *u*; e = 255; f = .1*uuu* — *uu*	QNaN (quiet `NaNs`)
s = *u*; e = 255; f = .0*uuu* — *uu* ≠ 0 (at least one of the *u* in f is nonzero)	SNaN (signaling `NaNs`)

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 <= fraction < 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 appear 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 subnormal single format. The minimum positive normal number is often referred to as the underflow threshold.

Table 2-3 Some Bit Patterns in Single Storage Format and their IEEE Values

Common Name	**Bit Pattern (Hex)**	**Equivalent 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
quiet `NaN` (with least fraction)	`7fc00000`	NaN (or QNaN)
signaling `NaN` (with greatest fraction)	`7fbfffff`	NaN (or SNaN)

`NaN`s (Not a Number) can be represented with any of the many bit patterns that satisfy the definition of a `NaN`. The hex values of the `NaN`s shown in Table 2-3 are just two of the many bit patterns that can be used to represent `NaN`s.

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`.

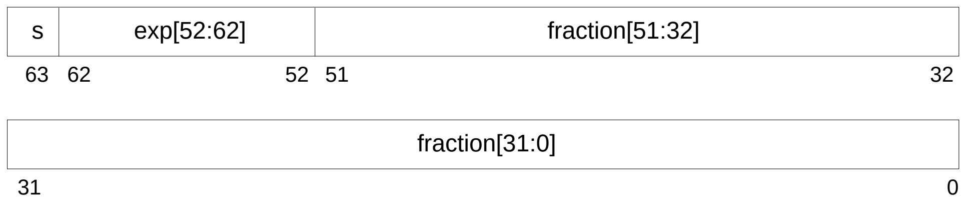

Figure 2-2 Double Storage Format

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*, i.e., 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**
$0 < e < 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$ (normal numbers)
$e = 0$; $f \neq 0$ (at least one bit in f is nonzero)	$(-1)^s \times 2^{-1022} \times 0.f$ (subnormal numbers)
$e = 0$; $f = 0$ (all bits in f are zero)	$(-1)^s \times 0.0$ (signed zero)
s = 0; e = 2047; f = .000 — 00 (all bits in f are zero)	+INF (positive infinity)
s = 1; e = 2047; f = .000 — 00 (all bits in f are zero)	–INF (negative infinity)
s = *u*; e = 2047; f = .1*uuu* — *uu*	QNaN (quiet NaNs)
s = *u*; e = 2047; f = .0*uuu* — *uu* ≠ 0 (at least one of the *u* in f is nonzero)	SNaN (signaling NaNs)

Notice that when 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 appear 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 IEEE double format.
- The minimum positive normal number is the smallest positive number which is representable in IEEE double format; it is certain that no precision has been lost.
- The maximum subnormal number is the largest number which is representable in IEEE double format; it is possible that some precision has been lost.
- The minimum positive subnormal number is the smallest positive number which is representable in IEEE double format; it is possible that some precision has been lost.

Table 2-5 Some Bit Patterns in Double Storage Format and their IEEE Values

Common Name	Bit Pattern (Hex)	Equivalent 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

Table 2-5 Some Bit Patterns in Double Storage Format and their IEEE Values

Common Name	Bit Pattern (Hex)	Equivalent Value
quiet NaN with greatest fraction	`7fffffff ffffffff`	QNaN
quiet NaN with least fraction	`7ff80000 00000000`	QNaN
signaling NaN with greatest fraction	`7ff7ffff ffffffff`	SNaN
signaling NaN with least fraction	`7ff00000 80000001`	SNaN

`NaN`s (Not a Number) can be represented by any of the many bit patterns that satisfy the definition of `NaN`. The hex values of the `NaN`s shown in Table 2-5 illustrate the point that the leading (i.e. most significant) bit of the fraction field determines whether a `NaN` is quiet (leading fraction bit = 1) or signaling (leading fraction bit = 0).

Double Extended Format (SPARC Architecture)

This floating-point environment's quadruple precision format conforms to the IEEE definition of double extended formats. It occupies four 32-bit words. It 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 Table 2-3. The highest 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. The next two 32-bit words (counting downwards) contain `f[63:32]` and `f[95:64]`. 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`.

Table 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`.

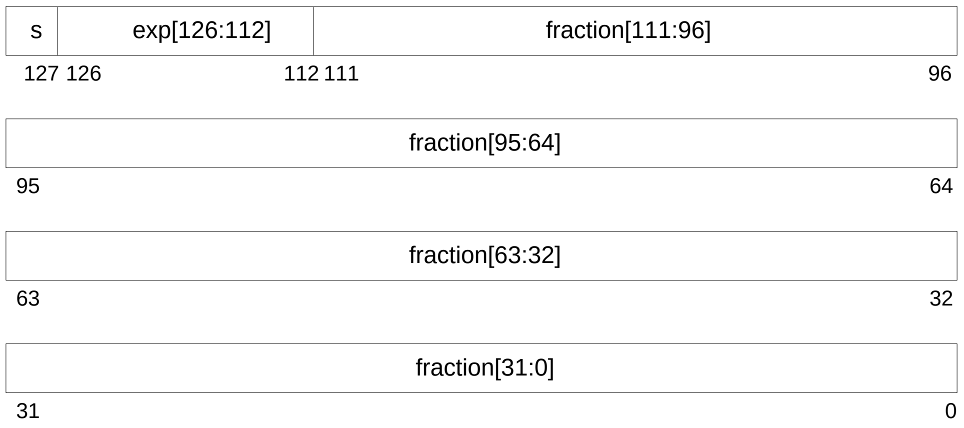

Figure 2-3 Double Extended Format (SPARC)

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, on the one hand, and the value represented by the bit pattern in quadruple-precision format on the other; *u* means *don't care*, i.e., 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)	$(-1)^{s} \times 2^{-16382} \times 0.f$ (subnormal numbers)
e = 0, f = 0	$(-1)^{s} \times 0.0$ (signed zero)
s = 0, e = 32767, f = 0	+INF (positive infinity)
s = 1, e= 32767; f = 0	-INF (negative infinity)
s = *u*, e = 32767, f = 1*uuu...uu*	QNaN (quiet NaN)
s = *u*, e = 32767, f = 0*uuu...uu* (at least one *u* is non-zero)	SNaN (signaling NaN)

Examples of important bit patterns in the double extended storage format appear in Table 2-7.

- The maximum positive normal number is the largest finite number representable in this IEEE double extended format.
- The minimum positive normal number is the smallest positive number which is representable in this IEEE double extended format and for which it is certain that no precision has been lost.
- The maximum subnormal number is the largest number which is representable in this IEEE double extended format and for which it is possible that some precision has been lost.
- The minimum positive subnormal number is the smallest positive number which is representable in this IEEE double extended format and for which it is possible that some precision has been lost.

Table 2-7 Some Bit Patterns in Double Extended Format (SPARC architecture)

Common Name	**Bit Pattern (SPARC)**	**Quadruple-Precision 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`	1.18973149535723176508575932662800 70e+4932
min normal	`00010000 00000000 00000000 00000000`	3.36210314311209350626267781732175 26e-4932
max subnormal	`0000ffff ffffffff ffffffff ffffffff`	3.36210314311209350626267781732175 20e-4932
min pos subnormal	`00000000 00000000 00000000 00000001`	6.47517511943802511092443895822764 66e-4966
+∞	`7fff0000 00000000 00000000 00000000`	+∞
–∞	`ffff0000 00000000 00000000 00000000`	–∞
quiet `NaN`	`7fffffff ffffffff ffffffff ffffffff`	QNaN
signaling `NaN`	`7fff0000 00000000 00000000 00000001`	SNaN

The hex values of the `NaN`s shown in Table 2-7 are just two of the many bit patterns that can be used to represent `NaN`s.

Double Extended Format (x86 Architecture)

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 x86 family of Intel 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 x86 family of Intel 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`.

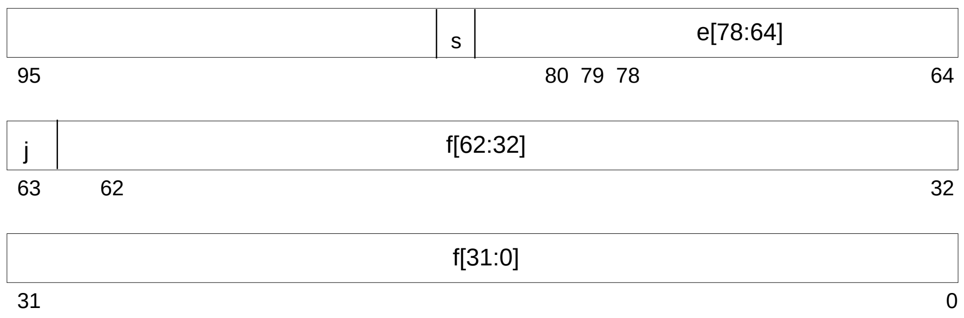

Figure 2-4 Double Extended Format (x86)

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 fields, on the one hand, and the value represented by the bit pattern on the other; *u* means *don't care*, i.e., 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 Architecture)

Double Extended Bit Pattern (x86)	**Value**
j = 0, 0 < e <32767	Unsupported
j = 1, 0 < e < 32767	$(-1)^s$ x $2^{e-16383}$ x 1.f (normal numbers)
j = 0, e = 0; f ≠ 0 (at least one bit in f is nonzero)	$(-1)^s$ x 2^{-16382} x 0.f (subnormal numbers)
j = 1, e = 0	$(-1)^s$ x 2^{-16382} x 1.f (pseudo-denormal numbers)
j = 0, e = 0, f = 0 (all bits in f are zero)	$(-1)^s$ x 0.0 (signed zero)
j = 1; s = 0; e = 32767; f = .000 — 00 (all bits in f are zero)	+INF (positive infinity)

Table 2-8 Values Represented by Bit Patterns (x86 Architecture)

Double Extended Bit Pattern (x86)	**Value**
j = 1; s = 1; e = 32767; f = .000 — 00 (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 will provoke 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.

For the double extended format, the difference between a normal number and a subnormal number is that the explicit leading bit of the significand of a normal number is 1, whereas the explicit leading bit of the significand of a subnormal number is 0 and the biased exponent field e must also be 0. Subnormal numbers in double extended format were called double extended format denormalized numbers in IEEE Standard 754.

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 this IEEE double extended format.

- The minimum positive normal number is the smallest positive number which is representable in this IEEE double extended format and for which it is certain that no precision has been lost.
- The maximum subnormal number is the largest number which is representable in this IEEE double extended format and for which it is possible that some precision has been lost.
- The minimum positive subnormal number is the smallest positive number which is representable in this IEEE double extended format and for which it is possible that some precision has been lost.

Table 2-9 Some Bit Patterns in Double Extended Format and their Values (x86)

Common Name	Bit Pattern (x86 Architecture)	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`	1.18973149535723176502e+4932
min pos normal	`0001 80000000 00000000`	3.36210314311209350626e-4932
max subnormal	`0000 7fffffff ffffffff`	3.36210314311209350590e-4932
min pos subnormal	`0000 00000000 00000001`	1.82259976594123730126e-4951
+∞	`7fff 80000000 00000000`	+∞
–∞	`ffff 80000000 00000000`	–∞
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 00000000 80000001`	SNaN

NaNs (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 (i.e. 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

In this section we will briefly discuss the notions of range and precision for a given storage format. Then we will state the ranges and precisions corresponding to the IEEE single and double formats, and to the implementations of IEEE double extended format to SPARC and Intel x86 architectures. In explaining the notions of range and precision, we will refer to the IEEE single format.

The IEEE standard specifies that 32 bits should be used to represent a floating point number in single format. Since 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 representation of the largest and smallest positive numbers which can be represented in this particular format?"

We rephrase the question and introduce the notion of range. Our question is therefore:

> "What is the range, in decimal notation, of numbers which 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 which can be represented in IEEE single format (if we restrict ourselves to positive normalized numbers) is as follows:

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

A second question refers to the precision (or as many people refer to it, the accuracy, or the number of significant digits) of the numbers represented in a given format. We will explain these notions 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. Let us 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.

On the other hand, we will obtain 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:

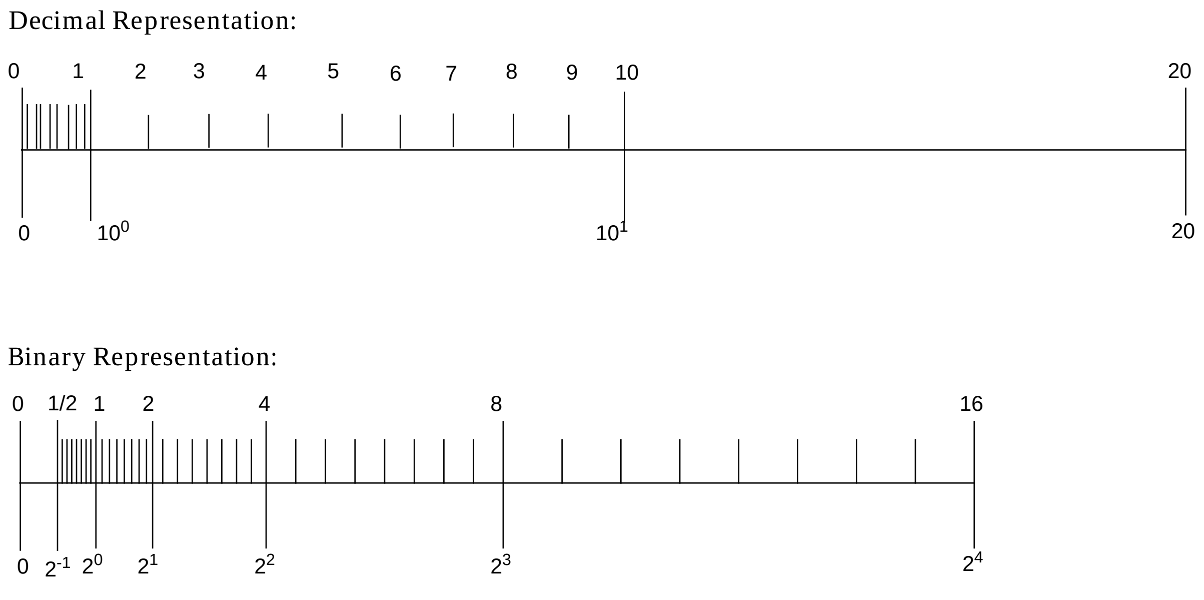

Figure 2-5 Comparison of a Set of Numbers Defined by Digital and Binary Representation

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

We will 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, we are interested in converting from decimal to binary and back to decimal, as well as converting from binary to decimal and back to binary.

It is important to notice that since 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, will result in choosing one of the two neighboring numbers from the second set (which one specifically, is a question related to rounding).

