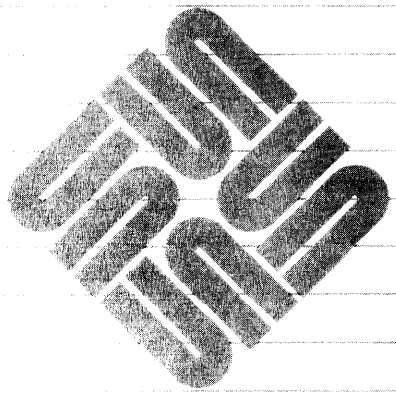
sun®
microsystems

Floating-Point Programmer's Guide
for the Sun Workstation®

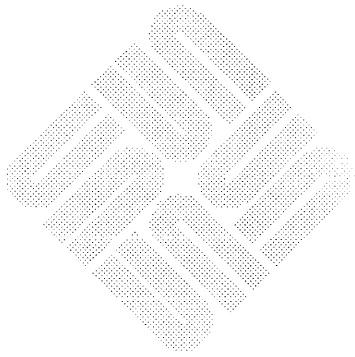


Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Trademarks and Copyright

Revision History

Revision	Date	Comments
50	2 June 1986	3.2 β release
51	23 June 1986	3.2 postβ fixes
A	12 September 1986	3.2 FCS release



Contents

Preface

Sun Microsystems provides a variety of software and hardware floating-point options. Some options provide fast performance; most conform substantially, and all conform at least partially, to the requirements of the ANSI/IEEE Std 754-1985, the IEEE Standard for Binary Floating-Point Arithmetic.

Manual Scope

This manual describes features of Sun Microsystem's various available floating-point implementations, including details of their use, performance, and conformance to the IEEE floating-point arithmetic standard.

Note:

All specific claims for conformance, accuracy, and speed are based on tests and measurements made using a preliminary version of Sun Software Release 3.2. Results obtained with the final Release 3.2 or with other releases may vary.

Audience

This manual is limited to information not easily obtainable elsewhere; you should be familiar with the ANSI/IEEE Standard 754-1985, with the MC68881 floating-point processor (if you have a Sun-3), and the Weitek WTL 1164/1165 chip set (if you have a Floating-Point Accelerator or FPA).

If you want or need more information than is contained in this manual, see one or more of the sources listed in the bibliography later in this preface.

What is in This Manual

The description of floating-point options on Sun Microsystems computers is divided into four chapters and eight appendices:

Chapter 1 describes the hardware and software options that can affect floating-point numeric operations on Sun Microsystems computers including hardware, compiler code generation, and expression evaluation.

Chapter 2 describes floating-point numerics as applied to Sun Micosystems hardware and software. This includes discussions of rounding errors and differring numerical results depending on different floating-point options, using floating-point operations from compiled and assembly language programs, and operational details of Sun Microsystems' floating-point implementations.

Chapter 3 describes aspects of the ANSI/IEEE Standard 754-1985, and Sun Microsystems floating-point conformance to the standard.

Chapter 4 describes benchmarks that have been and can be used to test the performance and conformance of floating-point numerics on Sun Microsystems computers.

Appendix A describes changes made to `adb` to support the FPA, examples of FPA disassembly, and FPA register use.

Appendix B describes changes made to `dbx` and `dbxtool` to support the FPA, examples of FPA disassembly, and FPA register use.

Appendix C describes the FPA assembly-language syntax supported by `as` in Release 3.1 and later.

Appendix D describes a number of IEEE floating-point functions that are not usually available in higher-level languages.

Appendix E describes the input files used by the `SPICE` benchmark program.

Appendix F describes the differences in function between the two different mask versions of the MC68881 coprocessor used by Sun's floating-point processors.

Appendix G describes assembly-level inline code expansions that let you integrate assembly-language routines into C, FORTRAN, and Pascal programs.

Appendix H describes aspects of the floating-point implementation affected by compliance with the UNIX System V interface.

References for Further Information

The following provide more information about Sun's language processors:

FORTRAN Programmer's Guide, 800-1371-02, Sun Microsystems, 1986.

Pascal Programmer's Guide, 800-1376-01, Sun Microsystems, 1986.

Assembly Language Reference Manual, 800-1372-02, Sun Microsystems, 1986.

The following provide more information about Sun's floating-point hardware:

Sun Floating-Point Accelerator User's Manual, 800-1378-02, 1986.

Preliminary Data, WTL 1164/1165 Low-Latency 64-bit IEEE Floating Point Multiplier/ALU, 231669-001, Weitek Corp., Sunnyvale, CA, 1986.

MC68881 Floating-Point Coprocessor User's Manual, MC68881UM/AD, Prentice-Hall, 1985.

MC68020 32-Bit Microprocessor User's Manual, second edition, MC68020UM/AD REV 1, Motorola Inc., 1985.

Fast Floating-Point Processor Integration Manual, Sky Computers, Lowell, MA, October 1984.

The following provide more information about the IEEE Standard:

IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, IEEE, New York, 1985.

Apple Numerics Manual, Addison-Wesley, 1986.

Coonen, *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic,* Ph. D. thesis, University of California, Berkeley, 1984.

Cody et al., "A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic," *IEEE Computer,* August 1984.

Stevenson et al., Cody, Hough, Coonen, various papers proposing and analyzing a draft standard for binary floating-point arithmetic, *IEEE Computer,* March 1981.

Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *IEEE Computer,* January 1980.

The Proposed IEEE Floating-Point Standard, special issue of the ACM *SIGNUM Newsletter,* October 1979.

The following defines the UNIX† System V Interface:

System V Interface Definition, Issue 2, AT&T, Indianapolis, IN, 1986.

The following provide more information about testing and error analysis:

Dongarra, *Performance of Various Computers Using Standard Linear Equations Software in a FORTRAN Environment,* Argonne National Laboratory, Argonne, IL, 1986.

Karpinski, "Paranoia: a Floating-Point Benchmark," *Byte,* February 1985.

Spafford, Flaspohler, *A Report on the Accuracy of some Floating Point Math Functions on Selected Computers,* Technical Report GIT-CS 85/06, Georgia Institute of Technology, Atlanta, 1985.

"Appendix: Accuracy of Numerical Calculations," in *HP-15C Advanced Functions Handbook,* 00015-90011, Hewlett-Packard, 1982.

Cody and Waite, *Software Manual for the Elementary Functions,* Prentice-Hall, 1980.

Bunch, Dongarra, Moler, Stewart, *Linpack Users' Guide,* SIAM, Philadelphia, 1979.

Curnow, Wichmann, "A Synthetic Benchmark," *The Computer Journal,* British Computer Society, London, February 1976.

Sterbenz, *Floating-Point Computation,* Prentice-Hall, 1974.

Test Program Information

For information on machine-readable source of the IEEE Test Vectors or the `SPICE` program, contact Cindy Manley, EECS/ERL Industrial Support Office, 461 Cory Hall, University of California, Berkeley, CA 94720.

For information on machine-readable source for the `Paranoia` test program, contact Richard Karpinski at *ucbvax!ucsfcgl!cca.ucsf!dick.*

For information on Alex Liu's test programs, contact him at *zliu@weyl.berkeley.edu.*

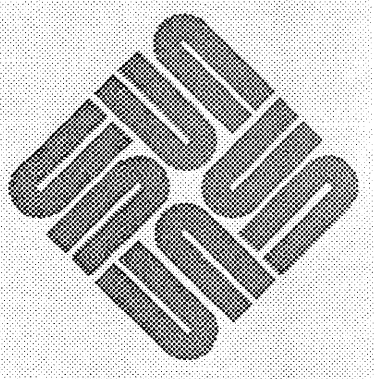
† UNIX is a trademark of AT&T Bell Laboratories.

Netlib provides a numerical source code distribution service from which many popular benchmark programs can be obtained, including `paranoia`, `elefunt`, `linpack`, and `whetstone`, as well as the current version of Dongarra's Argonne Technical Memorandum containing `Linpack` benchmark results. For netlib instructions, send a message "send netlib-paper from misc" to: *research!netlib* or *netlib@anl-mcs.arpa.*

For information on the FCVS tests, write to Federal Software Testing Center, Suite 1100, 5203 Leesburg Pike, Falls Church, VA 22041.

1

Sun Floating-Point Options



1

Sun Floating-Point Options

This manual describes floating-point support for Sun Microsystems' Sun-2 and Sun-3 systems. Sun's earlier models are not discussed here: The phrase "runs on any Sun" means that programs compiled for a Sun-2 run on any Sun-2 or Sun-3, while programs compiled for a Sun-3 run on any Sun-3. Throughout this manual, mention of "all" Sun systems refers to Sun-2's and Sun-3's.

1.1. Hardware Floating-Point Options

The floating-point hardware options usable with a Sun Microsystems computer depend on the model Sun being used, either a Sun-2 or Sun-3.

Sun-2 Systems

Sun-2 systems use a 10 MHz MC68010 as their CPU and use either a Multibus or VME system bus. The Sky Fast Floating-Point Processor (FFP) board exists in both Multibus and VME versions for the Sun-2. Programs compiled for Sun-2's will usually run on Sun-3's as long as they do not exploit any specific features unique to Sun-2's, such as the Sky FFP.

Sun-3 Systems

Sun-3 systems use either a 15 or 16.7 MHz MC68020 as their CPU and use a VME system bus. An MC68881 floating-point coprocessor is standard on some models and optional on others. Sun-3's use MC68881's of either the mask set A79J, running at 12.5 MHz, or of the mask set A93N, running at 16.7 MHz. Also available on some Sun-3 models is the Sun Floating-Point Accelerator (FPA), which is an optional board based upon the Weitek 1164/1165 chip set. The FPA requires the presence of an MC68881.

Programs compiled for Sun-3's will **not** run on Sun-2's.

1.2. Compiler Code Generation Options

Sun's compilers for FORTRAN (`f77`), C (`cc`), and Pascal (`pc`) use the options described below in the same way. With each of these compilers, if you compile and link programs in separate steps, you should usually specify the same set of options at each step. For example, for FORTRAN:

```
f77 -pg -O -ffpa -c part1.f
f77 -pg -O -ffpa -c part2.f
f77 -pg -O -ffpa part1.o part2.o -o executable.out
```

Note:

None of Sun's compilers attempt extensive code optimization *unless requested.*

Full Optimization (-O)

The -O option is the usual way to request maximum code optimization. In Release 3.2 f77, -O invokes /usr/lib/iropt to perform the following global optimizations on the program:

- Invariant code removal from loops
- Induction variable strength reduction
- Common subexpression elimination
- Copy propagation
- Register allocation
- Dead code elimination

Note that iropt's global optimizations sometimes cause conformance problems with the IEEE floating-point standard.

Floating-point variable register allocation is discussed below under the -fstore option.

In Release 3.2 cc and pc, -O invokes a more limited set of global optimizations. In all three compilers, -O also invokes /lib/c2 to perform local optimizations on the compiled code before assembly.