Let us look at some examples first. Assume we are trying to represent the number with the following decimal representation in IEEE single format:

$x = x_1.x_2\ x_3...\ \text{X}\ 10^n$

In the above example, the information contained in *x* has to be coded in a 32-bit word. Generally, this might be impossible (if there are too many digits in *x*), for example, some of the information might not fit in 32 bits. For example, let’s take:

$y = 838861.2,\ z = 1.3$

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 X 10^5 assigned to *y* and the value printed out is 0.000000125, which is seven decimal orders of magnitude smaller than *y*. We say that the accuracy of representing *y* in IEEE single format is about 6 to 7 significant digits, or in other words, 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*. We say that the accuracy of representing *z* in IEEE single format is about 7 to 8 significant digits, or in other words, that *z* has about 7 *significant digits* if it is to be represented in IEEE single format.

We can now formulate the question:

> Assume we 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*?

We rephrase the question:

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

One can prove that 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 the cases when the conversions are exact, when one could think of infinitely many digits being accurate).

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

The complete picture is given in Table 2-10:

Table 2-10 Range and Precision of Storage Formats

Format	Signif. Digit Binary Repr.	Smallest Positive Normal Number	Largest Positive Number	Signif. Digits Decimal Repr.
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

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. Since 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}, since 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 non-zero 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 us 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:

```
computed result = (true result) ±roundoff
```

In IEEE arithmetic, with rounding mode to nearest,

$$0 \leq \texttt{roundoff} \leq 1/2\ ulp$$

of the computed result.

ulp stands for unit in the last place. The least significant bit of the fraction of a number in its standard representation, is the *last* place. If the roundoff error is less than or equal to one half unit in the last place, then the calculation is correctly rounded.

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 never widens but narrows. Conversely, as the magnitude of numbers gets larger, the gap between neighboring representable numbers widens.

For example, imagine we 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.

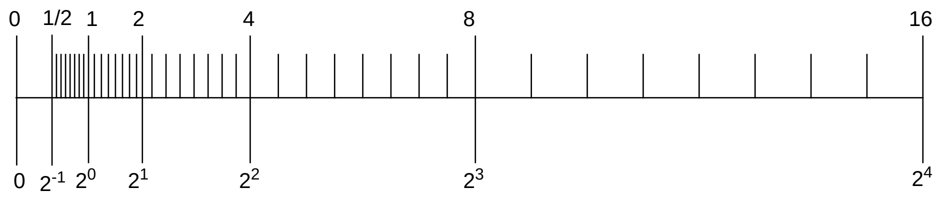

Figure 2-6 Number Line

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 2-12, `nextafter(x,+∞)` denotes the next representable number after `x` as you move along the number line towards `+∞`.

Table 2-12 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.

Since 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.

$$\text{gradual underflow: } |\text{error}| < \frac{1}{2}\ ulp \text{ in } \lambda$$
$$\text{Store 0: } |\text{error}| \approx \lambda$$

There is a significant difference between $\frac{1}{2}$ 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 an inner product.

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

With gradual underflow, `result` will be 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 will scale their calculations if they are able somehow to anticipate where minuteness might degrade accuracy.

The second example, deriving a complex quotient, isn't amenable to scaling.

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

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

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 may 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 things to do for each numerical program.

The Math Libraries 3

This chapter describes the implementation of the mathematical functions that comprise the math libraries: `libm.a`, `libm.so`, and `libsunmath.a`. Some attention will be given to IEEE supporting functions and to functions that convert data between IEEE and non-IEEE formats.

Some information can also be obtained from the manual page `Intro(3)`.

This chapter has the following organization:

Math Library

The operating system math library contains the functions required by the various standards which the operating system conforms to. The libraries `libm.a` and `libm.so` are part of the operating system on Solaris 2.x; `libm.a` is the static version and `libm.so` is the shared version.

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

	SunOS 4.x	SunOS 5.2 and later

Standard location for libm	`/usr/lang/SC3.0.1/lib/`
	`libm.a`
		`/usr/lib/`
		`libm.a`
		`libm.so`

Header Files	`/usr/lang/SC3.0.1/include/cc_4.1.3/`
	`floatingpoint.h`
	`math.h`
	`sys/ieeefp.h`
		`/usr/include/`
		`float.h`
		`floatingpoint.h`
		`ieeefp.h`
		`math.h`
		`values.h`
		`sys/ieeefp.h`
		`sys/machsig.h`

Table 3-1 lists the functions in libm with the names used for calling them from a C program

Table 3-1 Contents of `libm` (`libm.a` and `libm.so`)

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	`bessel(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`

Table 3-1 Contents of `libm` (`libm.a` and `libm.so`)

Type	Function Name
IEEE classification function	`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`.

Value-added Math Library

The library `libsunmath` is part of the libraries supplied with all DevPro compilers.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:

	SunOS 4.x	**SunOS 5.2 and later**
Standard location for libsunmath	`/usr/lang/SC3.0.1/lib/` `libm.il` `libsunmath.a`	`/opt/SUNWspro/SC3.0.1/lib/` `libm.il` `libm_mt.a` `libmopt.a` `libsunmath.a` `libsunmath_mt.a`
Header Files	`/usr/lang/SC3.0.1/include/cc_4.1.3/` `sunmath.h`	`/opt/SUNWspro/SC3.0.1/include/cc/` `sunmath.h` `/opt/SUNWspro/SC3.0.1/include/f77/` `f77_floatingpoint.h`

Table 3-2 lists the functions in `libsunmath`; the names are those used for calling the double precision version of functions from a C program

Table 3-2 Contents of `libsunmath`

Type	Function Name
Functions from Table 3-1	single, extended and quadruple precision available, except for `matherr`
Elementary transcendental functions	`exp2, exp10,` `log2,` `sincos`
Trigonometric functions (degree arguments / values)	`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`, `finite` (in 4.x; in 5.x it is in `libc`), `isinf`, `isnormal`, `issubnormal`, `iszero`
Functions that supply useful IEEE values	`min_subnormal, max_subnormal,` `min_normal, max_normal,` `infinity, signaling_nan, quiet_nan`
Random number generators	`d_addran_, d_addrans_, d_lcran_,` `d_lcrans_, d_shufrans_, i_addran_,` `i_addrans_,` `i_lcran_, i_lcrans_, i_shufrans_,` `r_addran_, r_addrans_, r_lcran_, r_lcrans_,` `r_shufrans_, u_addrans, u_lcrans_,` `u_shufrans_`
Data conversion	`convert_external`

Table 3-2 Contents of `libsunmath` (Continued)

Type	Function Name
Control rounding mode and floating-point exception flags	`ieee_flags`
Floating-point trap handling	`ieee_handler, sigfpe`
Show status	`ieee_retrospective`
Toggle hardware between standard and nonstandard modes (advisory)	`standard_arithmetic, nonstandard_arithmetic`

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 3-3.

Table 3-3 Calling Single, Double, and Quadruple `libm` Functions

Language	Single	Double	Quadruple

C, C++	`#include <sunmath.h>`
	`float x,y,z;`
	`x = sinf(y);`
	`x = fmodf(y,z);`
	`x = max_normalf();`
	`x = r_addran_();`
		`#include <math.h>`
		`double x,y,z;`
		`x = sin(y);`
		`x = fmod(y,z);`
		
		`#include <sunmath.h>`
		`double x,y,z;`
		`x = max_normal();`
		`x = d_addran_();`
			`#include <sunmath.h>`
			`long double x,y,z;`
			`x = sinl(y);`
			`x = fmodl(y,z);`
			`x = max_normall();`

FORTRAN	`REAL x,y,z`
	`x = sin(y)`
	`x = r_fmod(y,z)`
	`x = r_max_normal()`
	`x = r_addran()`
		`REAL*8 x,y,z`
		`x = sin(y)`
		`x = d_fmod(y,z)`
		`x = d_max_normal()`
		`x = d_addran()`
			`REAL*16 x,y,z`
			`x = sin(y)`
			`x = q_fmod(y,z)`
			`x = q_max_normal()`

In general, 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`. Since 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 may result.

Note – The x86 version of FORTRAN supports real and double precision only; it does not support REAL*16. The x86 version of C, however, supports long double.

IEEE Support Functions

This section describes the IEEE recommended functions, the functions which supply useful values, `ieee_retrospective`, and `standard_arithmetic` and `nonstandard_arithmetic`. Refer to Chapter 4, "Exceptions and Signal Handling," for details on the functions `ieee_handler` and `ieee_flags`.

`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 efficiently as bit mask operations.

Table 3-4 `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`

Table 3-4 `ieee_functions`(3m) (Continued)

Function	Description
`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 3-5 `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
`<stdio.h>`	Header file
`ieee_retrospective(*f)`	

`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 will not generate any inexact exceptions.

Table 3-6 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 3-7 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 – One must declare d_<function>s as double precision and q_<function>s 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 special functions described by the `ieee_values`(3m) man page.

The bit patterns that implement these functions can be seen in `libm.il`. Table 3-8, Table 3-9 and Table 3-11 show how to use the functions described by `ieee_values`(3m).

Table 3-8 IEEE Values: Single Precision

IEEE value	**Decimal value and IEEE 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 3-9 IEEE Values: Double Precision (SPARC)

IEEE Value	**Decimal Value and IEEE representation**	**C, C++**	**FORTRAN**
max normal	1.7976931348623157e+308 `7fefffff ffffffff`	`x = max_normal();`	`x = d_max_normal()`
min normal	2.2250738585072014e-308 `00100000 00000000`	`x = min_normal();`	`x = d_min_normal()`
max subnormal	2.2250738585072009e-308 `000fffff ffffffff`	`x = max_subnormal();`	`x = d_max_subnormal()`
min subnormal	4.9406564584124654e-324 `00000000 00000001`	`x = min_subnormal();`	`x = d_min_subnormal()`

Table 3-9 IEEE Values: Double Precision (SPARC) (Continued)

IEEE Value	Decimal Value and IEEE representation	C, C++	FORTRAN
∞	Infinity `7ff00000 00000000`	`x = infinity();`	`x = d_infinity()`
Quiet NaN	`NaN` `7fffffff ffffffff`	`x = quiet_nan(0);`	`x = d_quiet_nan(0)`
Signaling NaN	`NaN` `7ff00000 00000001`	`x =signaling_nan(0);`	`x = d_signaling_nan(0)`

Table 3-10 IEEE Values: Double Precision (x86)

IEEE Value	Decimal Value and IEEE representation	C, C++	FORTRAN
max normal	1.7976931348623157e+308 `ffffffff 7fefffff`	`x = max_normal();`	`x = d_max_normal()`
min normal	2.2250738585072014e-308 `00000000 00100000`	`x = min_normal();`	`x = d_min_normal()`
max subnormal	2.2250738585072009e-308 `ffffffff 000fffff`	`x = max_subnormal();`	`x = d_max_subnormal()`
min subnormal	4.9406564584124654e-324 `00000001 00000000`	`x = min_subnormal();`	`x = d_min_subnormal()`
∞	Infinity `00000000 7ff00000`	`x = infinity();`	`x = d_infinity()`
Quiet NaN	`NaN` `ffffffff 7fffffff`	`x = quiet_nan(0);`	`x = d_quiet_nan(0)`
Signaling NaN	`NaN` `00000001 7ff00000`	`x =signaling_nan(0);`	`x = d_signaling_nan(0)`

Table 3-11 IEEE Values: Quadruple Precision (SPARC only)

IEEE value	Decimal value and IEEE representation	C, C++	FORTRAN
max normal	1.18973149535723176508575932662800070e+4932 `7ffeffff ffffffff ffffffff ffffffff`	`q = max_normall();`	`q = q_max_normal()`
min normal	3.36210314311209350626267781732175260e-4932 `00010000 00000000 00000000 00000000`	`q = min_normall();`	`q = q_min_normal()`
max subnormal	3.36210314311209350626267781732175200e-4932 `0000ffff ffffffff ffffffff ffffffff`	`q = max_subnormall();`	`q = q_max_subnormal()`
min subnormal	6.47517511943802511092443895822764655e-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 3-12 IEEE Values: Double Extended (x86 0nly)

IEEE value	Decimal value and IEEE representation (80 bits)	C, C++	FORTRAN
max normal	1.18973149535723176509e+4932 `7ffe ffffffff ffffffff`	`q = max_normall();`	`q = q_max_normal()`
min positive normal	3.36210314311209350626e-4932 `0001 80000000 00000000`	`q = min_normall();`	`q = q_min_normal()`
max subnormal	3.36210314311209350626e-4932 `0000 7fffffff ffffffff`	`q = max_subnormall();`	`q = q_max_subnormal()`
min positive subnormal	6.47517511943802511092443e-4966 `0000 00000000 00000001`	`q = min_subnormall();`	`q = q_min_subnormal()`
∞	Infinity `7fff 00000000 00000000`	`q = infinityl();`	`q = q_infinity()`
quiet NaN	NaN `7fff 00000000 00000000`	`q = quiet_nanl(0);`	`q = q_quiet_nan(0);`
signaling NaN	NaN `7fff 00000000 00000001`	`q = signaling_nanl(0);`	`q = q_signaling_nan(0)`

nonstandard_arithmetic()

As discussed in Chapter 2, “IEEE Arithmetic,” IEEE arithmetic handles underflowed exceptions 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.

In order to obtain some information whether this is a case in a specific program, one 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, since 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 IEEE754 style of arithmetic—SuperSPARC is such an implementation.

ieee_retrospective()

The `libsunmath` function `ieee_retrospective` prints to `stderr` information about unrequited exceptions and nonstandard IEEE modes. It determines

- 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 exceptions 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 is raised, then that exception occurred at some point during program execution, and was ignored. If a trap is enabled for an exception, then a signal handler was established for that exception by the user program. The `ieee_retrospective` message is meant to alert the user about exceptions that may need to be investigated (if the exception is *raised*), or to remind one that exceptions may have been handled by a signal handler (if the exception is *trapped*.) After an exception traps to a signal handler, the exception is no longer raised. Chapter 4, “Exceptions and Signal Handling,” discusses exceptions, signals, and traps, and Chapter 5, “Compile-Debug-Run,” shows how to investigate the cause of a raised exception.

`ieee_retrospective` can be called anytime, but it is usually called before exit points. FORTRAN programs call `ieee_retrospective` on exit by default.