Partial Optimization (-P)

Partial optimization (-P) generates smaller data structures but often produces code almost as good as -O.

Partial optimization is called for in cases where -O results in inordinately long compilation times: some modules generate huge internal data structures in the global optimizer iropt that can lead to enormous compilation times with -O. If you think that is happening you can investigate this by printing out the compiler's phases with the -v option; if compilation seems to hang up in iropt, use -P instead of -O.

Don't use -P as a general substitute for -O since -O usually produces more efficient code.

Optimization Side Effects

iropt's global optimizations sometimes cause problems with conformance to the IEEE floating-point standard. Floating-point variable allocation to registers is discussed with the -fstore option. Dead code elimination, common subexpression elimination, and copy propagation may cause source code statements to not be executed when expected, and consequently IEEE exceptions may not be raised or may be raised a different number of times or in different places than the source code suggests.

For examples, see the subroutine nextd in the section "How to Use Special Features of the IEEE Standard" in Chapter 3, and the results of the Paranoia benchmark for the MC68881 in Chapter 4.

Inline Expansion

Inline expansion of external calls is an optional third phase of optimization. You can select it independently of other optimization choices by specifying one or more .il files along with the source files in a compilation.

The inline subroutine expansion program /usr/lib/inline replaces subprocedure calls with equivalent code inline in the calling procedure, using

templates contained in the `.il` files. User-defined `.il` files may be used in addition to or in place of Sun-provided standard `.il` files. See Appendix G for a discussion on using inline subroutine expansion.

Floating-Point Code Generation Options

Sun supports two approaches to floating-point code generation: switched and in-line. Switched code determines at runtime what the fastest available floating-point hardware is and selects from among available library routines to exploit that hardware. Thus switched code runs on any Sun, with any or no floating-point hardware, and takes advantage of whatever hardware is present. The results of floating-point computations may vary depending on the hardware that was used. Select switched floating-point code generation at compile time with the compiler option `-fswitch`.

Select software floating-point code generation with the compiler option `-fsoft`. Code so compiled runs on any Sun and always produces the same numerical result. Code compiled with `-fsoft` contains calls to floating-point libraries and is about as fast as switched floating point on machines without floating point hardware.

In-line code generation puts floating-point instructions in line with CPU instructions; such code generated in-line for specific hardware, by avoiding the overhead of library calls, *always* runs faster than switched code running on the same hardware. Remember that in-line code only runs on machines with the specific hardware installed for which the code was compiled.

The available floating-point code generation hardware options are:

- `-fsky` Requires Sky FFP on Sun-2.
- `-f68881` Requires MC68881 on Sun-3.
- `-ffpa` Requires MC68881 and Sun FPA on Sun-3.

Default Floating-Point Code Generation

`-fsoft` is the default floating-Point code generation mode, in the absence of other specifications. This was chosen so that programs that use floating point in a casual way will obtain the same numerical results on all Suns.

You may change your default by defining the environment variable `FLOAT_OPTION` with a `.cshrc` or `.login` command like

```
setenv FLOAT_OPTION f68881
```

Other legal choices are `fswitch`, `fsoft`, `fsky`, or `ffpa`.

Bear in mind when converting makefiles and shell scripts, that Sun releases before 3.0 made switched floating-Point the default, with `-fsky` the only option.

Mixing Floating-Point Code Generation Styles

An executable program composed of several independently compiled modules can contain only one kind of in-line hardware floating-point. If this restriction is violated, one of the following error messages

```
Undefined: fsky_used
Undefined: f68881_used
Undefined: ffpa_used
```

will be issued by the compiler when it tries to link the modules together.

Although the practice is not recommended, you can link together modules compiled with `-fsoft`, `-fswitch`, and any one type of in-line hardware floating point.

The compilers defeat attempts to specify `-fsky` on a Sun-3 or `-ffpa` on a Sun-2.

Libraries and Executable File Sizes

For small and moderate sized programs, inline code generation creates the smallest executable files because the fewest library subroutines are required. `-fsoft` requires library subroutines for all operations, while switched floating-point (`-fswitch`) requires library subroutines for software floating-point and all the inline options and so is the largest of all. There are, however, some code generation technicalities that affect these generalizations:

Each compiler searches certain libraries by default:

Libraries Searched

Compiler	Normal	Profiling (-p or -pg)
cc	c	c_p
f77	F77 I77 U77 m c	F77_p I77_p U77_p m_p c_p
pc	pc m c	pc_p m_p c_p

In this table `m`, for instance, is shorthand for option `-lm` or the equivalent file name `/usr/lib/libm.a`, except for `c`, which corresponds to `-lc` or `/usr/lib/libc.a`. These libraries contain subroutines which are in turn called by code generated by the compilers. The compilers may generate code to call different subroutines, depending on the `-f...` option in effect at compiler time.

In general, `libm.a` and `libc.a` contain different subroutines corresponding to each `-f...` option. `libpc.a`, `libF77.a`, `libI77.a`, and `libU77.a` do not; the subroutines in these libraries are therefore created with `-fswitch`, and so if they are called, then much of switched floating point is linked into the final executable file.

This means that the executable file may be larger than expected. In some cases the switched floating point may be bypassed by using the inline subroutine expansion files described in Appendix G.

FORTRAN source-language constructs that generate calls to subroutines in `libF77.a` include, for real or double precision, `x**y`, `mod`, `atan2`, `cabs`, and `nint`; for complex or doublecomplex, almost all operators and functions. With the Sun-provided inline subroutine expansion files, switched floating-point subroutines are not linked in except for for complex and doublecomplex `x/y`,

`x**y`, `sqrt`, `exp`, `log`, `sin`, and `cos`.

Summary: Which Option When?

Compile programs with `-fsoft` that use floating point in a casual way or not at all; this is the usual default anyway. Compile programs on a Sun-2 with `-fswitch` that use floating point intensively but that are intended to be run on all possible Sun-2 and Sun-3 hardware configurations. Compile programs that are intended to use particular floating-point hardware intensively with in-line code for that hardware.

Remember that the Sun FPA can only support 32 simultaneous processes. Consequently, on systems with a Sun FPA, `-ffpa` and `-fswitch` should not be used for programs that use floating point casually, reserving floating-point contexts for intensive floating-point programs.

Programs that exploit certain details of the IEEE Standard may require `-fsoft`, `-f68881`, or `-ffpa`; see Chapter 3.

1.3. Expression Evaluation Options

Some languages prescribe precise evaluation rules for expressions like these:

```
a + b
c * (d + e)
```

Others do not. Different kinds of floating-point hardware also lend themselves to different methods.

FORTRAN Style

The traditional method of expression evaluation in FORTRAN is that operations are performed in the precision of the most precise operand. Thus, `a + b` is evaluated in single precision if `a` and `b` are both single-precision, and in double precision if either `a` or `b` is double-precision.

Greatest Available Precision Style

Since floating-point expressions usually suffer rounding errors when evaluated, these errors may be minimized if expressions are evaluated in the highest available precision. Thus many C compilers evaluate `a + b` in double precision even if `a` and `b` are both single-precision.

Sun's FORTRAN and Pascal compilers generally perform FORTRAN style expression evaluation. An exception is when the `-f68881` option is specified; then expressions are evaluated in the extended-precision registers of the MC68881. Sun's C compiler generally performs double-precision expression evaluation; if `-f68881` is specified, evaluation is in extended precision. The `-fstore`, `-fsingle`, and `-fsingle2` options, described below, provide alternate methods.

`-fstore`

When the `-f68881` option is used in conjunction with `-O` or `-P`, the compilers allocate floating-point variables to the MC68881's internal extended-precision registers, which are more precise than the single- or double-precision variables declared in FORTRAN, C, or Pascal. This means that, for instance,

```
X = expression
if (X .eq. 1.0) ...
```

may execute differently depending on whether X is allocated to an extended-precision register. If not, then the value of "expression" is rounded to the declared precision of X prior to the comparison, possibly changing its value; if X were so allocated, the rounding would not occur, affecting the subsequent comparison.

This optimization may adversely affect some programs that depend on the store to force rounding to storage precision. Those programs can be compiled without optimization, or with optimization and the additional option `-fstore`, which insures that rounding to storage precision occurs when specified by a source language assignment.

`-fstore` is implemented in Release 3.2 by not allocating floating-point variables to extended-precision registers. `-fstore` only has effect when `-f68881` and `-O` or `-P` are specified.

The combination `-fstore -O` is safer than `-O` in that a few programs either won't work or work poorly with just `-O`. `-O` is usually slightly more accurate; the difference in speed varies considerably among programs.

The `-fstore` option is not intended to affect anonymous temporary expressions. Thus the statement

```
x = (x + y) - y
```

may still be evaluated in extended precision until the final store to `x`, even if `-fstore` is specified. This may cause problems with some artful programs, such as some in the `Elefunt` test suite of Cody and Waite. The `Elefunt` programs test elementary functions by computing various identities involving those functions; inaccuracies in the elementary function implementations are reflected by the extent to which the identities fail to be satisfied. To be meaningful, the arguments of the functions must be correctly related. That is, if `x` and `x+k` are both arguments to functions in the identity, then `x` must be such that `x+k` is computed exactly. Any error in `x+k` prior to evaluating the function under test will likely contaminate the results to such an extent that they are meaningless for measuring the errors in the function under test. `x` is *purified* by perturbing it slightly until `x+k` can be computed exactly.

A specific example in the `Elefunt` test of `alog` is an identity wherein `x` and `y=(17/16)*x` are both required. To compute `y` exactly on binary, decimal, and hexadecimal machines, over the limited range `sqrt(1/2) <= x <= 15/16`, it suffices that `x`'s least significant four bits be zero; this is accomplished by the following code:

```
eight = 8.0
x = (x + eight) - eight
y = x + x / 16.0
```

This code works correctly unless the expression `(x + eight) - eight` is evaluated in higher precision than the storage precision of `x`. If the latter occurs, then the rounding of `(x + eight)` will not occur where intended and `x` will

not be purified. Since it is the intermediate expression `(x + eight)` that is of interest, `-fstore` will not affect the result. You must change the source code to

```
eight = 8.0
x = x + eight
x = x - eight
y = x + x / 16.0
```

Then `-fstore` will force rounding to storage precision after `(x + eight)`, and `y` will be ultimately computed without any rounding error.

Extended-precision expression evaluation is usually desirable because it minimizes rounding error, and it is good practice to write out expressions with minimal stores of intermediate results. But programs which intend to force rounding errors to occur at certain times must sometimes be rewritten in this way when using extended precision.

`-fsingle` and `-fsingle2`

In C, floating-point expression evaluation and parameter passing are defined to occur in double precision, even if all operands are single-precision. `cc`'s `-fsingle` and `-fsingle2` options permit more FORTRAN -like floating-point expression evaluation.

`-fsingle` causes floating-point operations to be performed in single precision when operands are single-precision. `-fsingle2` causes single-precision procedure parameters to be passed as single-precision rather than double-precision. Do not mix C modules compiled with `-fsingle2` with modules compiled without it.

In particular, the following will not work if compiled with `-fsingle2`:

```
#include <math.h>

float x,y;
y = sqrt(x);
```

Standard C libraries expect double-precision floating-point arguments.

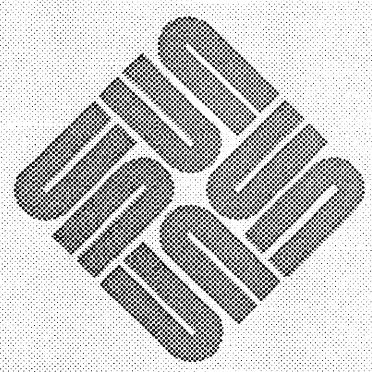
Constant Expressions

Constant expressions may be evaluated at compile time or at run time. From the point of view of the IEEE Standard, run-time evaluation of constant expressions is desirable in order to observe the rounding modes in effect at runtime and to preserve the IEEE exceptions.

Sun's FORTRAN compiler (`f77`) doesn't evaluate floating-point constant expressions at runtime, while `cc` and `pc` both do, using the default round to nearest mode. All three compilers evaluate integer constant expressions at runtime.

2

Floating-Point Numerics



2

Floating-Point Numerics

2.1. Rounding Errors and Different Numerical Results

Rounding Errors

Expression evaluation and precision are important because floating-point calculations are usually inexact. The set of numbers representable in a particular finite-precision format is not closed under the usual arithmetic operations. When the computed result differs from the exact infinite-precision result, we say that roundoff or a rounding error has occurred. Correctly implemented floating point delivers the exact result if it is representable, or one of at most two nearest representable numbers otherwise. Computers vary as to the rule used for choosing between the two nearest neighbors; some computers with defective arithmetic do not guarantee to produce representable results correctly or to produce one of the nearest neighbors for unrepresentable results.

Sun's computers generally follow the IEEE Standard for Binary Floating-Point Arithmetic, which specifies that, by default, inexact results return the closer of the two nearest neighbors, and if they are equally close, the one with the least significant bit 0. This rule is the default "round to nearest" rounding direction mode. Rounding errors caused by the correct application of the rounding rule to inexact results are inevitable consequences of finite-precision arithmetic, rather than defects in the machine.

Different Numerical Results

Different floating-point implementations produce different results. Even when all floating-point operations conform to the IEEE Standard, elementary transcendental functions will usually vary depending on the hardware. Unlike the rational operations (`+`, `-`, `*`, and `/`) and the algebraic function `sqrt`, it is not economical to produce correctly rounded results for the elementary exponential, logarithmic, and trigonometric transcendental functions. Consequently different implementations make different trade-offs between speed and accuracy.

Differences arise even if no transcendental functions are involved. The MC68881 and Sun FPA both conform to the IEEE Standard, yet they produce different results because the MC68881 uses extended-precision registers, which are not present on the FPA. The MC68881's results are usually more accurate. The MC68881's results can vary depending on which of the `-O`, `-P`, or `-fstore` flags are used, since these affect how often results are rounded from extended precision to storage precision. Finally, results computed with the A79J

mask set MC68881 may differ from those computed with the later versions of the MC68881.

Sun's software floating-point conforms to the IEEE Standard's default modes and agrees with the Sun FPA on operations specified by the Standard in the default modes.

Due to performance requirements and the fact that its microcode space is completely full, the Sky FFP does not always produce the results specified in the IEEE Standard, even for the rational operations `+`, `-`, `*`, and `/`. For these operations and `sqrt`, computed results sometimes differ by one bit from the standard results; at other times the signs of zero results vary from the Standard.

If you specify `cc`'s `-fsingle` or `-fsingle2` options, results may be different from what they would have been otherwise.

Detecting Ill Condition and Instability

When the answer to a problem is inordinately sensitive to changes in the input data, it is said to be **ill-conditioned.** Ill condition is a property of a problem and input data and is not affected by changing the algorithm used to obtain the result.

Even for well-conditioned problems, some numerical algorithms are much more sensitive to roundoff than others. The sensitive algorithms are said to be **unstable.** Instability can sometimes be cured by changing algorithms.

The distinction between ill-condition and instability is often academic in large realistic applications too complicated to be analyzed. The symptoms are the same: a computation previously believed to be correct on another machine, or with a different compiler or operating system, or even with different compiler options, now gives drastically different results with the same input data. Naturally, such results suggest that the new hardware, compiler, or operating system is defective. How can that possibility be separated from the effects of ill condition or instability?

One way to investigate sensitive results is to introduce random small changes in the input data and observe the effect on the results. But the right random changes may be difficult to determine or awkward to insert. The four rounding direction modes specified in the IEEE Standard allow one to obtain similar effects.

On Sun-3's with an MC68881 installed, you can vary the IEEE rounding modes with the assembly-language function, `_fpmode_`, mentioned in "How to Use Special Features of the IEEE Standard" in Chapter 3. Run a program first in the default round-to-nearest mode, then in another mode such as round-toward-zero. Compare the results, and if numerical results vary enough to bother you, then you are either trying to solve an ill-conditioned problem or using an unstable algorithm.

2.2. An Example - `SPICE tdo`

`SPICE` is an integrated circuit emulation program. One of the `SPICE` 2G.6 input files listed in Appendix E, `tdo`, models a tunnel diode oscillator. In the transient analysis section of the program's output, a voltage `v1` is monitored as a function of time. The computed behavior is that the voltage is initially a constant 0.12 v, then either declines abruptly to about 0.02 v or else jumps equally abruptly to about 0.48 v. Which behavior is computed seems to depend on random factors of code generation causing slight variations in rounding errors. For

instance, the abrupt decline is obtained with `-fsoft -O` and `-ffpa -O`, while the abrupt rise is obtained with `-fsky -O` and `-f68881 -O`, in all cases compiling with software Release 3.1 and default roundings. Experiments with `-f68881 -O` show that changing the rounding direction mode from round to nearest to round toward zero, or changing the rounding precision mode from extended to double, is sufficient to change the computed behavior. Since oscillators usually exhibit positive-feedback physical characteristics, it seems likely that computing the transient behavior is an inherently ill-conditioned problem. On the other hand, determining the steady-state oscillation frequency might be well-conditioned.

2.3. Floating-Point Programs

Sun recommends that you write programs using floating-point in higher-level languages whenever possible.

FORTRAN is the best-suited compiled language that Sun supports for those parts of large applications where floating-point usage is substantial, and data structures, control structures, and input and output are simple. FORTRAN has more intrinsic types, operators, and functions designed to support floating-point computations and ease writing correct, compact, easily optimized numerical procedures.

Many other desirable language features are poorly designed in FORTRAN or lacking altogether, so Sun makes it relatively easy to call FORTRAN computational modules from programs written in C or Pascal.

In Pascal, you just declare such modules to be "`external fortran`".

In C, remember that a FORTRAN declaration like

```
xtype function x ( y )
ytype y
```

usually corresponds to a C declaration like

```
xtype  x_ (y)
ytype *y ;
```

For more about programming on Sun's computers in FORTRAN and Pascal, see the Sun *FORTRAN Programmer's Guide* and the Sun *Pascal Programmer's Guide.*

Assembly Language

The assembly language interfaces to `-fswitch`, `-fsoft`, and `-fsky` floating point are not documented and vary in different software releases. If necessary, you can examine compiled code generated by the `-S` option to infer the interface conventions in effect for a particular release.

For `-f68881` and `-ffpa`, the interface is more permanent since specific features have been added to the Sun-3 assembler to accommodate these floating-point units. The following lines of code are comparable:

Source:

```
x(i) = x(i) + c * y(i)
```

MC68881 assembly code:

```
fmoved  c,fp0
fmuld   ay@+,fp0
faddd   ax@,fp0
fmoved  fp0,ax@+
```

FPA code:

```
fpmoved c,fpa0
fpmuld  ay@+,fpa0
fpaddd  ax@,fpa0
fpmoved fpa0,ax@+
```

In both examples, `ax` and `ay` are pointers to `x(i)` and `y(i)`, respectively.

The syntax of MC68881 instructions generally follows that of Motorola's MC68881 manual, with the following exceptions:

- Instructions are lower case
- No dot separates the opcode from the operand type
- CPU address mode notations are different

Furthermore, unlike examples in the MC68881 manual, dyadic operations

```
fopt    fpn,fpn
```

may not be abbreviated to the single-operand form

```
fopt    fpn
```

FPA instruction syntax is designed to parallel that of MC68881 instructions where it makes sense to do so. Op codes begin with "`fp`" rather than "`f`" and register names are `fpa0..fpa31` rather than `fp0..fp7`. Register-to-register operations have type `s` or `d` rather than `x`.

2.4. Precision in Decimal Digits

The relationship between binary floating point and decimal floating point is one of the most widely misunderstood aspects of floating-point arithmetic. No binary floating-point format has an exactly equivalent number of decimal digits, since the relative spacing of representable floating-point numbers varies in different places for different bases. The following describe the IEEE formats:

Single Precision

IEEE single precision is always *more* precise than 6 decimal digits and always *less* precise than 9 decimal digits. Seven decimal digits are often less precise but sometimes more; eight decimal digits are often more precise but sometimes less.

Double Precision

IEEE double precision is always *more* precise than 15 decimal digits and always *less* precise than 17 decimal digits. Sixteen decimal digits are sometimes less precise and sometimes more.

Extended Precision

MC68881 extended precision is always *more* precise than 18 decimal digits and always *less* precise than 21 decimal digits. Nineteen decimal digits are often less precise but sometimes more; 20 decimal digits are often more precise but sometimes less.

2.5. FCVS Tests

The following examples illustrate the confusion surrounding precision issues. The Federal Compiler Validation Service (FCVS) has a series of programs intended to measure conformance of implementations to the FORTRAN standard, ANSI X3.9-1978. While investigating why certain test programs failed on Suns in mysterious ways that depended on the setting of the `-f...` and `-O` options, it became apparent that the programs were setting standards that could only be passed by luck. Here are two examples taken from the FCVS78 Version 2.0; official FCVS versions since 1983 have been modified to display the message "INFORMATIVE - UNDER REVIEW BY FSTC" for certain tests in programs 371..374, 818, and 820.