The syntax for calling this function is:

C, C++	`ieee_retrospective_();`
FORTRAN	`call ieee_retrospective()`

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

```
 Note: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Inexact; Underflow;
 Note: Rounding direction toward zero; see ieee_flags(3M).
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Overflow;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

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 software implementations of `exp`, `log`, `atan`, `sin`, `cos` and `tan` use table-driven algorithms. This yields functions which are accurate and fast. The trade-off is size. The random number facilities and the base conversion routines also use table-driven algorithms.

The `libm` in the SPARCompilers products for the Solaris 2.0 and 1.*x* release contains table-driven implementations of `exp`, `log`, `atan`, `sin`, `cos` and `tan`.

Table 3-13 shows the accuracy for some of the functions in libm. Accuracy is measured with BeEF, the Berkeley Elementary Function test programs, written by Z. Alex Liu. This program reports the worst observed error, measured in `ulps` (`ulp` stands for unit-in-the-last-place).

Table 3-13 Double Precision Table-Driven Elementary Functions: Accuracy

		ulp error	
Function	**Range**	**f77 1.4**	**f77 2.0**
`exp`	[-665.42, 665.42)	-0.87 ulp	-0.85 ulp
`log`	$[2^{-16.5}, 2^{16.5})$	+0.81 ulp	+0.81 ulp
`atan`	$(-\infty, \infty)$	-0.59 ulp	-0.64 ulp
sin	$[0, \pi/2)$	-0.64 ulp	-0.60 ulp
cos	$[0, \pi/2)$	+0.63 ulp	+0.63 ulp

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 $\pi/2$.

Since 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 may 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 a 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, since 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`, `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 two facilities for generating uniform pseudo-random numbers, `addrans`(3m) and `lcrans`(3m). `addrans` is an additive random number generator, and `lcrans` is a linear congruential random number generator.

In addition, `shufrans`(3m) shuffles a set of pseudo-random numbers to provide even more randomness for applications that need it.

The functions in `addrans`(3m) are generally more efficient but their theory is not as refined as those in `lcrans`(3m). Refer to the article "Random Number Generators: Good Ones Are Hard To Find", by S. Park and K. Miller, *Communications of the ACM*, October 1988, for a discussion of the theoretical properties of these algorithms. Additive random number generators are discussed in Volume 2 of Knuth's *The Art of Computer Programming.*

The random number generators are available in four versions, for the four data types: signed integer, unsigned integer, single precision floating-point, and double precision floating-point. These functions can be used in two ways: to generate random numbers one at a time (one per function call), or to generate arrays of random numbers (one array per function call.)

When using `addrans` and `lcrans` to provide variates one at a time, they are constrained to the intervals shown in Table 3-14.

Table 3-14 Interval for Single-Value Random Number Generators

Type	**Lower Bound**	**Upper Bound**
	Value	**Value**
Signed integer	–2147483648	2147483647
Unsigned integer	0	4294967295
Single precision floating point	0.0	0.9999999403953552246
Double precision floating point	0.0	0.9999999999999998890

That is, these are the upper and lower bounds over which the data is uniformly distributed. These bounds cannot be altered.

However, if `addrans` or `lcrans` are used to generate arrays of data, the upper and lower bounds of the interval over which the data is uniformly distributed can be controlled.

Appendix A, “Examples,” shows an example that uses `addrans` to generate an array of 1000 random numbers, uniformly distributed over the interval [0, 3.7e-9].

`lib`*C Support Functions*

This section describes the base conversion feature for `lib`C support.

Base Conversion

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

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

The algorithms for base conversion are now table driven as well. These algorithms yield correctly-rounded base conversion routines. In addition to improved accuracy, the table-driven algorithms reduce the worst-case times for correctly-rounded base conversion.

Base conversion is not too difficult for integer formats but gets more complicated when floating point is involved; as with other floating-point operations, rounding errors occur.

For instance, on conversion from internal floating point to an external decimal representation, the mathematical problem to be solved in default rounding mode is:

```
In the case of printf %f or FORTRAN F format with n digits after
the decimal point:
Given integers i and e, find an integer J such that
          | i × 2**e × 10**n – J | ≤ 0.5

In the case of printf %e or FORTRAN E format with n significant
digits:
Given integers i and e, find integers J and E such that
          | i × 2**e – J × 10**E | ≤ 0.5 × 10**E
and
          10**(n–1) ≤ J ≤ 10**n – 1

In both cases, distinguish exact conversions, and
round halfway cases to nearest even.
```

Correct rounding is required by the IEEE Standard 754. The standard requires correct rounding for typical numbers whose magnitudes range from 10^{-44} to 10^{+44}, but permits slightly incorrect rounding for larger exponents—our table-driven algorithm for rounding is accurate through the entire range. (See section 5.6 of IEEE Standard 754.)

See Appendix E, “References,” for references on base conversion. Particularly good references are Coonen’s thesis and Sterbenz’ book.

Exceptions and Signal Handling 4

This chapter describes IEEE floating-point exceptions, SunOS operating system signals, and how to handle signals. It describes:

The SunOS operating system adopts the IEEE 754 philosophy towards floating-point exception handling. One objective of the IEEE 754 and 854 standards is referred to in the IEEE 854 Standard:

> ... 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.

(See the IEEE 854 standard, p. 18.) As recommended by the standards, software is available so that the user may override default results, and do something special in response to an exception. This is done by establishing a signal handler.

4

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.)

Attempting to take the square root of a negative number is, or causes, an exception, typically thought of as an exception caused by an invalid operation. In such a case several things happen:

- First, the fact that an exception occurred is noted, usually by hardware.
- Either a result is returned.
- Or, if a signal handler is defined for the invalid operation exception, it generates a signal (e.g. `SIGFPE`). Control is then transferred to the specified signal handler.

There are only five types of IEEE floating-point exceptions: `invalid`, `division by zero`, `overflow`, `underflow` and `inexact`. The first three (`invalid`, `division`, `overflow`) sometimes indicate a program error. To facilitate debugging, these exceptions are called `common exceptions`, and `ieee_handler`(3m) gives an easy way to trap on `common` exceptions only. The other two exceptions (`underflow`, `inexact`) are seen very often, and they usually do not indicate program error. In fact, many more floating-point operations than not cause the `inexact` exception.

Table 4-1 condenses information found in IEEE Standard 754-1985. It defines the five floating-point exceptions, and the response of an IEEE arithmetic environment when the exceptions are raised.

Table 4-1 IEEE Floating-Point Exceptions

IEEE Exception	**Reason Why This Arises**	**Example**	**Default Result in the Absence of Signal Handler for that Exception**

Invalid operation	An operand is invalid for the operation about to be performed.
	
	For x86 this exception can also occur because of a stack fault.
		$\sqrt{(-x)}$ for x > 0
		fp_op(signaling_NaN)
		`0` $\times \infty$
		`0/0`
		∞ / ∞
		`x REM 0`
		Unordered comparison (see note (1))
		Invalid conversions (see note (2))
			Quiet `NaN`

Division by zero	Divisor is zero, and dividend is a finite non zero number, or, more generally, when an exact infinite result is delivered by an operation on finite operands.	`x/0` for `x` $\neq$ 0, ∞ `or NaN`
		`log(0)`
			Correctly signed infinity

Overflow	Correctly rounded result is larger in magnitude than the largest number in the destination precision (that is, the range of the exponent is exceeded.)	Double precision: MAXDOUBLE + 1.0e294 exp(709.8) Single precision: (float)MAXDOUBLE MAXFLOAT + 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

Table 4-1 IEEE Floating-Point Exceptions (Continued)

IEEE Exception	**Reason Why This Arises**	**Example**	**Default Result in the Absence of Signal Handler for that Exception**

Underflow	Tininess occurs whenever a nonzero result computed, as though the exponent range and the precision were unbounded, would lie strictly between $\pm 2^{E\_min}$ (see note (3))	Double precision:
		nextafter(min_normal,-∞)
		nextafter(min_subnormal,-∞)
		MINDOUBLE ⁄ 3.0
		exp(-708.5)
		
		Single precision:
		(float)MINDOUBLE
		nextafterf(MINFLOAT, -∞)
		expf(-87.4)
			Subnormal or zero

Inexact	Rounded result of a valid operation is different from the infinitely precise result.	2.0 / 3.0 (float)1.12345678 log(1.1) MAXDOUBLE + MAXDOUBLE, when no overflow trap	The result of the operation (rounded, overflowed, or underflowed, as the case may be).
Denormalized Operand (x86 only)	An arithmetic instruction attempted to operate on a denormal operand	1.0 + MINDOUBLE / 2.0	The result of the operation (rounded, overflowed, or underflowed, as the case may be).

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. The table below shows which predicates cause the exception `invalid` when the relation is unordered.

Unordered Comparisons

Predicate			
math	**c, c++**	**F77**	*invalid* **Exception if unordered**
=	==	.EQ.	no
≠	!=	.NE.	no
>	>	.GT.	yes
≥	>=	.GE.	yes
<	<	.LT.	yes
≤	<=	.LE.	yes

2. Invalid conversion: Attempt to convert `NaN` or `infinity` to integer, or integer overflow on conversion from floating-point format.
3. `E_min`: the minimum exponent is –126, –1022 and –16382, for IEEE single, double, and extended precisions. See Chapter 2, "IEEE Arithmetic" for a description of the IEEE floating-point formats.

The exceptions are prioritized in decreasing order according to the following list:

	x86:	**SPARC**:
a.	`invalid`	`invalid`
b.	`overflow`	`overflow`
c.	`division`	`division`
d.	`underflow`	`underflow`
e.	`inexact`	`inexact`
f.	denormal	

The only exceptions that can occur simultaneously are `overflow` with `inexact`, and `underflow` with `inexact` (and `denormal` on x86). If trapping `overflow`, `underflow`, `inexact` is enabled, the `overflow` and `underflow` traps take precedence over the `inexact` trap; they all take precedence over `denormal` on x86.

One could argue that correctly signed infinity is a reasonable and justified result for x / 0 when x ≠ 0. Division by zero is signaled as exceptional simply for historical continuity with older systems, and also because most programmers don't divide by zero on purpose. The name of the exception, "division", obscures the fact that this exception is raised on other occasions as well, namely, when an infinite result is the precise answer to an operation on finite operands.

SPARC

Each exception has two flags associated with it, current and accrued. The current exception flags are always up-to-date with the last floating-point operation executed. Each is set, or cleared, with every nontrapping floating-point operation. Both the current and accrued exception flags are contained in the FSR (Floating-point Status Register). The accrued flag is set at the first occurrence of the exception and persists for the duration of the user process, or until it is explicitly cleared by the user process.

x86

The SW (Status Word) provides flags for accrued exceptions as well as flags for the status of the floating-point stack.

Exceptions and Signals

`SIGFPE` is the SunOS operating system name for the signal that the operating system delivers to the user process when a floating-point exception occurs. `SIGFPE` is not generated by default in the SunOS operating system, but only when both of the following conditions hold:

- One or more floating-point exceptions occur during the course of a floating-point operation, and
- At least one of the floating-point exceptions has a signal handler associated with it. One can use `ieee_handler` to set up a trap handler so that `SIGFPE` is signaled when an exception occurs.

Here are two examples when `SIGFPE` is essential:

- If one uses `dbx` to find the code that raises an exception, then the `dbx` command `catch FPE` instructs `dbx` to halt the process being debugged when any `SIGFPE` is signaled.
- If `SIGFPE` is signaled but not caught, the process aborts and dumps core.

Table 4-2 lists a few examples of exception handling tasks and the corresponding function calls.

Table 4-2 Examples of Exception Handling Tasks

Function to Execute	**C, C++**	**FORTRAN**
Trap common exceptions only	i = ieee_handler ("set", "common", hdl);	i = ieee_handler (’set’, ’common’, hdl)
Dump core on division by zero	i = ieee_handler ("set", "division", SIGFPE_ABORT);	i = ieee_handler (’set’,’division’, SIGFPE_ABORT)
Suppress warning in F77 or Pascal that exceptions were raised but not cleared		i = ieee_flags (’clear’, ’exception’,’all’) *[prior to exit]*
Add warning to C programs that exceptions were raised but not cleared	ieee_retrospective_(); [*prior to exit*]	[ieee_retrospective *is called automatically in* F77]
Treat subnormal operands and results as zero if faster	nonstandard_arithmetic();	call nonstandard_arithmetic

The SunOS operating system together with `libsunmath` incorporates the IEEE exception model outlined in Table 4-1. When `SIGFPE` is signaled, additional information that describes its type is provided by the system. A user-defined signal handler is passed several parameters, including the signal number and the code number that describes the type. The five types of `SIGFPE` have the same signal number, `8`, and are differentiated by the code number. On Solaris 2.x, `<sys/signal.h>` defines the signal numbers; `<sys/machsig.h>` defines the codes assigned to floating-point exceptions. On Solaris 1.x, `<sys/signal.h>` defines both.

Table 4-3 describes the symbols the SunOS operating system defines for the various types of SIGFPE.

Table 4-3 Types for Arithmetic Exceptions

Solaris 1.x SIGFPE Type	Solaris 2.x SIGFPE Type	IEEE Type
FPE_INTDIV_TRAP	FPE_INTDIV	
FPE_INTOVF_TRAP	FPE_INTOVF	
FPE_FLTINEX_TRAP	FPE_FLTRES	inexact
FPE_FLTDIV_TRAP	FPE_FLTDIV	division
FPE_FLTUND_TRAP	FPE_FLTUND	underflow
FPE_FLTOPERR_TRAP	FPE_FLTINV	invalid
FPE_FLTOVF_TRAP	FPE_FLTOVF	overflow

Some integer arithmetic exceptions have corresponding signals in the SunOS operating system. Integer exceptions are not floating-point exceptions, but if they happen to raise SIGFPE, then sigfpe(3) may be used to establish signal handlers for them.

The term *default result* refers to the value that is delivered as the result of a floating-point operation that causes an exception. IEEE Standard 754-1985 defines default results for operations that cause exceptions. The intent of the Standard, in specifying these defaults, is to allow execution of the program to continue with some sensible result, but also with a definite indication that an exception occurred.

In fact, the IEEE Standard defines two sets of default results: those returned when the exception *is not* trapped by a signal handler (the usual case), and those returned when the exception *is* trapped by a signal handler. A trapped exception is one whose trap enable mask is set (ieee_handler sets the trap enable mask as part of establishing a user-specified signal handler). UNIX signal handlers have no provision for returning the IEEE default results in the trapped case.

Table 4-1 shows the default results for untrapped floating-point exceptions. For the overflow, underflow, and inexact exceptions, the rounding mode affects the default results.

Recall that floating-point exceptions must raise `SIGFPE` for the signal handler to come into play. Untrapped floating-point exceptions do not raise `SIGFPE`, although they do set the current and accrued exception bits in the hardware status register. Trapped floating-point exceptions always raise `SIGFPE`. The SunOS operating system (Solaris 2.x or Solaris 1.1 or later) leaves the destination register unchanged, as described in the *SPARC Architecture Manual*, Version 8, Appendix N.

This means that the register that would have been written with the result of the floating point computation is left with whatever value it held prior to the exception. Note that for this reason, it is not a good idea to rely on the results that happen to be returned by a particular hardware implementation. `-fnonstd` and `-fast` set the trap enable mask for the common exceptions (which, if raised at runtime, and signaled but not caught, lead to premature program termination).

For emphasis, we repeat the cautionary note regarding default results in the presence of signal handlers: Although specified by the standard, there is no way to deliver standard default results to UNIX signal handlers. When a signal handler does nothing but return and continue execution at the next instruction, some value will be in the destination of the instruction that trapped, namely, the previous contents of that destination.

It is possible to substitute the default results with user specified values when floating-point instructions lead to exceptions. For example, one might prefer to substitute the largest double precision number for infinity as the result of a floating-point operation that divides by zero. Substituting the default results can be done using the following steps:

1. Establish a signal handler that traps on the exceptions whose default results are to be changed.

2. Write the signal handler to determine the address of the instruction causing the exception, examine the instruction, determine the destination register, and finally write the desired value to the destination register.

Since it is not known at compile time which floating-point instructions will lead to exceptions, there is no way to avoid this runtime decoding of the instruction and finding of the destination register. The only way to gain access to this runtime information is through a signal handler. Note that using a signal handler to deliver a user-supplied result for an exceptional operation carries a big performance penalty.

libsunmath *Support For IEEE Modes & Exceptions*

Several libsunmath functions provide interfaces to help programmers work with floating point exceptions and rounding modes. Table 4-4 summarizes the tasks that can be accomplished using these interfaces.

Table 4-4 Software Support for IEEE Exceptions

Action	man **page reference**
set rounding direction	ieee_flags (3m)
get rounding direction	ieee_flags (3m)
get accrued exception flag	ieee_flags (3m)
clear exception flags	ieee_flags (3m)
restore default IEEE modes: clear all exception flags and reset rounding direction to round to nearest	ieee_flags (3m)
set exception handler	ieee_handler (3m)
get exception handler	ieee_handler (3m)
clear exception handler	ieee_handler (3m)
get ieee values (e.g. ∞, NaN)	ieee_values (3m)
print all accrued exceptions	ieee_retrospective (3m)

ieee_handler *(3m)*

ieee_handler is used primarily to establish a signal handler for a particular floating-point exception, or group of exceptions. ieee_handler can also be used to decommission floating-point signal handlers, and to determine whether or not a signal handler is established for a given exception.

It is recommended that one use ieee_handler (3m) in lieu of the lower-level functions sigfpe(3) and signal(3f). For an exception to raise SIGFPE, its trap enable mask must be set. ieee_handler(3m) sets the relevant bits in the trap enable mask in the hardware floating point status register, in addition to establishing the user-defined signal handler as the procedure to which control is transferred in the event of a signal, while signal and sigfpe(3m) do not set the trap enable mask.

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 a function whose type is `sigfpe_handler_type`, as defined in `<floatingpoint.h>` (`<f77_floatingpoint.h>` for FORTRAN programs). The value returned to `i` indicates whether or not the system succeeded in carrying out *action*. A value of `0` indicates *action* succeeded; any other value indicates *action* failed.

The three input parameters may take the 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

It is a good programming practice always to check that a call to `ieee_handler` succeeds (by checking that `0` is returned to `i`).

If a program calls `ieee_handler` with *handler* specified as `SIGFPE_DEFAULT` or `SIGFPE_IGNORE`, then the hardware trap enable bits are not set. If *handler* is specified as `SIGFPE_ABORT`, then the user process will abort and possibly create a core file when the specified exception occurs.

Appendix A, "Examples" contains examples of complete programs that establish signal handlers from C or C++, and FORTRAN. We will give here a few code fragments illustrating the usage of `ieee_handler.`

This fragment of C code demonstrates how to abort on division by zero:

```
#include <sunmath.h>
if (ieee_handler("set", "division", SIGFPE_ABORT) != 0)
   printf("ieee trapping not supported here \n");
```

~~Solaris 2.x~~

The following example sets up a signal handler named `continue_hdl` that traps on the common exceptions (invalid, division, and overflow), and prints a warning message. The value of the destination register is unchanged:

```
#include <sunmath.h>
#include <stdio.h>
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>

extern void continue_hdl(int sig,siginfo_t *sip,ucontext_t *uap);

main(){
     ...

if (ieee_handler("set", "common", continue_hdl) != 0)
     printf("ieee trapping not supported here \n");
     ...
}

void continue_hdl(int sig,siginfo_t *sip,ucontext_t *uap);
{
     printf("ieee exception code %d occurred at address %X\n",
     sip->si_code, sip->_data._fault._addr);
}
```

Solaris 2.x

And here is a similar program fragment in FORTRAN:

```
program your_program
external continue_hdl
     ...
i = ieee_handler('set', 'common', continue_hdl)
if (i.ne.0) print *, 'ieee trapping not supported here'
     ...
end

integer function continue_hdl(sig, sip, uap)
integer sig
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
print *, 'ieee exception code ', sip.si_code,
         ' at address ', sip.fault.address
end
```

Reset default IEEE exception handling for all exceptions:

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

Similar action in FORTRAN:

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

Abort on division by zero:

```
#include <f77_floatingpoint.h>
  i = ieee_handler('set', 'division', SIGFPE_ABORT)
  if(i.ne.0) print *,'ieee trapping not supported here'
```

Remember that FORTRAN programs that use the C preprocessor to include header files should use the suffix `.F` (instead of `.f`).

`ieee_flags`*(3m)*

`ieee_flags` (3m) is the recommended 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 which are the possible values for the parameters are shown in Table 4-5:

Table 4-5 Parameter Values for `ieee_flags`

Parameter	**C or C++ Type**	**Possible Value**
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.

We will shortly discuss some of the arithmetic features that can be modified by using `ieee_flags`.

The possible rounding directions are: round towards nearest, round towards zero, round towards +∞, or round towards –∞. The IEEE default rounding direction is *round towards nearest*. This means that when the mathematical result of an operation lies halfway between two representable numbers, the one nearest to the mathematical result will be delivered as the result. If the mathematical result lies halfway between the two representable numbers, then *round to nearest* mode specifies *round to the nearest even* number (final bit is zero.)

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:

Parameter	**Possible value**
action	get, set, clear, clearall
mode	direction
in	nearest, tozero, negative, positive
out	nearest, tozero, negative, positive

Rounding precision can not be set on SPARC-based systems. Calls to `ieee_flags` regarding rounding precision (i.e. setting mode = precision) have no effect on computation.

Another aspect of IEEE arithmetic that can be affected from user programs is the status of the accrued exception flags. There is an accrued flag associated with each of the five IEEE floating-point exceptions.

An accrued exception flag is set when the corresponding bit in the hardware floating-point status register is set to 1. It is cleared when that bit is (re)set to 0. Accrued exception bits can be examined, set or cleared. Generally, a user program *examines* or *clears* the accrued exception bits.

If an exception is raised at any time during program execution, then its flag is set, and remains set, unless it is explicitly cleared. In FORTRAN, clearing accrued exceptions is done by a call as shown:

```
      call ieee_flags('clear', 'exception', 'overflow', out)
```

For C or C++, querying whether an exception has been raised can be done as follows:

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

the string returned in `out` is:

- `not available` – if information on exceptions is not available
- `"" (the empty string)` – if there are no accrued exceptions or, in the case of x86, the denormal operand is the only accrued exception
- if the value of the third argument, `in`, is the string that names one of the five IEEE exceptions, then the name of that exception is returned in out if the exception flag is set, and the name of the exception with the highest priority is returned otherwise; if `in` is set to any other string, then the accrued exception with the highest priority is returned. For example there is no special meaning attached to "`all`" in the following call:

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

As another example, to examine whether the `division-by-zero` accrued exception flag is set one can make the following FORTRAN call:

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

The parameter `out` is set to '`division`', if the flag is set. Otherwise, the highest priority exception is returned in `out`.

SPARC

To determine which exceptions are set, decode the binary representation of the integer value `i` returned by `ieee_flags`. The least significant five bits describe the state of all five accrued exception flags (the low-order bit is bit 0). The file `<sys/ieeefp.h>` describes the machine-dependent encodings.

Table 4-6 Exception Bits (SPARC)

Machine	**bit position in integer** `i`				
	4	**3**	**2**	**1**	**0**
SPARC	invalid	overflow	underflow	division	inexact

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

```
/*
 *   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);
```

x86

To determine which exceptions are set, decode the binary representation of the integer value `i` returned by `ieee_flags`. The least significant six bits describe the state of all six accrued exception flags (the low-order bit is bit 0). The file `<sys/ieeefp.h>` describes the machine-dependent encodings.

Table 4-7 Exception Bits (x86)

Machine	**bit position in integer `i`**					
	5	**4**	**3**	**2**	**1**	**0**
x86	inexact	underflow	overflow	division	denormalized	invalid

Compile-Debug-Run 5

This chapter discusses compiler options, code generation options and performance tips. It also summarizes procedures for using the debuggers to examine floating-point data.

Compile

This section discusses code generation options and recommendations on compilation command lines for optimum performance. Also included are some tips about how to best use the compilers. These remarks describe the C and FORTRAN products for the SunOS 4.x and 5.x operating systems, but most of the remarks apply to C++ and Pascal as well. Refer to the compiler man pages (`cc`(1), `f77`(1), `CC`(1), `pc`(1)) for a complete listing of the compilation options available with each compiler. We list here a few items relevant to numerical computations.

- On SPARC, the default code generation option is `-xcg89`.
- On x86, the default code generation option is `-x386`.
- The switch -V causes the compiler to print version information for each of the invoked tools.

Table 5-1 lists compiler flags relevant to numerical computations.

Table 5-1 Compiler Flags

cc flags	**f77 flags**	**action**
-#	-v	verbose mode
-###	-dryrun	show components but do not execute
-B dynamic, -B static, -dn/-dy	-B dynamic, -B static, -dn/-dy	control dynamic/static linking
-dalign	-dalign	generate double load/store instructions
	-f	align common blocks
-fast	-fast	select optimum options for speed
-fnonstd	-fnonstd	put hardware in nonstandard mode (if available)
-fsimple	-fsimple	simple floating-point model
-L<dir>, -l<name>	-L<dir>, -l<name>	link with the specified libraries
R <path> (SunOS 5.x only)	R <path> (SunOS 5.x only)	link with the specified libraries
-lsunmath, -lm		link with libsunmath.a
-misalign	-misalign	allow for misaligned data
-native	-native	best native floating-point
-O, -xO[1,2,3,4]	-O, -xO[1,2,3,4]	control optimization levels
-p, -xpg	-p, -xpg	collect data for profiling
-V	-V	output version information
-x386, -x486, -xpentium	-x386,-x486, -xpentium	code generation for specific platforms (x86 only)
-X[a,c,s,t], -xlibmieee	-ansi	conformance to standards
-xa	-xa	basic block profiling for `tcov`
-xcg89, -xcg92	-xcg89, -xcg92	code generation for specific platforms (SPARC only)

Table 5-1 Compiler Flags

cc flags	f77 flags	action
-xlibmil/ -xnolibmil	-xlibmil/ -xnolibmil	in-line/do not in-line math templates
-xsb	-xsb	collect source browser information
-xunroll=n	-xunroll=n	control unrolling of loops
-Y<c>,<dir>		look for component c in directory dir

Some compilation options have effects that are global; the code generation options, profiling options and data alignment options are among these. Therefore one should use the following flags consistently in the compilation and linking steps:

```
-fast -dalign -misalign -xa -p -xpg
```

If one module is compiled with one of these options, and is linked with some other module not compiled with the same code generation option (or profiling option, or data alignment option), the program may not execute correctly. The man pages for `cc` (1) and `f77` (1) describe compilation options in detail.

It is perfectly fine, and even recommended at times to compile different modules at different `–xO`*n* levels.

Code Generation Options

SPARC

On SPARC, the code generation options are `-xcg89` and `-xcg92`. As described in Appendix B, “SPARC Behavior and Implementation”, a SPARC may have a:

- TI TMS390Z50 SuperSPARC chip
- Fujitsu 86904 microSPARC II chip
- TI TMS390S10 microSPARC chip
- Weitek 8701 POWERuP chip
- TI 602a-based FPU (Floating Point Unit)
- Weitek 8601 or Fujitsu 86903-based FPU
- Weitek 3170, 3171 or 3172

- TI 8847-based FPU
- Weitek 1164/1165-based FPU
- or no floating-point hardware.

One important difference between the earlier Weitek 1164/1165-based FPU and the rest of the SPARC floating-point options is the hardware square root implemented by the newer FPUs.

Caution – In SunOS 4.x, code compiled with any code generation option will not run on machines with the Weitek 1164/1165 FPU.

---

x86

In SunOS 5.x, the code generation options are `-x386,-x486` and `-xpentium`.

SPARC

The utility `fpversion` determines the type of floating-point unit installed, and recommends either the `-cg89` or `-cg92` code generation switch. Refer to Appendix B, "SPARC Behavior and Implementation for a description of `fpversion` (1). You can use earlier code generation options than the one recommended by `fpversion`, but using later ones is discouraged. You should not link together modules compiled with different code generation options.

Optimization Options

In addition to `-O`, `-xO1`, `-xO2`, `-xO3` and `-xO4`, there are several compilation switches related to optimizing: `–fast`, `–xlibmil`/`–xnolibmil`, `–fnonstd`, and `–dalign`. Recall that some of these flags are preceded by an `x` when used with C.

-fast

`–fast` is a macro for several common optimization switches that make it easier to take advantage of the different types of optimizations available.

SPARC

Used with the C compiler, `-fast` expands to:

```
-native -xO2 -dalign -xlibmil -fnonstd -fsingle -fsimple
```

Note – On SunOS 4.x, `-fast` expansion also includes `-xlibmopt`.

Used with **`f77`**, `-fast` translates to:

```
-native -xO3 -dalign -xlibmil -fnonstd -fsimple -xlibmopt
```

x86

Used with the C compiler, `-fast` expands to:

```
-xO2 -xlibmil -nostore -fnonstd -native
```

Used with **`f77`**, `-fast` translates to:

```
-xO2 -xlibmil -nostore -fnonstd -native
```

Some of the options invoked by `–fast` might be inappropriate for some programs. Understanding these options and applying them only when appropriate is better than using `–fast`.

For example, since `–fast` invokes `–fnonstd,` which in turn causes SPARC workstations to run in nonstandard arithmetic mode, programs compiled with `–fast` might carry out computations with less accuracy. Also, on exit, they might display a warning message:

```
Note: nonstandard floating-point arithmetic mode
was enabled and was never cleared; see ieee_functions(3M)
```

See the discussion of `-fnonstd`, below, for more information.

Some of the flags implied by `-fast` can be overridden by subsequent switches.

- The optimization level `–xO2` can be overridden by subsequently specifying an explicit `-O,` or `–xO`*n*. The option `–native` can be overridden by an explicit <*code_generation_option*>.
- `–xlibmil` can be overridden by a subsequent `–xnolibmil`.
- `–xlibopt` can be overridden by a subsequent `–xnolibopt`

No other switches invoked by `–fast` can be overridden by other compilation switches (see Table 5-2). Note, however, that the nonstandard exception handling enabled by `-fnonstd` is overridden if the program establishes a signal handler, or by calling the `libsunmath` function `standard_arithmetic` from the user code.

Table 5-2 Selectively Overriding `-fast`

Examples of Options	**Effect**
-fast -xO4	-xO4 is used instead of the `-xO2` or `-xO3` implied by `-fast`
`-fast -xnolibmil`	inline templates are not used

To override some switch invoked by `–fast`, one should specify the overriding switch after specifying `–fast`. One can use the flags `-#` or `–v` (depending on the compiler) to see explicitly which flags are actually being passed to the compiler.

Note – Do not use `–fast` with programs that depend on IEEE standard exception handling.

`-native`

When `–native` is specified, the compilers choose at compile time the fastest *<code_generation_option>* for the machine where the compilation occurs.

Table 5-3 *<code_generation_option>* for `-native`

Compiling on ...	`-native` **translates to ...**
SPARCserver 6xx with SuperSPARC	-xcg92
SPARCstation 10 models 30, 41, 52, 54	-xcg92
SPARCserver 6xx with TI60Za	-xcg89
SPARCserver 4xx	-xcg89
SPARCstation 2 with TI602a / IPX / ELC	-xcg89
SPARCStation 1 with Weitek 3170 / IPC / SLC	-xcg89
SPARC with TI 8847	-xcg89
SPARC with Weitek 1164/1165	-xcg87
SPARC with no floating-point hardware	-xcg89
386 based machine	-x386
486 based machine	-x486
pentium based machine	-xpentium

`-fsimple`

Simple floating-point model

Allow optimization using mathematically equivalent expressions. The optimizer is allowed to act as if a simple floating-point model holds during compilation and runtime. It is allowed to optimize without regard to roundoff or numerical exceptions.

With `–fsimple`, the optimizer can assume the following:

- The IEEE 754 default rounding and trapping modes hold.
- No exceptions arise other than inexact.
- The program does not test for infinities or NaNs.
- The program does not depend on the sign of zero.

-xlibmil/-xnolibmil, -xlibmieee

When using the switch `–xlibmil` it is not necessary to specify the full path name of the in-line template file on the compile line. The compilers choose the in-line template file that matches the prevailing code generation option. For example, on SPARC `–xlibmil` causes the in-line template file `/opt/SUNWspro/SC3.0/cg89/libm.il` to be used during the inlining pass of the compilation. The switch `–xnolibmil` can be used to override a previously specified `–xlibmil`.

The switch `-xlibmieee` causes the math functions to return values in the spirit of the IEEE standard when exceptions occur. Refer to Appendix B for more details on standards compliance.

-fnonstd

The switch `–fnonstd` causes nonstandard initialization of floating-point hardware. By default, the arithmetic is done according to the IEEE 754 standard. The IEEE 754 floating-point arithmetic model requires that default results be provided in exceptional situations (e.g. overflows, divisions by zero), and that underflows be gradual. Specifying `–fnonstd` during the link step has two effects:

- It causes hardware traps to be enabled for the floating-point exceptions invalid, division by zero, and overflow. If the hardware trap is enabled when an exception occurs, the system signal `SIGFPE` is generated. Unless the program includes a signal handler for the exception that occurred, it will terminate with a memory dump. See Chapter 4, “Exceptions and Signal Handling for more information on `SIGFPE`.
- It causes the floating point unit to operate in abrupt underflow mode if possible. See Chapter 2, “IEEE Arithmetic for the benefits of gradual underflow. Although some programs may run faster when floating-point hardware operates in abrupt underflow mode (when available), some

calculations may suffer a greater loss of accuracy than when they are run with gradual underflow. In the case of x86, abrupt underflow does not occur because the underflow is handled by hardware.

Programs linked with `–fnonstd` display this warning message upon completion:

```
Note: this program was linked with -fast or -fnonstd
and so may have produced nonstandard floating-point results.
```

Recall that nonstandard mode can be set dynamically as well. In fact, a program can switch back and forth between standard and nonstandard modes (if available) by using the functions `standard_arithmetic` and `nonstandard_arithmetic`. See Chapter 3, "The Math Libraries for more information.

-dalign (SPARC only)

With RISC architectures, including the SPARC architecture of the SPARC, proper data alignment can help performance. By default, the compilers do not assume that data is properly aligned. The switch `–dalign` instructs the compilers to assume data alignment. Under that assumption, the compiler will try to perform aggressive optimizations.

With `–dalign`, all double precision data is aligned on double word boundaries, and double word loads and stores are used wherever possible, to allow faster data access. If a program is compiled with `–dalign` and subsequently uses misaligned data as input, there may be unexpected results.

One should keep in mind two things: first, if one module of a program is compiled with `–dalign`, all modules should be compiled with `-dalign`, and second, that the –`dalign` switch is invoked by `–fast`.

Redefining `libm` and `libsunmath` *Functions vs.* `-xlibmil`

The SunOS 5.x operating system allows programmers to redefine any math library entry point. Some entry points call other `libm` or `libsunmath` entry points, however.

To assure that this internal use is not affected by users' redefinitions, the libraries use the weak symbol mechanism provided by the SVR4 link editor to distinguish between internal and external references to `libm` and `libsunmath` functions. Each public math library entry point is in fact a "weak" symbol; associated with it is the actual entry point, an internal symbol. which is the math library symbol preceded with two underscores.

For example the library symbol `exp` is in fact a "weak" symbol; associated with it is the actual entry point `_ _exp`. Any other `libm` or `libsunmath` function which would normally call the function, in fact, calls the actual entry point. In our example the power function `pow` which would in principle call `exp`, calls the internal symbol `_ _exp` instead. That is, redefining `exp`, does not affect `pow` — it still calls the original `_ _exp` entry point.

Note however that if one replaces functions that have in-line templates, one should not compile with `-xlibmil`. The reason is that if an in-line template exists, then compiling with `-xlibmil` (or `-fast`, which invokes `-xlibmil`), causes the function to be inlined during one of the compilation passes, before the link editor resolves external references. By the time the link editor resolves symbols, the reference to the function has already effectively been resolved and there's no longer any hook for the externally defined function to link with.

Functions having in-line template implementations include square root, hypotenuse, IEEE support, IEEE classification and IEEE values.

To verify the exact list of functions that have in-line template counterparts examine the following text files:

- For SunOS 4.x examine `/usr/lang/SC3.0.1/lib/lim.il`
- For SunOS 5.x examine `/opt/SUNWspro/SC3.0.1/lib/libm.il`

For more information about weak symbols and library name space, refer to *C 3.0.1 User's Guide.*

About Precision

Traditional C promotes single precision parameters to double precision, and uses double precision temporaries to evaluate arithmetic expressions that involve single precision variables. This is unlike FORTRAN, where single precision expressions are always evaluated using single precision arithmetic, and single precision parameters are always passed as single precision values. Compiling with `-Xt` or `-Xs` forces traditional expression evaluation to be used.

Since `-Xt` is the default compilation option with the C compiler when no `-X{a,c,t,s}` option is specified, traditional expression evaluation is the default.

ANSI C compilers pass prototyped single precision floating point parameters as single precision, and may use single precision expression evaluation.

Often, the extra precision is welcome because of the increased accuracy, but this might imply lower speed. The C compiler switch `–fsingle` causes the compiler to issue single precision machine instructions for arithmetic expressions on `float`s.

Compiling with `-Xa` or `-Xc` causes arithmetic expressions on `float`s to be evaluated using single precision arithmetic. C++ always defines `-Xa`, so single precision expression evaluation for arithmetic expressions involving floats is the default for **`CC`**.

C and C++ use the ANSI C approach for passing `float` parameters: `float`s are passed and returned as `float`s. Note however that in traditional C, `float` parameters were promoted to double precision, and passed as double precision values. `libsunmath` provides single precision versions of the mathematical functions that expect and return true single precision types. The prototypes in `<sunmath.h>` and `llib-lsunmath.ln` reflect this. However, the `libc` routines `[fs]printf` promote `float` arguments to `double` before performing base conversion.

The FORTRAN compilation switch `–r8` causes variables declared with type `REAL` to act like IEEE 64-bit double precision variables. Declared variables, declared functions, literal constants and intrinsic functions are adjusted. Type declarations that specify the size (for example, `REAL*4` or `REAL*8`) are not affected; implicit declarations are not affected. However, variables already declared `DOUBLE PRECISION` are promoted to quadruple precision. Refer to `f77` (1), the *FORTRAN 3.0.1 Reference Manual*, and the *FORTRAN 3.0.1 User's Guide* for more information.

Debug

This section gives a few hints about how to use `dbx` (source-level debugger) and `adb` (assembly-level debugger) to examine data held in floating-point registers. This will be achieved mostly by looking at an example of using `dbx` and `adb` to investigate the cause of a floating-point exception. Refer to the *Debugging a Program* manual for more information.

Material in this manual about adb is here for the convenience of the reader. It should be noted, though, that adb is not part of the compiler language products, but rather it is a tool bundled with the SunOS operating system.

Recall that in order to use dbx, programs should be compiled with the –g flag. Any executable (compiled with –g or not) can be used with adb.

A Few Useful Debugging Commands

Table 5-4 shows examples of debugging commands for the SPARC architecture

Table 5-4 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	examine &$f0/X	<f0=X
decimal equivalent		<f0=f
IEEE double precision	examine &$f0/2X	<f0=X; <f1=X
decimal equivalent	print $f0	<f0=F
Examine all fp registers	examine &$f0/32X	$x for f0-f15 $X for f16-f31
Examine all registers	examine &$g0/64X	$r; $x; $X
Examine fp status register	examine &$fsr/X	<fsr=X
Put single precision 1.0 in f0		3f800000>f0
Put double prec 1.0 in f0/f1	assign $f0 = 1.0	3ff00000>f0; 0>f1
Continue execution	cont	:c
Single step	step (or next)	:s
Exit the debugger	quit	$q

Table 5-5 shows examples of debugging commands for the x86 architecture

Table 5-5 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 one can just say:

```
myfunction:b
```

Second, one 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
```

SPARC

The main in FORTRAN programs is known as `MAIN_` to `adb`. To set a breakpoint at a FORTRAN main in `adb`:

```
MAIN_:b
```

When examining the contents of floating-point registers, the hex value shown by the `dbx` command `&$f0/X` is the base-16 representation, not the number's decimal representation. For SPARC, the `adb` commands `$x` and `$X` display both the hexadecimal representation, and the decimal value—decimal only for x86. For SPARC, the double precision values show the decimal value next to the odd-numbered register.

With `adb`, one can not write into the floating-point registers until they have been touched by the executing program.

SPARC

When displaying floating point numbers, one 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

Since `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 which was used to display the first sixteen 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]  +4.2016390366939049272332234 e-13         EMPTY
 st[6]  +4.2016390366939049272332234 e-13         EMPTY
 st[7]  +2.7224999213218694649185636              EMPTY
```

x86

Note – cw is the control word; sw is the status word.

Using the Debuggers to Investigate an IEEE Exception

Consider the following program:

```
        program driver

        double precision x,y

        x = -4.2d0
        y = routine(x)
        print * , x, y

        end


        double precision function routine(x)

        double precision x

        foo = sqrt(x) - 1.0d0

        return
        end
```

Compiling and running this program will produce:

```
    -4.2000000000000 NaN
  Note: the following IEEE floating-point arithmetic exceptions
  occurred and were never cleared; see ieee_flags(3M):
  Inexact;  Invalid Operand;
  Note: IEEE NaNs were written to ASCII strings or output files;
 see econvert(3).
  Sun's implementation of IEEE arithmetic is discussed in
  the Numerical Computation Guide.
```

To determine the cause of the "`Invalid Operand`" floating-point exception, one can add a signal handler, recompile with the `-g` option and use the source-level debugger dbx. Alternatively, one can use a combination of adb and dbx at the assembly level with no changes or recompilation of the sources.

Using `dbx` *to Locate the Instruction Causing an Exception*

SPARC

The simplest way to locate the code that causes a floating-point exception is to set up a signal handler, recompile with the `-g` option, and then use `dbx` to track down the faulting instruction. First, add a signal handler to your program. The program becomes:

```
       program find_exception

       double precision x, y
       external routine
       external common_handler

       i = ieee_handler("set", "common", common_handler)
      if (i.ne.0) print *, "Could not establish fp signal handler"
       x = -4.2d0
       y = routine(x)
       print *, x, y

       end


       double precision function routine(x)

       double precision x

       routine = sqrt(x) - 1.0d0

       return
       end
```

SunOS 5.x on SPARC

and

```
      integer function common_handler(sig, sip, uap)

      integer sig
c define the structure siginfo, as in <sys/siginfo.h>
      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 SunOS 5.x codes for ieee signals (see <sys/machsig.h>):
c invalid: 7
c divide: 3
c overflow: 4
c underflow: 5
c inexact: 6

      write(*,10) sip.si_code, sip.fault.address
 10   format("ieee exception ",i4," occurred at address ",z8)

      end
```

SPARC

Next, recompile with the `-g` flag and run the program to determine the address of the instruction that causes the invalid operation exception:

```
ieee exception    7 occurred at address    10F94
    -4.2000000000000 -NaN
 Note: IEEE NaNs were written to ASCII strings or output files;
see econvert(3).
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Overflow;  Division by Zero;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

SPARC

Finally, use dbx to associate the hex address 0x10f94 with the source code:

```
(dbx) stopi at 0x10f94
(2) stopi at 69524
(dbx) run
Running: find_exception2
stopped in routine at 0x10f94
routine_+0x24:  fsqrtd  %f2, %f4
(dbx) where
routine(x = -4.2), line 20 in "find_exception2.f"
MAIN(), line 10 in "find_exception2.f"
(dbx) cont
ieee exception    7 occurred at address    10F94
    -4.2000000000000 -NaN
 Note: IEEE NaNs were written to ASCII strings or output files;
see econvert(3).
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Overflow;  Division by Zero;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.

execution completed, exit code is 0
program exited with 0
```

dbx translates the hex address, 0x10f94, to its decimal equivalent, 69524, when it confirms the breakpoint. The traceback displayed in response to the dbx command where shows the line number in the source code that caused the exception.

Using adb *to Locate the Instruction Causing an Exception*

SPARC

In this case the steps are:

1. Run the program inside adb, and stop before much computation occurs.
2. Set the trap enable masks in the floating-point status register to trap on the exception that concerns you. This is necessary so that SIGFPE is raised.
3. Continue the adb execution; adb will break automatically when SIGFPE is raised. This breakpoint is the address of the faulting instruction.

4. Re-run the program from inside `dbx`, stopping on the faulting instruction, and then request a traceback with `where` after the floating-point instruction, and continue execution.