The following fragment is from test 10 of FCVS program 374, intended to test single-precision tangent:

```
        PIVS = 3.14159265358979323846264338
        AVS = TAN((3.0 * PIVS / 2.0) - 1.0 / 1024.0)
        IF (AVS - 1023.9) fail, pass, 40100
40100   IF (AVS - 1024.1) pass, pass, fail
```

This fragment is intended to be a test of tan, to see if

```
AVS = TAN(1.5 * π - 2**-10) = 1/TAN(2**-10)
```

which is approximately 1024 - .0003. So the passing criterion seems rather generous:

```
    if (abs(avs-1024) < 0.1) then pass.
```

But whether this test passes or fails is a matter of random luck for single precision. Suppose that the only error is the error in the value assigned to `PIVS`; it was intended to be π, but since π is not representable, the actual value contained

in PIVS is not π but π+ε. Conventional error analysis shows that the passing criterion is equivalent to:

```
abs(ε) < (2/3) * (0.1) * 2**-20 = 6.4e-8.
```

But binary floating-point arithmetic of *p* significant bits cannot in general approximate a number to better than half a unit in the last place, or 2**(1-*p*) for numbers between 2 and 4; IEEE single precision has 24 significant bits, so arbitrary numbers between 2 and 4 would not be expected to be approximated to better than ε = 1.2e-7, and in particular, the IEEE single-precision number closest to π differs from π by ε = 8.7e-8. Thus there is no hope of passing unless *additional* rounding errors are made in the multiplication or tangent; these additional rounding errors must be aptly chosen to cause the final result to be acceptable!

The following fragment is from test 12 of FCVS program 820, intended to test complex cosine:

```
      COMPLEX AVC, BVC
      REAL R2E(2)
      EQUIVALENCE (AVC, R2E)

      AVC = CCOS(( 3.1416, 0.0) * (-10000.0, 0.0))
      IF (R2E(1) - 0.99725E+00) fail, 40122, 40121
40121 IF (R2E(1) - 0.99736E+00) 40122, 40122, fail
40122 IF (R2E(2) + 0.50000E-04) fail, pass, 40120
40120 IF (R2E(2) - 0.50000E-04) pass, pass, fail
```

This is intended to test whether

```
ABS(COS(1e4 * 3.1416)-.997305) < 5.5e-5
```

The correct result cos(31416.0) is .9973027. But once again, passing is a matter of random luck on binary computers. Suppose the only error is the conversion of 3.1416 to a machine-representable value 3.1416+ε. Error analysis shows that the passing criterion is equivalent to

```
-8.0e-8 < ε < 7.0e-8
```

but as before, the best general bound for the approximation error is 1.2e-7, and in particular the closest IEEE single-precision number to 3.1416 differs from it by 1.1e-7. Machines with 24 significant bits or less can only pass the indicated criterion by lucky rounding in the multiplication and cosine.

Oddly enough, this same test program considers acceptable a nonzero imaginary part of a complex cosine of a real argument.

2.6. Floating-Point and Signal Handlers

UNIX signals are asynchronous events, and when a signal handler uses the same floating-point device as the process it interrupted, confusion can ensue. Consequently part of Sun's signal handling protocol is designed to save and restore the state of the floating-point device currently in use, but it is good practice to design signal handling routines to be as short as possible and to avoid inessential floating-point computations.

Signals, once generated, are passed by UNIX to the interrupted process through a routine named `_sigtramp` which calls a signal handler defined by the user with `sigvec(2)`, `signal(2)`, or `signal(3f)`.

Before calling the user handler, `_sigtramp` saves certain CPU and floating-point state on the stack; if the signal handler terminates normally by returning to `_sigtramp`, the CPU and floating-point state are restored from the saved state. The CPU state includes the contents of the program counter (pc), stack pointer, status register, signal stack and mask information, and registers `a0-a1` and `d0-d1`. The floating-point state depends on which signal occurred and what kind of floating-point hardware is in use. In the case of signal #8, `SIGFPE`, no floating-point state is saved or restored; for all other signals, the hardware state of the Sky FFP, MC68881, or Sun FPA is saved when the signal occurs and restored when the signal handler returns.

Note that just as the signal handler is expected to follow the C compiler's conventions and restore CPU registers `a2-a7` and `d2-d7` if it uses them, it is also expected to restore certain floating-point registers, as described below.

In the case of the Sky FFP, the complete state is saved, consisting of the contents of the program counter of the Sky FFP and the contents of the four double-precision floating-point data registers. Before starting the signal handler, the Sky FFP is reset to accept new instructions. The contents of the data registers are unchanged. On return from the signal handler, the Sky FFP program counter and data registers are restored to their state prior to interruption.

In the case of the MC68881, the registers saved and restored include `fp0-fp1` and `fpcr/fpsr/fpiar`. The signal handler is responsible for saving and restoring `fp2-fp7` if it uses them.

In the case of the Sun FPA, the registers saved and restored include the MC68881's `fp0-fp1` and `fpcr/fpsr/fpiar` and the FPA's instruction pipe, `fpastatus`, `fpamode`, `imask`, `load_ptr`, `ierr`, and `fpa0-fpa3`. Before calling the signal handler, `imask` is set to 0, `fpamode` is set to 2 (the default), and `fpsr` and `fpcr` are set to 0. The signal handler is responsible for saving and restoring `fpa4-fpa31` if used. Upon returning from the signal handler, all the control and data registers are restored to their values at the time of the signal.

The exact format of the saved floating-point state is subject to change and should not be relied upon. Programs that use nonstandard returns from signal handlers may not work under new software releases.

`setjmp/longjmp`

The code implementing `setjmp` and `longjmp` does not save or restore any floating-point state. Use great care in creating programs that involve these functions and floating-point arithmetic.

Floating-Point Signals

Two floating-point signals, `SIGEMT` and `SIGFPE`, are used by software to test for the presence of an MC68881 and as signals for some error or exception conditions.

Signal #7 (`SIGEMT`)

Programs compiled with `-fswitch`, `-f68881`, and `-ffpa` test for the presence of an MC68881 by performing an MC68881 instruction that initializes the `fpcr` and `fpsr` registers to their proper IEEE defaults. If an MC68881 is present, the instruction executes uneventfully. If no MC68881 is present, a `SIGEMT` signal is generated and handled by a signal handler installed for the duration of this test. The `SIGEMT` signal also affects use of the debuggers `adb`, `dbx` and `dbxtool`; see "Debuggers and Floating Point" below.

Signal #8 (`SIGFPE`)

`SIGFPE` signals certain conditions that might be errors, exceptions, or opportunities. Generally these conditions arise synchronously during arithmetic operations in the underlying process, but they are infrequent enough that they are most efficiently treated as if they were asynchronous events. Since they usually tell something about the state of the floating-point computation, the floating-point status is *not* saved, reset, and restored as it is for all other signals.

Not all `SIGFPE` signals have to do with floating point. This is a complete list of defined `SIGFPE` codes:

```
#define     FPE_INTDIV_TRAP     0x14    /* integer divide by zero */
#define     FPE_CHKINST_TRAP    0x18    /* CHK [CHK2] instruction */
#define     FPE_TRAPV_TRAP      0x1c    /* TRAPV [cpTRAPcc TRAPcc] instr */
#define     FPE_FLTBSUN_TRAP    0xc0    /* [floating branch | set on unordered ] */
#define     FPE_FLTINEX_TRAP    0xc4    /* [floating inexact result] */
#define     FPE_FLTDIV_TRAP     0xc8    /* [floating divide by zero] */
#define     FPE_FLTUND_TRAP     0xcc    /* [floating underflow] */
#define     FPE_FLTOPERR_TRAP   0xd0    /* [floating operand error] */
#define     FPE_FLTOVF_TRAP     0xd4    /* [floating overflow] */
#define     FPE_FLTNAN_TRAP     0xd8    /* [floating Not-A-Number] */
#define     FPE_FPA_ENABLE      0x400   /* [FPA enable reg not set] */
#define     FPE_FPA_ERROR       0x404   /* [FPA bus error] */
```

Of these, `INTDIV`, `CHKINST`, and `TRAPV` are related to integer arithmetic or non-arithmetic instructions. `FLTBSUN...FLTNAN` are generated by the MC68881 and can be indirectly generated by the FPA. `FPA_ENABLE` and `FPA_ERROR` can only be generated by the FPA.

The `FPE_FPA_ERROR` code is used by the FPA to indicate that it cannot perform a particular operation; this may be because the operation is not a valid FPA instruction or the data is outside the domain accepted by the Weitek chips or the FPA microcode. The latter exceptions are part of normal FPA operation.

`SIGFPE` must be enabled and assigned to a signal handler that recognizes the `FPE_FPA_ERROR` code and handles it properly when an FPA is to be used. (This is done automatically by `-ffpa`). If `SIGFPE` is subsequently totally disabled, an `FPE_FPA_ERROR` signal will cause an infinite loop because the

FPA is never reset and cannot progress past the troublesome instruction.

Floating-point `SIGFPE` codes do not arise for programs compiled with `-fsoft` or `-fsky`. They do not arise for programs compiled with `-f68881` unless IEEE traps are enabled on the MC68881 by setting bits in the Exception Enabled byte of the fpcr register. The MC68881 exception enabled bits also affect FPA results during recomputation; see "FPA Recomputation" below.

Why Not Signal by Default?

Users new to the IEEE Standard often wonder why seemingly catastrophic events such as division by zero or overflow do not cause immediate error termination of the program. They have been conditioned to expect the latter by various systems of arithmetic that were unable to provide any reasonable response for exceptional conditions.

The IEEE Standard, however, provides default responses for floating-point operations that are meaningful and adequate for many purposes: Infinities and NaNs are the usual results of overflow and division by zero. Sun's floating point meets the standard by silently providing the correct default result. This is usually satisfactory when the result of an operation is a floating-point quantity; other cases, sometimes troublesome, include comparisons, conversion to integer format, and conversion from binary to ASCII. Programmers who haven't considered the possibility of floating-point operands that might be infinities or NaNs sometimes get surprising results in these cases.

When an MC68881 is available, the `SIGFPE` signals `FPE_FLTBSUN_TRAP` through `FPE_FLTNAN_TRAP` can be enabled to occur when specified floating-point exceptions arise. For instance, to cause `FPE_FLTOPERR_TRAP` to be signalled whenever an invalid operation occurs such as converting a NaN, infinity, or too-large value to an integer, it is necessary to set the `OPERR` bit in the Exception Enable Byte in the MC68881's `fpcr` register, described in Motorola's *MC68881 User's Manual.* This can be done either in assembly language, or by using the `_fpmode_` subroutine described in the section "How to Use Special Features of the IEEE Standard" in Chapter 3.

2.7. Debuggers and Floating Point

Floating-point hardware or software use will affect the way that you work with debuggers on Sun systems. Signal handling and disassembly operations are particularly affected. For more information about debuggers and floating-point, see Appendices B and C in this manual.

Signals

When using the debuggers `adb`, `dbx`, and `dbxtool`, be aware that certain signals occur in the normal course of floating-point operations. As mentioned above, a MC68881 instruction is attempted to determine if an MC68881 is present. If no MC68881 is present, `SIGEMT` is signalled and is caught by the debuggers, stopping the process being debugged. You can then either continue or, if you prefer, instruct the debugger in advance to ignore the signal by issuing the command `7:i` in `adb` or `ignore 7` in `dbx`. This command passes the signal directly to the debugged process, bypassing the debugger.

When an FPA is installed, `SIGFPE` is signalled each time FPA recomputation is required, as discussed below. When these signals occur, you may either continue, or issue `8:i` or `ignore 8` to ignore them. Almost every process that uses the FPA requires recomputation for at least the first inexact result.

Disassembly

MC68881 instructions can always be disassembled using MC68881 assembly language mnemonics. FPA instructions, however, are simply `movel` instructions to particular addresses, and consequently cannot be positively distinguished from other `movel` instructions. To allow control over this disassembly, `adb` and `dbx` provide ways to specify whether or not `movel` instructions should be considered as potential FPA instructions.

Sun's compilers generate FPA instructions using `movel`'s with "long absolute" and "address register with displacement" effective address modes. For example:

```
fpadds          <eadata1>,fpa0
fpmoves@1       fpa0,<eadata2>
```

generates exactly the same code as

```
movel           <eadata1>,0xe0000380
movel           a1@(0xc00),<eadata2>
```

The debuggers can recognize two patterns of `move` instructions as FPA instructions. First, instructions that access the FPA using absolute addressing may be recognized simply by examining the address; second, instructions that access the FPA using base + 16-bit displacement addressing may be recognized if the base register used has been designated as a pointer to the FPA's virtual address.

Note that you can make this designation at runtime, independent of the actual register contents. The default is to recognize absolute-mode FPA instructions only if an FPA is present. Base-mode FPA instructions are assumed to be moves by default, since no register is assumed to be an FPA base register.

FPA disassembly is enabled in `adb` by the command `1>F` and disabled with `0>F`. `1>B` designates `a1` as an FPA base register, while `-1>B` indicates that no register is being used as an FPA base register. The first example above would be disassembled as an FPA instruction if `F` were 1; the second if `F` were 1 and `B` were 1. Otherwise they would be disassembled as `movel`'s.

In `dbx`,

```
dbxenv fpaasm on
```

enables FPA disassembly, while

```
dbxenv fpaasm off
```

disables it. An FPA base register may be specified using

```
dbxenv fpabase <register>
```

while

```
dbxenv fpabase off
```

indicates that *no* register is assumed to be an FPA base register.

Printing all MC68881 Registers

In adb, $R displays MC68881 registers fp0-fp7 in hexadecimal (hex) and extended-precision decimal, and fpcr, fpsr, and fpiar in hex.

In dbx, the command &$fp0/nE displays n MC68881 registers starting with fp0, in hex and extended-precision decimal.

Printing all FPA Registers

In adb, $x displays FPA registers fpa0-fpa15 in hexadecimal, single-precision decimal, and double-precision decimal. It displays fpamode, fpastatus, state, imask, ldptr, ierr, and instruction pipe registers in hex. $X displays FPA registers fpa16-fpa31 in hex, single decimal, and double decimal, and fpamode, fpastatus, state, imask, ldptr, ierr, and instruction pipe registers in hex.

In dbx, the command &$fpa0/nF displays n FPA registers starting with fpa0, in hex and double-precision decimal. To display FPA registers in hex and single-precision decimal, use &$fpa0s/nf.

Printing in Single-Precision

In adb, a 4-byte memory location, fpa*n*, or a register dn can be displayed in single-precision decimal; use addr/f or <fpan=f or <dn=f in adb.

In dbx, use addr/f to print a single location in hexadecimal and single-precision decimal, addr/nf to print n locations. The notation &$regname can be used as a handle for printing registers; note that registers do not actually have addresses.

Printing in Double-Precision

In adb, an 8-byte memory location, fpan, or a register pair dn:dn+1 can be displayed in double-precision decimal; use addr/F or <fpan=F or <dn=F.

In dbx, use addr/F to print a memory location or register in hexadecimal and double-precision decimal, addr/nF to print n registers or memory locations.

Printing in Extended-Precision

A 12-byte memory location or fp*n* can be displayed in extended-precision decimal: In adb, use addr/e or <fpn=e.

In dbx, use addr/E or &$fpn/E.

Modifying Floating-Point Registers or Memory Locations

In adb you cannot load memory or registers with data in an ASCII floating-point representation. Hexadecimal data can be used to load one 32-bit word at a time. The words of a register can be designated separately, to make loading MC68881 and FPA registers easier. Thus the most significant, middle, and least significant words of fp1 can be referred to as fp1, fq1, and fr1. The more significant and less significant words of the FPA register fpa1 can be referred to as fpa1 and fqa1.

In dbx you can set a register or memory location with a value expressed in floating-point decimal notation.

dbx converts data into extended-precision format for an MC68881 register, and into double- or single-precision format for an FPA register, depending on whether a name in $fpa0..$fpa0 or $fpa0s..$fpa31s is used. The

data is converted according to the declared type of the variable for variables defined in a compiled language. `dbx` also provides the command

```
set 81 <fpreg> = xyz
```

to set an MC68881 register to an arbitrary 3-word bit pattern, formed by concatenating the 32-bit values *x*, *y* and *z*. No such command exists for FPA registers in Release 3.2.

2.8. FPA Recomputation

The Sun FPA relies on the Sun-3 CPU's MC68881 to perform many operations that would be difficult or inefficient to perform with the Weitek 1164/1165 chip set. These operations include any in which IEEE arithmetic exceptions arise. When one of these exceptions is detected in the Weitek chips, the operation is terminated and a bus error notifies the CPU that recomputation on the MC68881 is required.

The operating system intercepts the bus error, diagnoses it as coming from the FPA, and sends a `SIGFPE` to the process in question. That process is responsible for having a correct `SIGFPE` signal handler installed that determines if the `SIGFPE` is due to an FPA request for recomputation and handles it accordingly. When `-ffpa` is requested, such a signal handler is installed by default; the following is a simplified version of that default handler:

```
#include <signal.h>

int
sigfpe_handler(sig, code, scp)
    int             sig, code;
    struct sigcontext *scp;

{

    if (code == FPE_FPA_ERROR) {
        if (fpa_handler(scp))
            return;
        }
    } else {
        fprintf(stderr, "SIGFPE Handler pc %X code %X \n", scp->sc_pc, code);
        fflush(stderr);
    }
    abort();
}
```

`fpa_handler()` is the standard FPA recomputation routine contained in `libc`. It returns 1 if the FPA error was fixed by recomputation and 0 otherwise, as in the cases that the FPA error was caused by hardware failure or incorrect software. `fpa_handler()` is used to support the IEEE Standard arithmetic features pertaining to rounding modes, exceptions, and traps.

`fpa_handler()` uses the MC68881 to recompute the desired result. The MC68881's single and double rounding precision modes are used to obtain the results anticipated by the IEEE Standard for simple single- and double-precision operations. More complex instructions, such as elementary transcendental functions and combination instructions, such as add followed by multiply or multiply followed by add, are recomputed in extended rounding precision mode to obtain maximum accuracy.

Rounding Modes

Rounding modes are supported directly by the Weitek 1164/1165 for simple instructions; the elementary transcendental functions, however, are microcoded. When an elementary transcendental function is called with a rounding mode other than the default (round to nearest) recomputation is invoked in order to compute the function on the MC68881.

Exceptions

Exceptions are reported by the Weitek chips after each operation, but those chips do not accumulate exceptions or allow optional trapping on them. Consequently, whenever the Weitek chips report one of the exceptions other than inexact, the FPA hardware signals that recomputation is required. The CPU receives that signal when it attempts to perform a subsequent FPA instruction; the CPU's program counter points to the subsequent instruction rather than to the instruction which suffered the exception; the latter's address is lost. When an exception is signalled, the result delivered by the Weitek chips is *not* written to the destination FPA register specified by the FPA instruction, since in many cases that would destroy input data required to recompute the correct result.

The recomputation routines examine the FPA's instruction pipe, clear it, and recompute the first instruction in the pipe, which is the one that generated the exception. The recomputed result is placed in the FPA register that was the instruction's intended destination, and any IEEE exception flags raised during the course of the computation are OR'ed into the MC68881's Accrued Exception byte in the `fpsr` register.

After recomputation occurs, the contents of the FPA's `fpastatus` register and the MC68881's Condition Code, Quotient, and Exception Status bytes are undefined, except that after recomputation of an FPA comparison instruction (which happens when one or both operands are NaNs), `fpastatus` is properly defined so it can be read and then written into the MC68020's status register.

Inexact Exceptions

Inexact exceptions arise more often than not during floating-point operations. If inexact is the only exception that arose, the Weitek-computed numerical result is correct. Causing a recomputation on each such exception would unacceptably degrade performance. Consequently the FPA has a register called `imask` that allows software to control whether inexact exceptions invoke recomputation. Sun-supplied software uses `imask` in conjunction with the MC68881's status and control registers to maintain an accurate record of inexact exception occurrences without recomputing for each one. Sun's software conventions for `imask` use may be defeated if users alter `imask` other than indirectly through the function `_fpmode_` described below.

When a `-ffpa` program begins execution, `imask` is set to 1 so that the FIRST inexact exception detected by the FPA causes a recomputation. After the recomputation, the inexact Accrued Exception bit in the MC68881's `fpsr` is turned

on and `imask` is cleared UNLESS the inexact operation Exception Enable bit is set in the MC68881's `fpcr`. The latter bit, normally cleared, may be set by the user to indicate that EVERY inexact FPA operation should cause recomputation by the MC68881, with a trap occurring if the MC68881 recomputation is also inexact.

If you use either the function `_fpstatus_` to clear the MC68881 `fpsr`'s inexact Exception Accrued bit, or `_fpmode_` to set the `fpcr`'s inexact operation Exception Enabled bit, then the FPA's `imask` will be set so that the next inexact operation will cause recomputation.

Traps

Traps on FPA instructions can be obtained by setting bits in the MC68881's Exception Enable byte in the `fpcr` register. When an exception is detected by a Weitek chip the FPA calls for recomputation; if that recomputation causes a MC68881 exception corresponding to a set Exception Enable bit then a MC68881 trap occurs, manifested as a `SIGFPE` with a code equal to one of

```
FPE_FLT{BSUN,INEX,DIV,UND,OPERR,OVF,NAN}_TRAP
```

The MC68881 trap occurs after the FPA recomputation takes place; the pc in the `SIGFPE` handler parameter `scp` will point to an FPA instruction in the user's code, the same one pointed to by the orgignal recomputation request. The address of the last MC68881 instruction executed in `fpa_handler()` is in the MC68881's `fpiar` register.

Writing a `SIGFPE` trap handler for an MC68881 trap generated indirectly by the FPA requires considerable care. The same trap handler must handle both the FPA's request for recomputation and the MC68881's trap signal.