Here is a copy of the interactive `adb` session. The aim is to find the absolute address of the instruction which raises the exception.

```
demo% adb a.out
MAIN_:b
:r
breakpoint      __cg89_used:    sethi   %hi(0xfffffc00), %g1
<pc,10?ia
__cg89_used:    sethi   %hi(0xfffffc00), %g1
MAIN_+4:        add     %g1, 0x390, %g1          ! -0x70
MAIN_+8:        save    %sp, %g1, %sp
MAIN_+0xc:      sethi   %hi(0x22c00), %l7
MAIN_+0x10:     add     %l7, 0x258, %l7          ! _end + 0xe58
MAIN_+0x14:     sethi   %hi(0x21c00), %o1
MAIN_+0x18:     or      %o1, 0x88, %o1           ! 0x21c88
MAIN_+0x1c:     ld      [%o1], %o0
MAIN_+0x20:     ld      [%o1 + 0x4], %o1
MAIN_+0x24:     st      %o0, [%l7 - 0x1000]
MAIN_+0x28:     st      %o1, [%l7 - 0xffc]
MAIN_+0x2c:     add     %l7, -0x1000, %o0
MAIN_+0x30:     call    routine_
MAIN_+0x34:     nop
MAIN_+0x38:     fstod   %f0, %f0
MAIN_+0x3c:     st      %f0, [%l7 - 0xff8]
MAIN_+0x40:
MAIN_+0x2c:b
:c
breakpoint      MAIN_+0x2c:     add     %l7, -0x1000, %o0
<fsr=X
                40021
08040021>fsr
:c
SIGFPE 8: numerical exception (invalid floating point operation)
stopped at      routine_+0x34:  std     %f8, [%fp - 0x10]
```

```
routine_,14?ia
routine_:       sethi   %hi(0xfffffc00), %g1
routine_+4:     add     %g1, 0x3b0, %g1          ! -0x50
routine_+8:     save    %sp, %g1, %sp
routine_+0xc:   st      %i0, [%fp + 0x44]
routine_+0x10:  ld      [%fp + 0x44], %o1
routine_+0x14:  ld      [%o1], %f2
routine_+0x18:  ld      [%o1 + 0x4], %f3
routine_+0x1c:  fsqrtd  %f2, %f4
routine_+0x20:  sethi   %hi(0x21c00), %o2
routine_+0x24:  or      %o2, 0x98, %o2           ! 0x21c98
routine_+0x28:  ld      [%o2], %f6
routine_+0x2c:  ld      [%o2 + 0x4], %f7
routine_+0x30:  faddd   %f4, %f6, %f8
routine_+0x34:  std     %f8, [%fp - 0x10]
routine_+0x38:  ba      routine_ + 0x40
routine_+0x3c:  nop
routine_+0x40:  ldd     [%fp - 0x10], %f0
routine_+0x44:  ba      routine_ + 0x4c
routine_+0x48:  nop
routine_+0x4c:  ret
routine_+0x50:
routine_+0x1c=X
                10c64
```

The steps are the following: Set break in MAIN. Run; the execution stops in MAIN. List the code in order to find a location where the Floating-Point Status Register (FSR) is likely to be set to the state it is going to be when the exception occurs, but the exception itself had not occurred yet.

Set a break point at that location. Continue execution. Execution will stop at the second break point. Examine the content of FSR. Modify the content of FSR so that the invalid operand exception will be trapped (see Figure 5-1). Continue execution. Execution will stop when the exception occurs (or shortly afterwards). One suspects that the exception is caused by `fsqrtd` or `faddd`, and that it happens in the function `routine.` Inquire the absolute address of the instruction containing `fsqrtd.`

With the suspicion that absolute address 0x10c64 raises an IEEE exception, one can use `dbx` to obtain more information about what function this address belongs to.

Since the program was not compiled with `–g`, `dbx` will not be able to provide symbolic information such as the line in the source code that caused the exception, but it will provide a useful traceback, and the name of the *function* that houses the code causing the instruction.

```
demo% dbx a.out
(dbx) stopi at 0x10c78
(2) stopi at 68728
(dbx) run
Running: find_exception
stopped in routine_ at 0x10c78
routine_+0x30:  faddd   %f4, %f6, %f8
(dbx) where
routine_(0x21e58, 0xcccccccd, 0xeffff840, 0x0, 0xef72dcfc, 0x0)
at 0x10c78
MAIN(0x34, 0x0, 0x0, 0x0, 0x0, 0xef7886e4) at 0x10bcc
```

Now we know that the exception is raised in the function called `routine`, called by `MAIN`. In this case this is enough information to determine the case if the invalid operation. In general, further investigation with `adb` can expose the parameters passed to `routine_`. The next things to look at is the value of the parameter passed to `routine`. For this we will run the program under `adb` one more time, stop at `routine_+0x1c`, and examine the content of the `f2`/f3 pair..

```
routine_+0x1c:b
:r
breakpoint      routine_+0x1c:  fsqrtd      %f2,%f4
<f2=F
                -4.2000000000000002e+00
```

We can see now that the exception is caused by the fact that the program tries to compute the square root of `-4.2000000000000002e+00.`

Run

This section offers up a grab-bag of runtime tips that may be useful to programmers of numerical software, as well as topics arising from frequently asked questions.

fix_libc_ *(SPARC on SunOS 4.x only)*

The libcx distributed as part of the value-added compiler releases targeted for SunOS 4.x contains a few components from libc that fix bugs in the SunOS 4.1 libc that affect programmers. All programs will get these fixes automatically unless the -nocx flag is specified at link time.

fix_libc_ is a no-op function that references all the libc bug-fix elements, primarily bug fixes related to base conversion. It is simply a vehicle for distributing necessary libc bug fixes to people using the unbundled compilers. It increases executable sizes by a fair amount compared to what they would have been if the items in question were loaded from libc.so, the shared library version of libc.a.

Use the -nocx flag (SunOS 4.x only) if the size of the executable is of critical concern. Refer to the compiler man pages (cc(1), f77(1), CC(1), pc(1)) for more details.

Bug fixes include:

Table 5-6 fix_libc_ bug fixes

_fix_libc_ contains:	Many bugs in base conversion routines are fixed. Also, base conversion is significantly faster, compared to SunOS 4.0. Quadruple precision base conversion is supported (%Lf). 64 bit long long integer base conversion is supported (%lld).

Exit Status of C Programs

C programs should always explicitly return an exit status for non-void functions:

```
/* at exit points of non-void C functions*/
return(0);
```

In the past, the C compiler supplied an implicit `return(0)` for non-void functions that lacked an explicit return value. The compiler no longer supplies an implicit `return(0)` for such functions. In addition to being good programming practice, supplying an explicit return value would eliminate errors reported from related software. Take for example the following program:

```
#include <stdio.h>
main()
{
printf("this message comes from main\n");
}
```

If run interactively from a shell, this program behaves as expected:

```
demo% a.out
this message comes from main
```

However, if the same program is executed in an environment which examines the program's exit status, unexpected results may occur. In particular, if the program is executed from a makefile, an unexpected error code may be reported:

```
demo% make
cc example.c -o example
example
this message comes from main
*** Error code 1
make: Fatal error: Command failed for target ‘example’
```

This message is explained by noting that `make` examines the exit status of each program that it invokes; the program's exit status is the value returned by `main()` or passed to `exit()`. If `main()` does not return a value, or call `exit()`, the exit status is undefined and the program is in error.

`ieee_retrospective` *Function*

The math library, `libsunmath.a`, provides the `ieee_retrospective` function. This function examines a few aspects of a program's floating-point runtime environment and prints warning messages if necessary. See Chapter 3, "The Math Libraries" for a detailed discussion of this function. `ieee_retrospective` is called by default from FORTRAN but not from C. To achieve the same functionality from C programs, add the following call prior to exit points:

```
#include <sunmath.h>
......
ieee_retrospective_();
```

Suppose one wants to suppress `ieee_retrospective` messages from FORTRAN programs. While it is not possible to disable ieee_retrospective, it is possible to wipe out some of the situations which cause ieee_retrospective to output messages. One approach is to clear all outstanding exceptions, enabled traps, and non-standard mode, just before the program exits.

In FORTRAN, clear outstanding exceptions by calling `ieee_flags` with the following syntax:

```
character*8 out
i = ieee_flags('clear', 'exception', 'all', out)
```

For C the syntax is:

```
char *out;
i = ieee_flags("clear", "exception", "all", &out);
```

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

Call `standard_arithmetic` to clear exceptions accrued in nonstandard mode. The FORTRAN syntax is

```
call standard_arithmetic()
```

For C the syntax is:

```
standard_arithmetic();
```

If the program sends no other messages to standard error, try redirecting the output of `stderr` to a file.

The final and least desirable approach is to include a dummy function in the program. For FORTRAN the syntax is:

```
subroutine ieee_retrospective
return
end
```

Nonstandard Arithmetic and Kernel Emulation (SPARC only)

There are several ways to set the FPU to nonstandard mode: compile with `-fast` or `-fnonstd` or invoke `nonstandard_arithmetic`() from inside the program.

Not all SPARC implementations provide a nonstandard mode. Trying to set it on implementations which do not provide it is ignored. For the SPARC implementations that provide a fast or nonstandard mode, setting the FPU to this mode means that some or all underflowed results (and/or operands) are flushed to zero. If for some reason a floating point operation that would underflow is interrupted (for example, it is in the queue when a context switch occurs), it is later emulated by kernel software, which always uses standard IEEE arithmetic.

Note – For x86, an abrupt underflow cannot happen because underflow is handled by hardware.

Nonstandard mode is not emulated by the kernel since its behavior is undefined, and implementation-dependent. Thus, under unusual circumstances, it could happen that an executable that sets the FPU to nonstandard mode *might* produce slightly varying results depending on system load. This behavior has *not* been observed. It would affect only those programs that are very sensitive to whether or not a particular computation (*from among millions*) was handled with gradual underflow or with abrupt underflow.

Compiler Versions

There are several ways to determine the version of the compiler. The `version` utility accesses the self-identifying version id string included in all compiler components:

- For SunOS 4.x, `version` is located in `/usr/lang/version`
- For SunOS 5.x, it is located in `/opt/SUNWspro/bin/version`

```
demo% version ‘which f77‘
version of "/opt/SUNWspro/bin/f77":SC3.0.1 26 May 1994
```

This is intended to help users determine if they are somehow accidentally mixing compiler components.

The compiler switch `–V` provides descriptive information about each compiler component.

Optimizing Large Programs

Optimizing large programs for performance is an art. The steps listed here serve as introduction to how you might go about experimenting with the compilers and profiling tools, and with the source code itself, to improve runtime performance.

- Profile, compiling with `–pg` and using `gprof` (1). This helps focusing on the modules which account for most of the runtime. Concentrate on optimizing those modules.
- Experiment with different combinations of compilation options for different modules. Compile numerically intensive modules at `–xO4` and compile modules that don’t affect runtime much at a low optimization level, or with no optimization. Other experiments are to compile with or without in-line templates (`–xlibmil`) and to experiment with linking statically or with linking dynamically (when both versions of the libraries are available).

- Use `tcov` (1) to learn which loops, within modules, account for most of the computation time. Break out inner loops to smaller modules if that improves performance.
- Do some source level recoding if appropriate (for example reroll Linpack BLAS). Note that FORTRAN is often better than C for number crunching. Avoid very large subroutines. Initialize local variables.
- Do assembly language coding through `.s` or `.il` files where appropriate.

Refer to *SPARCompiler Optimization Technology Technical White Paper*, April 1990, Part number FE274-0/20K, for more details.

Refer to “You and Your Compiler or Tickling Your Compiler”, *SunScope 94 Exhibition and Conference Proceedings*, London, UK, January 1994. Complete text available via sunsite.doc.ic.ac.uk in `/sun/Papers/SunPerfOv+references/you_and_your_compiler.gz`

Examples A

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 version of `libm` and `libsunmath`. These examples were tested with the current C and FORTRAN compilers, on the Solaris operating system.

This appendix is organized into the following sections:

IEEE Arithmetic
The Math Libraries
Exceptions and Signal Handling
Compile-Debug-Run

IEEE Arithmetic

Examining Stored IEEE hexadecimal Representations

These examples show a way of examining the hexadecimal representation of floating-point numbers. Note that one can also use the debuggers to look at the hexadecimal representation of stored data, as well as write programs based on other ideas.

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

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

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

      /* 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);

     exit(0);
}
```

The output of this program looks like:

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

SPARC

And from FORTRAN:

```
      program print_ieee_values
c
c the purpose of the implicit statements is to insure
c that the f77_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()
      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
```

x86

x86 does not support quadruple precision; real*16. To run the FORTRAN example on x86, delete the real*16 declaration and the q_min_normal() function that calculates and prints the quadruple precision value.

```
      implicit real*16 (q)
   ...
      z = q_min_normal()
      write(*,7) z, z
 7    format('min normal, quad: ',1pe47.37e4,/,' in hex
',z32.32)
```

SPARC

The corresponding output looks like:

```
min normal, quad:   3.3621031431120935062626778173217526026E-4932
   in hex 00000000000000000000000000010000
min normal, double:2.2250738585072014-308 in hex 0010000000000000
min normal, single:1.1754944E-38 in hex 00800000
```

The Math Libraries

In this section, we include a few 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:

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

#define SIZE 400000

      program example

c
      implicit GENERIC (a-h,o-z)
      GENERIC x(SIZE), y, lb, ub
      real time(2), u1, u2

c
c will 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(time)
      u1 = time(1)
c
c compute exponentials
      do 16 i=1,SIZE
          y = exp(x(i))
 16   continue
c
c get the elapsed time
   call dtime(time)
   u2 = time(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
```

Note that compilation should be done using either the `-DDP` or `-DSP` flag. The suffix is `F` instead of `f` so that the preprocessor is invoked automatically.