Sun does not support user-written replacements for `fpa_handler()`. If you choose to write one, be sure to meet all hardware error-handling requirements; in particular, clear the FPA's pipe before attempting any other FPA instructions. Otherwise an infinite loop of `SIGFPE`'s is likely to result.

Performance

Because the treatment of floating-point exceptions is so variable, most floating-point programs that run reliably on machines with different types of arithmetic do not cause any floating-point exceptions to occur other than inexact. Consequently Sun's FPA recomputation scheme was designed without consideration of efficiency other than in the inexact case.

Programs that generate many exceptions may run more slowly than software floating point due to the intervention of the operating system to handle bus errors and signalling. The most likely cases where this might arise are computations which frequently underflow or which handle many NaNs. If recomputation performance is an issue, it may be partially ameliorated by using the Weitek chips in "fast" mode. "Fast" mode, although inconsistent with the IEEE Standard, causes no ill effects on programs that work correctly on a variety of non-IEEE machines, since those machines probably underflow abruptly to zero without creating subnormal (denormalized) numbers as IEEE machines do. In the Weitek chips' normal "IEEE" mode, recomputation is invoked if either operand

or the result is a subnormal number. In the Weitek chips' "fast" mode, recomputation is only invoked if the result would be a subnormal number; subnormal operands are treated as zero without generating any exception.

The Weitek `fpamode` register is intended for use by Sun software. If you alter it directly using an assembly language instruction

```
fpmove #0x3,fpamode  | Set "FAST" mode+standard rounding.
```

or by an equivalent FORTRAN call

```
    call fpamode(3)
```

be careful to preserve the values of mode bits that you do not intend to change. The default value of `fpamode` is 2, corresponding to rounding to nearest mode, treating subnormal operands in "IEEE" mode, and converting floating to integer in round-toward-zero mode.

The following program illustrates performance aspects in the face of underflows — a few (c = 0.49) or a lot (c = 0.51).

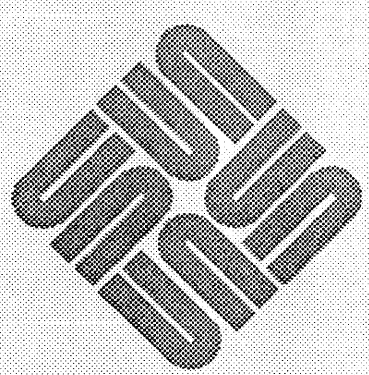
```
c   call fpamode(3)
    x = 1
    c = 0.49
c
c   or 0.51 for a lot
c
    do 1 i= 1, 10 000 000
1   x = c * x
    print *, c, x
    end
```

The execution time in seconds for various compilation options is as follows:

```
 c=0.49      c=0.51       Options

  112          491      -O -fsoft
  100          162      -O -f68881 -fstore
   20           70      -O -f68881
   15        13800      -O -ffpa
   15           15      -O -ffpa (call fpamode(3) uncommented)
```

3

IEEE-Standard Conformance



3

IEEE-Standard Conformance

The conformance of Sun software to the IEEE Standard varies depending on what compiler code generation options are chosen. The description in this chapter refers to the underlying arithmetic of Release 3.2 FORTRAN, Pascal, and C. Some of the features of the IEEE Standard that are unsupported in higher-level languages must be accessed through constructed subroutines, generally in assembly language; examples of these are listed later.

3.1. Supported IEEE Standard Features

Numeric Formats

All Sun floating-point options support single and double precision. Extended precision is supported only in assembly language for the MC68881, as are rounding precision modes. Signalling NaNs are only recognized with `-f68881` and `-ffpa`.

ASCII-to-Binary Conversion

Sun's current software does not support IEEE-standard conversion from ASCII strings representing decimal numbers to binary floating point or the reverse. Such conversion is always performed in the default rounding mode, signals no exceptions, and errors may be slightly larger than prescribed by the Standard, particularly in double precision. Assembly language programmers can obtain standard conversion using an A93N MC68881, if one is present.

Rounding Direction Modes, Exceptions, and Traps

Rounding mode, exception, and trap support are handled differently by the various floating-point options:

- `-fsoft` and `-fsky` don't support rounding modes, exceptions, or traps
- `-f68881` supports rounding direction and precision modes, exceptions, and traps
- `-ffpa` supports rounding direction modes and exceptions but not the full trapping mechanism recommended by the IEEE Standard

Numerical Results

Correct single- and double-precision numerical results are obtained for all operations (except conversion between ASCII and binary) in the following cases:

- `-fsoft` in round-to-nearest mode
- `-ffpa`, provided the MC68881's version is A93N
- `-f68881`, if the MC68881 is A93N and MC68881 rounding precision is set to double

- `-fsky`'s nonzero results may vary from the Standard's prescription by one bit; zero results may have the incorrect sign.

Correct results for `-ffpa` and `-f68881` depend on the presence of an A93N mask version MC68881.

MC68881 Rounding Precision Mode

Extended-precision format is optional in the IEEE Standard, but if provided rounding precision modes must also be supplied. These modes govern the rounding of extended-precision results; mode settings make it possible to round extended-precision results to shorter precisions in order to simulate results obtainable on systems without extended precision.

The default MC68881 rounding precision mode is extended. This means that extended-precision operations upon extended-precision operands produce extended-precision results and exceptions that conform completely to the Standard. The design of the MC68881 makes this mode substantially more efficient than single or double rounding precision modes. `-f68881` sets up the default MC68881 settings, including extended rounding precision mode.

However, a double-precision operation that computes a double-precision result from double-precision operands will then be computed on the MC68881 with two roundings:

```
fmoved  x,fp0   | This operation is exact.
fmuld   y,fp0   | Round extended-precision fp0 here.
fmoved  fp0,z   | Round double-precision z here.
```

From the point of view of verifying double-precision conformance to the IEEE Standard, multiple roundings on a single arithmetic operation cause incorrect results to be generated occasionally, even though default extended-precision accumulation is generally faster and more accurate. The double rounding precision mode can be used to insure that only one rounding occurs (in `fmuld` in the example above), mainly for the purpose of verifying conformance with the IEEE test vectors and the `Paranoia` program discussed later.

Programs should not generally use double rounding precision mode unless they require, on an operation-by-operation basis, the same results as would be obtained without extended precision.

Ordered Comparisons

The FORTRAN logical operators `.eq.` `.lt.` `.le.` `.gt.` `.ge.` and their analogs in C and Pascal return false if either operand is a NaN. With `-f68881` and `-ffpa`, all except `.eq.` and `.ne.` also cause an exception if either operand is a NaN.

3.2. How to Use Special Features of the IEEE Standard

Certain features of the IEEE Standard are not readily available from higher-level languages. Standard language or library extensions will eventually be developed to allow machine-independent access to these features. Sun supports such standardization efforts and, in the meantime, provides some provisional assembly language methods for accessing the special features. Since these provisional methods are nonstandard and probably transitory, limit their use to situations in which there is no other way to achieve the intended effect.

Rounding Modes and Trap Enabling

Rounding modes and trap enabling are available with `-f68881` and `-ffpa`. These fields are contained in the MC68881's `fpcr` register; see the MC68881 User's Manual. The following `libc` routine exchanges a new 32-bit mode control word with the existing one:

`as:`

```
    .globl   _fpmode_
_fpmode_:
|   On entry, sp@(4) contains the address of the
|     new 32-bit value to be placed in fpcr.
|   On exit,  d0 contains the previous fpcr value.
```

`f77:`

```
    integer function fpmode( new )
    integer new
```

`cc:`

```
    int fpmode_(new)
    int *new ;
```

`pc:`

```
    function fpmode( new : integer ) : integer ;
    external fortran ;
```

Rounding direction modes are effective with `-f68881` and `-ffpa`. Rounding precision modes are only effective with `-f68881`.

Trap Enabling

Trap enabling with `_fpmode_` will cause `SIGFPE` signals when the MC68881 detects an exception whose trap is enabled. The `SIGFPE` code indicates the specific exception; the MC68881's `fpiar` register contains the address of the MC68881 instruction which caused the exception (the `pc` is usually advanced beyond that point before the exception is recognized).

Using Special Features From FORTRAN

```
c   Set rounding direction mode to round toward zero:

    integer oldmode, fpmode
    oldmode = fpmode(16)
```

```
c   Set rounding precision mode to double precision:

    integer oldmode, fpmode
    oldmode = fpmode(128)
```

```
c   Enable signalling SIGFPE on overflow, division by zero,
c   or invalid operation:

    integer oldmode, fpmode
    oldmode = fpmode(62464)
```

Exceptions, Condition Code, and Quotient Status

`-f68881` and `-ffpa` provide exceptions, condition code, and quotient status in the MC68881's `fpsr` register; see the MC68881 User's Manual. The `libc` routine `_fpstatus_` exchanges a new 32-bit status word with the existing one, somewhat like `_fpmode_`:

```
    .globl   _fpstatus_
_fpstatus_:
|       On entry, sp@(4) contains the address of the
|        new 32-bit value to be placed in fpsr.
|       On exit, d0 contains the previous fpsr value.
```

Only the Accrued Exception byte of `fpsr` is defined with `-ffpa`. Here is an example of testing the Accrued Exception byte from FORTRAN:

```
c   Test accrued exception bits to determine if overflow,
c   division by zero, or invalid operation has occurred;
c   also clear status bits:

    integer oldstatus, fpstatus
    oldstatus = fpstatus(0)
    if (and(oldstatus,128) .ne. 0) print *,' invalid operation occurred'
    if (and(oldstatus,16) .ne. 0) print *,' division by zero occurred'
    if (and(oldstatus,64) .ne. 0) print *,' overflow occurred'
```

Signalling NaNs

Signalling NaNs are detected by the MC68881 as NaNs containing a cleared leading fraction bit. When detected, `SIGFPE` is signalled with the `FPE_FLTNAN_TRAP` code. The FPA always causes recomputation when a NaN is encountered; the MC68881 distinguishes quiet from signalling during recomputation. `-fsoft` and `-fsky` treat all NaNs as quiet.