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

```
/*
 * 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

main()
{
      double   x[SIZE], y[SIZE];
```

```
        int       i, j, n;
        double    lb, 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;
            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_();
        exit(0);
  }
```

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

```
/*
      Demonstrate how to call 5 of the more interesting IEEE
      recommended functions from fortran.  These are implemented
      with "bit-twiddling", and so are as efficient as one could
      hope. The IEEE standard for floating-point arithmetic
      doesn't require these, but recommends that they be
      included in any IEEE programming environment.

      For example, to accomplish
          y = x * 2**n,
      since the hardware stores numbers in base 2,
      shift the exponent by n places.

      Refer to

      ieee_functions(3m)
      libm_double(3f)
      libm_single(3f)

      The 5 functions demonstrated here are:

      ilogb(x):  returns the base 2 unbiased exponent of x in
               integer format
      signbit(x): returns the sign bit, 0 or 1
      copysign(x,y): returns x with y's sign bit
      nextafter(x,y): next representable number after x, in
               the direction y
      scalbn(x,n): x * 2**n

      function         double precision    single precision
      --------------------------------------------------------
      ilogb(x)         i = id_ilogb(x)     i = ir_ilogb(r)
      signbit(x)       i = id_signbit(x)   i = ir_signbit(r)
      copysign(x,y)    x = d_copysign(x,y) r = r_copysign(r,s)
      nextafter(x,y)   z = d_nextafter(x,y)r = r_nextafter(r,s)
      scalbn(x,n)      x = d_scalbn(x,n)   r = r_scalbn(r,n)
*/



#include <values.h>
```

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

      x = MINDOUBLE
      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

      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 looks like:

```
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    -5.5000000000000 is   1
Starting from  4.9406564584124654-324, the next representable
   number towards -Inf is  0.0000000000000000E+00
Starting from  4.9406564584124654-324, the next representable
   number towards  1.0 is  9.8813129168249309-324
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    -5.50000 is   1
-5.5 was given the sign of 12.4 and is now  5.5
Starting from  1.4012984643248171E-45, the next representable
   number towards -Inf is  0.0000000000000000E+00
Starting from  1.4012984643248171E-45, the next representable
   number towards  1.0 is  2.8025969286496341E-45
Scaling  2.0 by 2**3 is 16.0

Note: the following IEEE floating-point arithmetic exceptions
occurred and were never cleared; see ieee_flags(3M):
Inexact;  Underflow;
Sun's implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
```

ieee_values

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

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

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]);

      exit(0);
}
```

When linking one should use both `-lm` and `-lsunmath`.

SPARC The output is similar to:

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

x86

The output is similar to:

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

SPARC

FORTRAN programs that use `ieee_values` functions should take care to declare the function’s type:

```
      program print_ieee_values
c
c the purpose of the implicit statements is to insure
c that the f77_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
```

x86

x86 does not support quadruple precision; real*16. To run the FORTRAN example on x86, delete the real*16 declarations, the assignment of zero and one, the q_nextafter() function, and the statement to print z.

```
        implicit real*16 (q)
     ...
        real*16 z, zero, one
     ...
        zero = 0.0
        one = 1.0
        z = q_nextafter(zero, one)
     ...
        print *, z
```

SPARC

The output is similar to:

```
      6.47517511943802511092443895822764655-4966
   Infinity
      3.40282E+38
```

Exceptions and Signal Handling

`ieee_flags` - *Rounding Direction*

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

```
#include <math.h>
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);

     exit(0);
}
```

SPARC

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

```
demo% cc rounding_direction.c -lm
demo% a.out
With rounding direction nearest,
sqrt(.5) = 0x667f3bcd 0x3fe6a09a = 7.071067811865476e-01

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

x86

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

```
demo% cc rounding_direction.c -lm
demo% a.out
With rounding direction nearest,
sqrt(.5) = 0x3fe6a09a 0x667f3bcd = 7.071067811865476e-01

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

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:

```
 Rounding direction is: tozero
 Note: Rounding direction toward zero; see ieee_flags(3M).
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

`ieee_flags` - *Accrued Exceptions*

Generally, a user program *examines* or *clears* the accrued exception bits. Here is a C program that examines the accrued exception flags:

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

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 integer value that describes all raised */
      /* exceptions fp_inexact, fp_division, ... defined in */
      /* <sys/ieeefp.h>                                     */
      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;

     /* "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);

      exit(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: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Inexact;  Underflow;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

The same can be done from 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
f77_floatingpoint.h.
*/
#include <f77_floatingpoint.h>

      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
```

x86

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 <sys/ieeefp.h>
/*
 * from <sys/ieeefp.h>, the exceptions according to bit number are
 *   fp_invalid      = 0,
 *   fp_denormalized = 1,
 *   fp_division     = 2,
 *   fp_overflow     = 3,
 *   fp_underflow    = 4,
 *   fp_inexact      = 5
 */

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);

      exit(0);
}
```

SPARC

For a SPARC machine, the exception flags have the following definitions:

```
/*
 *from <sys/ieeefp.h>, the exceptions according to bit number are
 *    fp_inexact      = 0,
 *    fp_division     = 1,
 *    fp_underflow    = 2,
 *    fp_overflow     = 3,
 *    fp_invalid      = 4
 */
```

x86

The output of the x86 C example is:

```
out is: division , fp exception code is: 2
```

ieee_handler - *Trapping Exceptions*

We now present C and FORTRAN examples that use `ieee_handler`:

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

extern void continue_hdl(int sig, siginfo_t *sip,
   ucontext_t *uap);

main() {
      char           *out;
      double          x;

      /* trap on common floating point exceptions */
      if (ieee_handler("set", "common", continue_hdl) != 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) */
      /* the value printed out is unrelated to the result */
      x = max_normal();
      x = x*x;
      printf("max_normal * max_normal = %g\n", x);

      ieee_retrospective_();

      exit(0);
}

void continue_hdl(int sig, siginfo_t *sip, ucontext_t *uap) {
      printf("fp exception %x at address %x\n",
      sip->si_code, sip->_data._fault._addr);
}
```

The output from this C program is:

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
 Note: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Inexact;  Underflow;
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Overflow;  Division by Zero;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

Here is the FORTRAN program:

```
      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
      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: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Inexact;  Underflow;
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Overflow;  Division by Zero;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

SPARC

We will now give a somewhat more complex example:

```
/*
 * Generate the 5 IEEE exceptions: invalid, division,
 * overflow, underflow and inexact.
 *
 * Trap on any floating point exception, print a message,
 * and continue.
 *
 * Note that one could also inquire about raised exceptions by
 *    i = ieee("get","exception","",&out);
 * where out will contain 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);

main()
{
      double   x, 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;

        /* 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 */
        exit(0);
}

void trap_all_fp_exc(int sig, siginfo_t *sip, ucontext_t *uap) {
        char     *label = "undefined";

/* 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: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Underflow;  Overflow;  Division by Zero;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

SPARC

The following program shows how one can use `ieee_handler` and the include files to modify the default result of certain 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);

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);
      /* 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_();

      exit(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 = *(int *) sip->_data._fault._addr;

      /*
       * 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.
       */
      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 will be:

```
double precision division: 1/0 = 1.79769e+308
single precision division: 1/0 = 3.40282e+38
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Division by Zero;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

`ieee_handler` - *Abort on Exceptions*

One can use `ieee_handler` to force a program to abort in case of certain floating-point exceptions:

```
#include <f77_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
```

Compile-Debug-Run

In this section we present miscellaneous examples which might be interesting, or useful.

In the previous section we showed examples of using `ieee_handler`. In general, when there is a choice between using `ieee_handler` or `sigfpe`, the former is recommended.

SPARC

There are instances, e.g. when trapping integer arithmetic exceptions, when `sigfpe` is the handler to be used. The following example traps on integer division by zero:

```
/* 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);

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);

      a = 4;
      b = 0;
      c = a / b;
      printf("%d / %d = %d\n\n", a, b, c);

      exit(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 program counter; ieee_handler does this by default,
 * but here we have to use sigfpe to set up the signal handler
 * for int div by 0
 */
      uap->uc_mcontext.gregs[REG_PC] =
          uap->uc_mcontext.gregs[REG_nPC];
}
```

In conclusion, we present 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 `driver.c`):

```
/*
 * a demo program that shows:
 *
 * 1. how to call f77 subroutine from C, passing an array argument
 * 2. how to call single precision f77 function from C
 * 3. how to call double precision f77 function from C
 */

extern int      demo_one_(double *);
extern float    demo_two_(float *);
extern double   demo_three_(double *);

main()
{
      double   array[3][4];
      float    f, g;
      double   x, 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_();
        exit(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
f77 -c drivee.f
     demo_one:
     demo_two:
     demo_three:
f77 -o driver driver.o drivee.o
```

The output will look like this:

```
from the fortran routine:
   array[  1][  1] =   0.
   array[  1][  2] =     1.0000000000000
   array[  1][  3] =     2.0000000000000

   array[  2][  1] =     2.0000000000000
   array[  2][  2] =     3.0000000000000
   array[  2][  3] =     4.0000000000000

   array[  3][  1] =     4.0000000000000
   array[  3][  2] =     5.0000000000000
   array[  3][  3] =     6.0000000000000

   array[  4][  1] =     6.0000000000000
   array[  4][  2] =     7.0000000000000
   array[  4][  3] =     8.0000000000000

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
```

SPARC Behavior and Implementation B

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.

This appendix has the following organization:

Floating-point Hardware

This section describes details of SPARC implementation of IEEE exceptions. See the *SPARC Architecture Manual*, Version 8, Appendix N “SPARC IEEE 754 Implementation Recommendations” for a brief description of what happens when a trap is taken, the distinction between trapped and untrapped underflow, and recommended possible courses of action for SPARC implementations that choose to provide non-IEEE (nonstandard) arithmetic mode.

Many SPARC systems have floating point units derived from cores developed by TI or Weitek.

- the TI family includes the TI8847 and the TMS390C602A.
- the 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 SPARC workstations.

Table B-1 lists the hardware floating-point implementations used by SPARC workstations. Since the SPARC architecture defines the instruction set implemented by a SPARC floating-point unit, the chips differ more in the technology used to construct them than in functionality.

Table B-1 SPARC Floating-Point Options

FPU	**Description**	**Appropriate for Machines**	**Notes**	**Compilation Switch**
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	-xcg89
Weitek 3170-based FPU		SPARCstation 1 (4/60) SPARCstation 1+ (4/65)	1989, 1990	-xcg89
TI 602a		SPARCstation 2 (4/75)	1990	-xcg89
Weitek 3172-based FPU		SPARCstation SLC (4/20) SPARCstation IPC (4/40)	1990	-xcg89

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
				-xcg89

Cypress 602	Resides on Mbus Module	SPARCserver 6xx	1991	-xcg89
TI TMS390Z50	SuperSPARC or SuperSPARC+	SPARCserver 6xx SPARCstation 10 Model 30, 40, 41, 52, 54, 402MP, 512MP	1992, 1993	`-xcg92`
TI TMS390S10	microSPARC	SPARCstation LX (4/30)	1992	-xcg92
TI TMS390S10	microSPARC	SPARCclassic (4/15)	1992	-xcg92

Table B-1 SPARC Floating-Point Options (Continued)

FPU	Description	Appropriate for Machines	Notes	Compilation Switch
TI TMS390Z50	SuperSPARC or SuperSPARC+	SPARCServer 1000 SPARCCenter 2000	1992, 1993	-xcg92
TI TMS390Z50 or Cypress 602	SuperSPARC or Cypress SPARC	SPARCSystem 6xxMP	1992, 1993	-xcg92
Weitek 1164/1165-based FPU or no FPU	Kernel emulates floating-point instructions	Obsolete	Slow. not recommended	-xcg89 or -xcg92

The systems based on SPARC FPUs preceding SuperSPARC, SuperSPARC+ and microSPARC implement the floating-point instruction set defined in the SPARC Architecture Manual Version 7. The systems based on the SuperSPARC, SuperSPARC+ and microSPARC FPUs implement the floating-point instruction set defined in the SPARC Architecture Manual Version 8.

The SuperSPARC and SuperSPARC+ FPUs implement the floating-point instruction set defined in the SPARC Architecture Manual Version 8 in hardware, except the quad precision instructions. Exceptional cases are handled in hardware.

The microSPARC FPUs implement SPARC Architecture Manual Version 8 floating-point instruction set in hardware, except FsMULd and quad precision instructions. Thus, for single-precision complex variables, use `-xcg89` instead of `-xcg92` for microSPARC FORTRAN programs.

Any unimplemented floating-point instruction causes a trap to the system kernel, which then emulates it. Similarly, running on a system with no FPU or disabled FPU, will emulate the floating-point instructions in software. Usually this causes a severe performance degradation.

The default code generation switch is `-xcg89` (it used to be `-xcg87` in SunOS 4.1.*x)*. The `-xcg89` and `-xcg92` flags are for `cc` and `f77`.

In accord with RISC (reduced instruction set computer) philosophy, complicated instructions such as the transcendental math functions are not implemented in hardware.

Handling Subnormal Results

On the TI-derived FPUs, a combined ALU and multiplier unit, the ALU wraps subnormal results whether `SIGFPE` has been enabled or not. (A *wrapped number* is created by multiplying the correct result by a constant power of two, prior to rounding.) But this wrapped result is never seen by software, since destination registers are never changed when traps occur. The correct subnormal result is computed by system software. The TI 8847 chip can be operated in nonstandard underflow mode, which forces all subnormals (inputs and outputs) to be flushed to zero without system software intervention.

In IEEE gradual underflow mode, if a wrapped result is produced by the multiplier, the result is passed to the ALU and unwrapped.

The TI 8847-based FPU nonstandard mode corresponds better to the intent of providing fast processing of small results than does the nonstandard mode of the Weitek 1164/1165-based FPU, which treats subnormal operands, but not results, as zero.

Status and Control Registers

The SPARC FPU has status and control registers associated with it: the floating-point status register (FSR) and the floating-point queue (FQ) set of control registers. Kernel software uses the FQ to recover from floating-point exceptions.

The floating-point status register (FSR) contains FPU mode and status information. The FSR is visible to user processes, and most fields can be written as well. The SPARC assembler and debugger know it as `%fsr`. Examples of `%fsr` use may be found in the file named `libmil`, containing in-line templates.

Figure 5-1 shows the bit assignments of the Floating-Point Status Register.

RD	u	TEM	NS	res	*ver*	*ftt*	*qne*	*u*	*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 5-1 SPARC Floating-Point Status Register

The fields relevant to exception handling are `TEM`, `NS`, `aexc` and `cexc`.

Field	Corresponding bits in register				
TEM, trap enable mask	NVM 27	OFM 26	UFM 25	DZM 24	NXM 23
NS, non standard floating-point			NS 22		
aexc, accrued exception bits	nva 9	ofa 8	ufa 7	dza 6	nxa 5
cexc, current exception bits	nvc 4	ofc 3	ufc 2	dzc 1	nxc 0

The trap enable masks, the bit that indicates nonstandard floating-point, and the exception bits are either on or off. The low-order bit is bit 0.

The current exception bits are updated by the hardware when each floating-point operation successfully completes.

Handling Floating-point Exceptions

There are two cases when the hardware does not successfully complete a floating-point operation:

- The operation is unimplemented (such as, the unimplemented `fsqrt[sd]` on Weitek 1164/1165-based FPUs).
- The hardware is unable to deliver the correct result.

In these cases, the process traps to the kernel, which emulates the floating-point operation and updates the FSR and destination registers.

If a floating-point exception results in a trap, then the destination floating-point register, the floating-point condition codes, *(fcc)*, and the *aexc* fields remain unchanged. The *cexc* field is updated to show which exception caused the trap. (If the trap is caused by an unfinished or unimplemented floating point operation, instead of by one of the IEEE 754 floating point exceptions, then *cexc* is also unchanged.)

If the exception does not result in a trap, then the destination register, *fcc*, *aexc* and *cexc* are updated to their new values.