Using `_fpmode_` and `_fpstatus_` from C

The following program comprehensively demonstrates using the `_fpmode_` and `_fpstatus_` functions from C to determine existing settings and install new ones.

```
enum fp_rounding_direction      /* rounding direction type */
    {
    fp_rd_nearest   = 0,
    fp_rd_zero  = 1,
    fp_rd_plus  = 2,
    fp_rd_minus = 3
    } ;
```

```
enum fp_rounding_precision      /* extended rounding precision type */
    {
    fp_rp_extended  = 0,
    fp_rp_single    = 1,
    fp_rp_double    = 2,
    fp_rp_3     = 3
    } ;

enum fp_accrued_type        /* accrued exceptions according to fpsr bit number */
    {
    fp_inexact  = 3,
    fp_divide   = 4,
    fp_underflow    = 5,
    fp_overflow = 6,
    fp_invalid  = 7
    } ;

enum fp_exception_type          /*  exceptions according to fpcx/fpsr bit number */
    {
    fp_inex1    = 8,
    fp_inex2    = 9,
    fp_dz       = 10,
    fp_unfl     = 11,
    fp_ovfl     = 12,
    fp_operr    = 13,
    fp_snan     = 14,
    fp_bsun     = 15
    } ;

unsigned
print_81_mode(newmode)          /* Exchanges floating-point mode and
                        prints out new mode settings. */
    unsigned            newmode;
{
    unsigned            oldmode, fpmode_();

    oldmode = fpmode_(&newmode);
    printf(" New floating-point mode: \n");
    printf(" Rounding direction toward ");
    switch ((newmode >> 4) & 3) {
    case fp_rd_nearest:
        printf(" nearest ");
        break;
    case fp_rd_zero:
        printf(" zero ");
        break;
    case fp_rd_minus:
        printf(" minus infinity ");
        break;
    case fp_rd_plus:
        printf(" plus infinity ");
        break;
    }
```

```
    printf("\n");
    printf(" Extended rounding precision ");
    switch ((newmode >> 6) & 3) {
    case fp_rp_extended:
        printf(" extended ");
        break;
    case fp_rp_single:
        printf(" single ");
        break;
    case fp_rp_double:
        printf(" double ");
        break;
    }
    printf("\n");
    printf(" Enabled exceptions: ");
    if ((newmode >> (unsigned) fp_inex1) & 1)
        printf(" inex1 ");
    if ((newmode >> (unsigned) fp_inex2) & 1)
        printf(" inex2 ");
    if ((newmode >> (unsigned) fp_dz) & 1)
        printf(" dz ");
    if ((newmode >> (unsigned) fp_unfl) & 1)
        printf(" unfl ");
    if ((newmode >> (unsigned) fp_ovfl) & 1)
        printf(" ovfl ");
    if ((newmode >> (unsigned) fp_operr) & 1)
        printf(" operr ");
    if ((newmode >> (unsigned) fp_snan) & 1)
        printf(" snan ");
    if ((newmode >> (unsigned) fp_bsun) & 1)
        printf(" bsun ");
    printf("\n");
    return (oldmode);
}

unsigned
print_81_status(newstatus)          /* Exchanges floating-point status and
                                     prints out old status settings. */
    unsigned            newstatus;
{
    unsigned            oldstatus, fpstatus_();

    printf(" Old floating-point status: \n");
    oldstatus = fpstatus_(&newstatus);
    printf(" Current exceptions: ");
    if ((oldstatus >> (unsigned) fp_inex1) & 1)
        printf(" inex1 ");
    if ((oldstatus >> (unsigned) fp_inex2) & 1)
        printf(" inex2 ");
    if ((oldstatus >> (unsigned) fp_dz) & 1)
        printf(" dz ");
    if ((oldstatus >> (unsigned) fp_unfl) & 1)
        printf(" unfl ");
```

```
    if ((oldstatus >> (unsigned) fp_ovfl) & 1)
        printf(" ovfl ");
    if ((oldstatus >> (unsigned) fp_operr) & 1)
        printf(" operr ");
    if ((oldstatus >> (unsigned) fp_snan) & 1)
        printf(" snan ");
    if ((oldstatus >> (unsigned) fp_bsun) & 1)
        printf(" bsun ");
    printf("\n");
    printf(" Accrued exceptions: ");
    if ((oldstatus >> (unsigned) fp_inexact) & 1)
        printf(" inexact ");    /* inex1 + inex2 */
    if ((oldstatus >> (unsigned) fp_divide) & 1)
        printf(" divide "); /* dz */
    if ((oldstatus >> (unsigned) fp_underflow) & 1)
        printf(" underflow ");  /* unfl */
    if ((oldstatus >> (unsigned) fp_overflow) & 1)
        printf(" overflow ");   /* ovfl */
    if ((oldstatus >> (unsigned) fp_invalid) & 1)
        printf(" invalid ");    /* operr + snan + bsun */
    printf("\n");
    return (oldstatus);
}

main()
{               /* Demonstration program */
    double          x;
    double          zero = 0.0, one = 1.0, three = 3.0, big = 1.0e+300,
            small = 1.0e-300;

    print_81_mode(0);   /* Default modes. */
    x = one / three;    /* Inexact. */
    print_81_status(0); /* Inexact message; clear status. */
    x = one / zero;     /* Divide by zero. */
    print_81_status(0); /* Divide by zero message; clear status. */
    print_81_mode(0x50);    /* Round toward zero, round extended to
                * single precision. */
    x = zero / zero;    /* Invalid operation. */
    x = big * big;      /* Overflow, inexact. */
    x = small * small;  /* Underflow, inexact. */
    print_81_status(0); /* Invalid operation, overflow,
                * underflow,inexact message, clear status. */
}
```

3.3. Special Library Entry Points

Special library entry points are defined to supply certain IEEE operations. The names and calling sequences are definitely subject to change, but they are listed here to support the IEEE conformance claims below. The first letter indicates the type of arithmetic:

- V... `-fswitch`

- F... `-fsoft`
- S... `-fsky`
- M... `-f68881`
- W... `-ffpa`

while the last letter indicates the precision:

- ...s single
- ...d double

Calling conventions are shown in the table below. Not all entry points have two arguments.

	First Argument/Result	*Second Argument*
Single precision	`d0`	`d1`
Double precision	`d0/d1`	`a0@`

The IEEE remainder, a function of two arguments, is available from these assembly language entry points:

```
.globl    Vrems,Vremd,Frems,Fremd,Srems,Sremd,Mrems,Mremd,Wrems,Wremd
```

Convert floating to integral value in integer format with rounding toward zero, a one-argument function, is available from these assembly language entry points:

```
.globl    Vints,Vintd,Fints,Fintd,Sints,Sintd,Mints,Mintd,Wints,Wintd
```

This function is directly available in FORTRAN (`INT`), Pascal (`trunc`), and C (`int`).

Convert floating to integral value in integer format with rounding to nearest, a one-argument function, is available from these assembly language entry points:

```
    .globl  Frints,Frintd,Srints,Srintd
```

Convert floating to integral value in integer format with rounding according to current IEEE rounding mode, a one-argument function, is available from these assembly language entry points:

```
    .globl  Mrints,Mrintd,Wrints,Wrintd
```

Convert floating to integral value in floating-point format with rounding toward zero, a one-argument function, is available from these assembly language entry points:

```
.globl    Vaints,Vaintd,Faints,Faintd,Saints,Saintd,Maints,Maintd,Waints,Waintd
```

This function is directly available in FORTRAN (`AINT`).

Convert floating to integral value in floating-point format with rounding to nearest, a one-argument function, is available from these assembly language entry points:

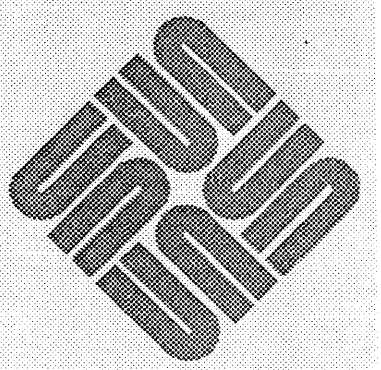
```
.globl  Farints,Farintd,Sarints,Sarintd
```

Convert floating to integral value in floating-point format with rounding according to current IEEE rounding mode, a one-argument function, is available from these assembly language entry points:

```
.globl  Marints,Marintd,Warints,Warintd
```

4

Benchmarks



4

Benchmarks

Several benchmarks are described in this chapter to give you some idea of the performance and degree of conformance of the floating-point support provided for Sun's systems.

The benchmarks described include conformance benchmarks, elementary function accuracy benchmarks, and performance benchmarks.

> **Note:** All specific claims for conformance, accuracy, and speed are based on tests and measurements made using a preliminary version of Sun Software Release 3.2. Results obtained with the final Release 3.2 or with other releases may vary.

The conformance benchmarks described in this chapter are:

- IEEE test vectors.
- `Paranoia`.

4.1. IEEE Test Vectors

Conformance to the IEEE Standard is primarily tested by running the IEEE test vectors, a collection of 15 sets of input data and output numerical results and exceptions, distributed by the University of California. A test program converts the input data from its distributed symbolic form to the corresponding binary representation, obtains the result delivered by Sun floating-point software or hardware, and compares the computed result and exceptions with those prescribed by the Standard.

Sun's test program tests single-precision results computed from single-precision operands, and double-precision results computed from double-precision operands. The test program obtains access to Sun's floating-point software or hardware primarily through FORTRAN. The following double-precision add test function is typical:

```
real*8 function addd ( x, y)
real*8 x, y
addd = x + y
end
```

Test vectors for the following functions are tested with FORTRAN subprograms like `addd`:

- `absolute value`
- `add`
- `compare`
- `divide`
- `multiply`
- `negate`
- `subtract`
- `square root`

Some functions are coded in assembly language as described in Appendix D. Test vectors for the following functions are tested with such assembly language subroutines:

- `copysign`
- `fraction part`
- `logb`
- `nextafter`
- `remainder`
- `round to integer`
- `scalb`

The test vectors as distributed contain 21002 test cases when all rounding modes are tested and 5280 test cases when only the default round-to-nearest mode is tested.

Software Test Vector Results

With `-fsoft`, IEEE test vectors find no numerical errors in the default rounding direction mode. Software floating point does not support other rounding direction modes or IEEE exceptions.

Sky Test Vector Results

With `-fsky`, IEEE test vectors find 59 numerical errors in the default rounding direction mode, 20 in addition, 20 in subtraction, 8 in multiplication, 2 in division, and 9 in square root. All are errors of one unit in the last place or errors in the sign of zero results. The errors arise because of limitations in the microcode for the Sky FFP. Sky floating point does not support other rounding direction modes or IEEE exceptions.

MC68881 Test Vector Results

With A93N `-f68881`, and with *double rounding* precision mode, IEEE test vectors find no numerical or exception errors in any rounding direction mode.

With A93N `-f68881`, and with *extended rounding* precision mode, IEEE test vectors find 22 numerical errors and 50 exception errors in addition (2, 0), subtraction (2, 0), multiplication (2, 27), division (11, 23), and square root (5, 0), testing all rounding direction modes. All numerical errors are one unit in the last place. The errors arise because extended rounding precision occasionally causes different results from double rounding precision as discussed above.

A79J MC68881 Test Vector Results

With A79J `-f68881`, and with *double rounding* precision mode, IEEE test vectors find 27 numerical errors and no exception errors, in multiplication (16), division (5), and scalb (6), testing all rounding direction modes. All numerical errors relate to erroneous zero results being computed when correctly rounded results would be the smallest normalized number.

Sun FPA Test Vector Results

With `-ffpa` and an A93N MC68881 installed, IEEE test vectors find no numerical or exception errors in any rounding direction mode.