The following pseudo-code summarizes the handling of IEEE traps. Note that the *aexc* field can normally only be cleared by software.

```
FPop generates an IEEE exception;
texc ← IEEE exceptions generated by this FPop;
if (texc and TEM) = 0
  then(aexc ← (aexc or texc); cexc = texc; f[]←result;
        fcc ← fcc_result)
  else (cause fp_exception_trap)
```

Handling Gradual Underflow

In floating-point environments built on the architecture of some of the SPARC processors (TI TMS390Z5 and TI TMS390S10), and the Intel x86 family, gradually underflowed results are almost always calculated by the floating-point unit in the cpu or coprocessor. Therefore, gradual underflow is much less likely to cause a performance degradation than it is when implemented in software. (For more information about gradual underflow, see the discussion of “Underflow” in Chapter 2, “IEEE Arithmetic”.)

If an application encounters frequent underflows, the user may want to determine how much system time the application 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
```

To determine if underflows occurred in an application, one can use the math library function `ieee_retrospective` to check whether or not the underflow exception flag is raised when the program exits. FORTRAN programs call `ieee_retrospective` by default and C and C++ programs need to call `ieee_retrospective` prior to any exit points.

The function `ieee_retrospective` prints a message similar to the following to standard error, `stderr`:

```
Note: the following IEEE floating-point arithmetic
exceptions occurred and were never cleared; see ieee_flags(3M):
Inexact; Underflow;
Sun’s implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
```

The math library provides two functions to help programs toggle between standard underflow mode and nonstandard underflow mode. A call to `nonstandard_arithmetic` turns off IEEE gradual underflow (if applicable), and a call to `standard_arithmetic` restores IEEE behavior.

C, C++	`nonstandard_arithmetic();`
	`standard_arithmetic();`

FORTRAN	`call nonstandard_arithmetic()`
	`call standard_arithmetic()`

One should use `nonstandard_arithmetic` with caution, since it causes the loss of the accuracy benefits of gradual underflow.

The `fpversion`*(1) Function — Finding Information about the FPU*

The utility `fpversion`(1), distributed with the unbundled compilers, identifies the installed CPU and FPU and estimates their clock speeds.

The `fpversion`(1) function determines the CPU and FPU type by interpreting the identification information stored by the CPU and FPU. The `fpversion` function is installed with the unbundled compilers, and is usually located in the same directory as the unbundled compilers.

On a SPARCstation 2, the information returned from `fpversion` is similar to this example (there might be differences depending on configuration, of course):

```
demo% fpversion
 A SPARC-based CPU is available.
 CPU's clock rate appears to be approximately 39.3 MHz.
 The clock rate is probably 40.0 Mz.

 Sun-4 floating-point controller version 2 found.
 A TI TMS390C602A-based FPU is available.
 FPU's frequency appears to be approximately 38.6 MHz.
 The clock rate is probably 40.0 Mz.

 Use “-xcg89” floating-point option.

 Hostid = 0x67003A21.
```

Note that `fpversion` is not instantaneous. Indeed, `fpversion` might take about a minute of user time to run—this is deliberate. `fpversion` determines the approximate clock rates for the CPU and FPU by timing a loop that executes simple instructions that run in a predictable amount of time.

The loop is executed relatively many times to assure that the timing measurements and clock rate estimates are correct.

Floating-point Hardware—Disable/Enable

It is possible to disable the Floating-Point Unit (FPU) on a SPARC workstation. This might be useful when trying to establish that the FPU is faulty. The probability of faulty hardware is far exceeded by the probability of programming error, and so the following examples should be used with extreme caution.

Floating-point hardware can be disabled/enabled in software by running the following scripts as root.

To disable the FPU:

```
#!/bin/sh -
# script to turn the FPU off
#
adb -k -w /kernel/unix /dev/mem <<!
fpu_exists/W0
!
```

It can be enabled again this way:

```
#!/bin/sh -
# script to turn the FPU on
#
adb -k -w /kernel/unix /dev/mem <<!
fpu_exists/W1
!
```

These scripts may be run in normal multi-user mode without rebooting, but one should avoid running them if any processes that use floating-point instructions are executing.

x86 Behavior and Implementation C

This appendix discusses x86 and SPARC compatibilty issues related to the floating-point units used in x86 machines

The hardware is 80386, 80486, and pentium microprocessors from Intel and compatible microprocessors from other manufacturers. While great effort went into compatiblity with the SPARC platform, several differences exist.

- On x86, real*16 is not implemented in FORTRAN.
- 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.
- On x86, each time a float or double floating-point number is stored or loaded, a conversion to double extended occurs; thus, raising the inexact flag much more often than in a SPARC machine.
- On x86, gradual underflow is implemented entirely in hardware.
- On x86, the `fpversion` utility is unavailable.

Standards Compliance D

The compilers, header files and libraries in the SPARCompilers language products for Solaris 2.*x* support multiple standards: System V Interface Definition, Edition 3, X/Open and ANSI C. Accordingly, the mathematical library `libm.a` and related files have been modified so that C programs comply with the standards. Users' programs are usually not affected, since the differences primarily involve exception handling.

This appendix has the followiong organization:

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. Since this would have 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 since 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 will replace references to `HUGE`, intended to be a large finite number, with `HUGE_VAL`, which is infinity on IEEE systems. `HUGE_VAL` would, for instance, be returned as the result of floating-point overflows. The values returned by libm functions for input arguments that raise exceptions would be those in the IEEE column in Table B-1, below. In addition, errno will no longer need 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 via `-xlibmil`, since this causes the hardware instruction for square root, `fsqrt[sd]`, to be used in place of a function call.

Table D-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	SING	+-infinity	+-infinity	+-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

Table D-1 Exceptional Cases and libm Functions

Function	errno	error message	SVID	X/Open	IEEE
lgamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
log(0)	EDOM	SING	-HUGE	-HUGE_VAL	infinity
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM	SING	-HUGE	-HUGE_VAL	-infinity
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
pow(0,0)	EDOM	DOMAIN	0.0	1.0 (no error)	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 Table D 1

Table D-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 SunOS 5.x.
- 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 would be 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 D-1 on page 147.
- 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` will use the SVID compliant versions of the libm functions listed in Table D-1 on page 147.

- For efficiency, programs compiled with in-line 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 will conform to the IEEE Floating-Point Standard, but may no longer conform to the error handling requirements of the System V Interface Definition.

`libm.a` *Notes*

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)` should be zero if `x/y` would overflow, 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.

References E

These manuals contain information on SPARCompiler language processors:

- *C 3.0.1 Transition Guide for SPARC Systems*
- *C 3.0.1 User's Guide*
- *C++ 4.0.1 User's Guide*
- *C++ 4.0.1 Library Reference Manual*
- *Tools.h++ Introduction and Reference Manual*
- *C++ 4.0.1 AnswerBook*
- *FORTRAN 3.0.1 Reference Manual*
- *FORTRAN 3.0.1 User's Guide*
- *FORTRAN 3.0.1 AnswerBook*
- *SPARCompiler Pascal 3.0.3 Reference Manual*
- *SPARCompiler Pascal 3.0.3 User's Guide*
- *3.0.1 Common Tools & Related Material AnswerBook*
- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*

The SPARC manual provides more information about SPARC floating-point hardware.

SPARC Architecture Manual, Version 8, Prentice-Hall, New Jersey, 1992.[1]

The following manual provides information about debugging tools and hardware diagnostics.

Debugging a Program

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.

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.

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, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974 (Out of print; most university libraries have copies.)

1. Also available by ordering through SPARC International Inc., 535 Middlefield Road, Suite 210, Milpitas, CA, 94025, (415) 321-8692, Order # SAV080SI9106

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.

Kahan, W., “A Survey of Error Analysis”, *Information Processing 71*, North-Holland, Amsterdam, 1972, pp 1214-1239.

Chapter 5: “Compile-Debug-Run”

Bentley, Jon Louis, *Writing Efficient Programs*, *Programming Pearls*, *More Programming Tools*, Prentice-Hall, Englewood Cliffs, New Jersey.

Bierman, Keith H., “You and Your Compiler or Tickling Your Compiler”, *SunScope 94 Exhibition and Conference Proceedings*, London, UK, January 1994. Complete text available via sunsite.doc.ic.ac.uk in `/sun/Papers/SunPerfOv+references/you_and_your_compiler.gz`

Bunch, J., Dongarra, J., Moler, C., Stewart, G., *Linpack User’s Guide*, SIAM, Philadelphia, 1979.

Farnum, C., “Compiler Support for Floating-Point Computation”, *Software-Practice and Experience*, Vol 18, 1988.

Metcalf, M., *FORTRAN Optimization*, Academic Press, 1982.

Muchnick, Steven S., The Sun Compiling System,” *Optimization in Compilers*, published by ACM Press/Addison-Wesley, 1992.

Appendix B: “SPARC Behavior and Implementation”

Manufacturer’s Documentation

The following documentation will provide 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

For information on machine-readable source of the IEEE Test Vectors or the SPICE program, contact EECS/ERL Industrial Relations Office, 461 Cory Hall, University of California, Berkeley, CA 94270.

For information on the Berkeley Elementary Functions test programs used to measure accuracy of elementary functions, contact Alex Liu at `Alex.Liu@Eng.Sun.COM`.

Netlib provides a numerical source code distribution service from which many popular benchmark programs can be obtained, including `paranoia`, `elefunt`, `linpack`, `whetstone` and `dhrystone`, as well as implementations of many of the Collected Algorithms of the ACM, and the current version of

Dongarra’s Technical Memorandum containing Linpack benchmark results. For information on benchmarks, send email with Subject: help to `netlib@research.att.COM`.

Glossary

This glossary describes computer floating-point arithmetic terms. It also describes terms and acronymns associated with parallel processing.

|| appended to a term designates it as one associated with parallel processing.

accuracy	An answer is judged accurate when it is believed to be unaffected by error. Contrast with precision. For example, “the result is accurate to six decimal places” implies that all the errors incurred while calculating the result are not large enough to change the sixth decimal place of the result.
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 in which 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 mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock.
backplane||	See MBus, multiprocessor bus, XDBus.
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 may 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 may 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 a schedulable entity 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 may also generate a write-through to main memory.

Cache for the SuperSPARC processor is organized into the following hierarchy:

- first-level instruction cache. The 20Kbytes, on-chip, instruction cache fetches up to four instructions per cycle without memory wait states.
- first-level data cache. The 16Kbytes, on-chip, data cache handles one 64-bit load or store per cycle.

- second-level external cache. The 1Mbyte (optional 2Mbytes) external cache is optimized to handle high bandwidth burst data transfers between the processor and cache.

Tables 1, 2, and 3 summarize the characteristics of the SuperSPARC cache.

Table 1: First-level Instruction Cache

SPARCServer Type	**Total Cache**	**Associ-ativity**	**Block Size**	**Sub-blocks**	**Write Policy**	**Bus Type & Protocol**
SS10 (no ext cache)	20Kbytes	5-way	64 bytes	2	write-back	MBus / circuit switched
SS10 (ext cache)	20Kbytes	5-way	64 bytes	2	write-through	MBus / circuit switched
SS1000	20KByte	5-way	64 bytes	2	write-through	XDBus / packet switched
SS2000	20Kbytes	5-way	64 bytes	2	write-through	XDBus / packet switched

Table 2: First-level Data Cache

SPARCServer Type	**Total Cache**	**Associ-ativity**	**Block Size**	**Sub-blocks**	**Write Policy**	**Bus Type & Protocol**
SS10 (no ext cache)	16Kbytes	4-way	32 bytes	none	write-back	MBus / circuit switched
SS10 (ext cache)	16Kbytes	4-way	32 bytes	none	write-through	MBus / circuit switched
SS1000	16KByte	4-way	32 bytes	none	write-through	XDBus / packet switched
SS2000	16Kbytes	4-way	32 bytes	none	write-through	XDBus / packet switched

Table 3: Second-level External Cache

SPARCServer Type	Total Cache	Associ-ativity	Block Size	Sub-blocks	Write Policy	Bus Type & Protocol
SS10 (no ext cache)	none	none	none	none	none	none
SS10 (ext cache)	1Mbyte	direct map	128 bytes	4	write-back	MBus / circuit switched
SS1000	1MByte	direct map	256 bytes	4	write-back	XDBus / packet switched
SS2000	1Mbyte	direct map	256 bytes	4	write-back	XDBus / packet switched
SS2000 (mod. no. xx)	2Mbytes	direct map	256	4	write-back	XDBus / packet switched

See 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 re-use 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 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 cache, competitive-caching, false sharing, write-invalidate, write-update.

chaining	A hardware feature of some pipelined 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; not the disparaging definition of common.
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 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 asynchronous control, multiprocessor system, thread.
concurrent processes||	Processes that execute in parallel in multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrence of an external event. See 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 may 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 data flow model, demand-driven dataflow.
critical region	An indivisible section of code that may 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 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 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 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 mutual exclusion, mutex lock, semaphore lock, single-lock strategy, spin lock.

deadlock||
A situation that may 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 which is also enabled; such as, a graph reduction model. A graph reduction program consists of reducible expressions which 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 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 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 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 would be 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 radix 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 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 cache, cache locality, direct mapped cache, false sharing, set associative cache, write-invalidate, write-update.
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 SPARCompilers. 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 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 which are called light-weight processes (LWPs). In Solaris 2.2 and above, 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 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 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 Solaris 2.2 and above, 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 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 lockstep 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 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 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 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 bound thread, mt-safe, single-lock strategy, thread, unbound thread, uniprocessor.
mutex lock	Synchronization variable to implement the mutual exclusion mechanism. See 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 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 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 may be inaccurate because it only approximates the real world. See 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 concurrence, multiprocessor system, multithreading, uniprocessor.
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 may be passed on to the parsing phase while the lexical analyser starts on the second program or module. See 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 (e.g., 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. “IEEE double precision format specifies 53 bits of precision” implies that the relative representation error in the normal range is bounded by 2^{-52}.
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 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 semaphore lock.
sequential processes||	Processes that execute in such a manner that one must finish before the next begins. See 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 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 via 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	A component of a floating-point number that consists of an explicit or implicit leading point to the left of the implied binary point, and a fraction field 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 array processing, pipeline, vector processing.
Single Instruction Single Data||	The conventional uniprocessor model, with a single processor fetching and executing a sequence of instructions which 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 lockstep coordination. In SPMD, processors may 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 may 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 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 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 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 non-zero floating point number with a biased exponent of zero. The subnormal numbers are those between zero and the smallest normal number.
thread‖	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 bound thread, light-weight processes, multithreading, unbound thread.
topology‖	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‖	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 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||	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 multithreading, single instruction single data, single-lock strategy.
vector processing||	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 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 would otherwise overflow or underflow 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 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 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 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 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 cache, coherence, competitive-caching, write-invalidate, write-update.

Index