Sun FPA Test Vector Results, `fpamode(3)`

With `-ffpa` and "fast" fpamode enabled, IEEE test vectors find 1616 numerical errors and 1032 exception errors in addition, subtraction, multiplication, division, comparison, round to integral value, and nextafter. This mode is not intended to conform to the IEEE standard.

4.2. `Paranoia` Test Program

The `Paranoia` test program was developed by W. Kahan at the University of California to test implementations of floating-point arithmetic by ostensibly machine-independent means. Shortcomings in numerics are detected and classified as DISASTERS, FAILURES, SEVERE DEFECTS, DEFECTS, FLAWS, and WARNINGS. At the end of the program, the number of shortcomings in each class is displayed, and if the arithmetic appears to conform to the IEEE Standard, the program so states. Sun tests Fortran, C, and Pascal versions of `Paranoia` that evolved from Kahan's original 1983 version in Basic.

In the following tables, DEFECTS, FLAWS, and WARNINGS encountered are enumerated Di, Fj, and Wk, and described afterward. 754 indicates the arithmetic appeared to conform to the IEEE Standard.

Single-Precision Paranoia Results

Compiler Options	Fortran	C	Pascal
-O -ffpa	754 W1	754 W1	754 W1
-O -f68881		754 W1	754 W1
-O -f68881 -fstore	754 W1		
-O -fsky	D1 D2 D3 F1 F2 W1	D4 F1 F2 W1	D1 F1 F2 W1
-O -fsoft	754 W1	754 W1	754 W1

Double-Precision Paranoia Results

Compiler Options	Fortran	C	Pascal
-O -ffpa	754 W1	754 W1	754 W1
-O -f68881		F1 W1	F1 W1
-O -f68881 -fstore	F1 W1		
-O -fsky	F1 W1	F1 W1	F1 W1
-O -fsoft	754 W1	754 W1	754 W1

```
    Key to Shortcomings:

D1: DEFECT: sqrt(  0.2401000e+04) -   0.4900000e+02  =    0.3814697e-05
    instead of correct value  0 .
D2: DEFECT: computed  (  0.20000000e+01)**(11) =   0.20480002e+04
```

```
    compares unequal to correct   0.20480000e+04 ;
    they differ by    0.24414063e-03.
    Error like this may invalidate financial
    calculations involving interest rates.
D3: DEFECT: computed  (  0.20000000e+01)**(120) =   0.13292334e+37
    compares unequal to correct   0.13292280e+37 ;
    they differ by    0.53875151e+31 .
D4: DEFECT:  Comparison alleges that what prints as Z = 1.40129846432481710e-45
    is too far from sqrt(Z) ** 2 = 0.00000000000000000e+00 .
F1: FLAW: lack(s) of guard digits or failure(s) to correctly round or chop
F2: FLAW: underflow can stick at an allegedly positive value  z0
    that prints out as   0.14012985e-44
W1: WARNING: computed  (  0.00000000e+00)**(  0) = NaN
    compares unequal to correct   0.10000000e+01
```

That peculiar results for `x**y` count only as WARNINGS when `x=0` and `y<=0` reflects a division of opinion among analysts. The original BASIC version of `Paranoia` reflects Kahan's view that `x**0` should be 1, even if `x` be zero, infinity, or NaN. Subsequent translators felt strongly differently, and so changed the code accordingly to note these cases as WARNINGS without otherwise counting them in determining the quality of implementations.

`-ffpa` Comments

In "fast" mode, selected by

```
call fpamode(3)
```

Paranoia discovers an inconsistency of underflow treatment: underflow itself leads to recomputation of a subnormal result, but in subsequent use that result is sometimes treated as zero. Thus in single precision, `Paranoia` output includes the following:

```
SERIOUS DEFECT:  X = 1.61630e-38 is Unequal to Z = 1.17549e-38,
yet X-Z yields 4.40810e-39.
```

"Fast" mode is not intended to conform to the IEEE Standard.

`-f68881` Comments

The FORTRAN version of `Paranoia` must be compiled with `-fstore` to force rounding to storage precision on assignment. If `-fstore` is omitted, Paranoia encounters numerous FAILURES and SERIOUS DEFECTS.

Double precision does not conform to the IEEE standard even with `-fstore` because of multiple roundings. However, by selecting double rounding precision mode with

```
call fpmode(128)
```

the result becomes 754 F1.

`-fsky` Comments

Shortcomings, particularly in single precision, are due to limited microcode space on the Sky FFP. C, FORTRAN, and Pascal defects differ because `sqrt` and `pow` are implemented in double precision in C, while `pow` is implemented in double precision in Pascal.

Summaries

Each program prints an overall summary of the tested arithmetic. The summaries corresponding to the tables above are as follows:

```
754 W1:
    No failures, defects nor flaws have been discovered.
    Rounding appears to conform to IEEE standard  p754.
    The arithmetic diagnosed appears to be excellent!
F1 W1:
    The arithmetic diagnosed seems satisfactory though flawed.
D1 D2 D3 F1 F2 W1:
D4 F1 F2 W1:
D1 F1 F2 W1:
    The arithmetic diagnosed may be acceptable despite inconvenient defects.
```

4.3. Elementary Function Accuracy Benchmarks

The accuracy of elementary transcendental functions is tested using the `Elefunt` FORTRAN test suite of Cody and Waite, and a preliminary C test suite of Alex Liu, of the University of California. More information on these programs may be obtained from sources listed in the Preface to this manual.

All programs are compiled with `-O`; MC68881's are used only in the default extended rounding precision mode. For these tests, A79J and A93N masks produce the same results.

`Elefunt` Test Programs

In Cody and Waite's program, each function is tested by checking several independent identities at many random points; reported errors measure the degree to which the computed function values fail to satisfy certain identities, rather than the errors in the computed function values themselves. In addition to testing identities at random points, as shown below, Cody and Waite's programs also test error responses at isolated points.

Cody and Waite advise that it may be necessary to modify their test programs when extended-precision expression evaluation is performed, as in the case of Sun's `-f68881` option. Sun's version of `Elefunt` accordingly contains changes to the codes that purify test arguments; these changes are listed below under the commented-out source lines they replace:

```
c    x = (x + eight) - eight          alog0870
     x = x + eight
     x = x - eight
c    x = (x + eight) - eight          alog0940
     x = x + eight
     x = x - eight
c    x = (x + w) - w                  powr0940
     x = x + w
     x = x - w
c    y = (x + y) - x                  sin00760
     y = y + x
     y = y - x
```

Below are worst-case differences detected in identities tested, expressed as the number of bits of precision lost.

Greatest number of bits lost - single-precision elefunt

compiler option->	-fsoft	-fsky	-f68881	-f68881 -fstore	-ffpa
function:					
sqrt	0	1.0	0	0	0
exp	1.0	2.0	0	1.0	1.0
log	1.0	2.6	0	1.0	1.0
log10	2.4	3.0	3.0	2.1	2.1
x**y	1.0	7.3	3.2	1.0	1.0
sinh/cosh	2.0	2.0	0	1.0	1.6
tanh	2.1	2.0	0	1.0	1.5
sin/cos	1.5	3.1	0	0.6	1.5
tan	1.8	2.6	0	1.3	1.8
asin/acos	1.9	2.0	0	1.0	1.0
atan	1.0	1.9	0	1.0	1.0

Greatest number of bits lost - double-precision elefunt

compiler option->	-fsoft	-fsky	-f68881	-f68881 -fstore	-ffpa
function:					
sqrt	0	0	0	0	0
exp	1.0	1.0	0	1.0	1.0
log	1.0	1.0	0	1.0	1.0
log10	2.4	2.6	3.1	2.1	2.1
x**y	2.3	2.4	6.2	2.2	2.2
sinh/cosh	2.1	2.1	0	1.0	1.6
tanh	2.1	2.1	0	1.0	1.5
sin/cos	1.5	1.5	0	0.7	1.5
tan	2.0	2.1	0	1.2	2.0
asin/acos	1.2	1.2	0	1.0	1.0
atan	1.0	1.4	0	1.0	1.7

Test Programs of Liu

In Liu's test program, 64 random points are tested in 64 regions for each function; reported errors are the errors in the computed function values themselves, tested against more accurate values generated internally by Liu's program. Tests of `e**x-1` and `log(1+x)` used assembly language entry points [FSMW]{exp1,log1}[sd].

Below are worst-case errors in ulps, units in the last place, detected for each function tested. A difference of one ulp corresponds to a difference in the least significant bit.

Greatest errors in ulps - single-precision Liu tests

compiler option->	-fsoft	-fsky	-f68881	-ffpa
function:				
sin	0.88	5.79	0.54	0.88
cos	0.90	5.95	0.53	0.90
atan	0.86	2.08	0.52	0.86
log	0.92	3.48	0.53	0.92
log(1+x)	0.88	0.90	0.53	1.06
e**x - 1	0.94	0.94	0.53	0.95

```
    Greatest errors in ulps - double-precision Liu tests

compiler option->       -fsoft     -fsky     -f68881    -ffpa

function:
sin                       1.01      0.99        0.57      1.01
cos                       1.01      0.90        0.55      1.01
atan                      1.06      0.98        0.56      1.02
log                       0.92      0.96        0.59      0.92
log(1+x)                  0.83      0.83        0.59      0.83
e**x - 1                  1.13      1.00        0.57      1.18
```

Monotonicity

Operations specified in the IEEE Standard are required to be monotonic; elementary transcendental functions are often expected to be monotonic in their primary domains. Although expectations about monotonicity are seldom explicit in programs, monotonicity failures can befuddle debugging.

Liu's program tests monotonicity at selected points; failures were detected only with -fsky and only in single precision as follows:

```
log:      4 monotonicity failures including   3.2768000000000000e+04
atan:   449 monotonicity failures including   5.2225278125000000e+05
```

Although exhaustive testing of monotonicity is infeasible for double precision, it is in principle possible for single precision. At present, the following single precision IEEE Standard and Fortran functions have been tested over the indicated primary domains by evaluation at each single-precision representable number in the domain. If monotonic, execution times are shown in hours; "no" means not monotonic:

```
IEEE function and domain                          -ffpa   -f68881  -fsky   -fsoft
                                                          A93N

x*x    [          0.     , Inf            ]          8       30       25
abs    [          0.     , Inf            ]  3       5       17       7
aint   [ -Inf            , Inf            ]  15      14                20
arint  [ -Inf            , Inf            ]  20      34
ceil   [ -Inf            , Inf            ]  22      46
floor  [ -Inf            , Inf            ]  21      46
dble   [ -Inf            , Inf            ]  14      10       72       37
int    [ -0.21474836e+10, 2.14748352e+9 ]          18
rint   [ -0.21474836e+10, 2.14748352e+9 ]          24
sqrt   [          0.     , Inf            ]  10      9        71       49
```

FORTRAN function and domain			-ffpa	-f68881 A93N	-fsky	-fsoft
acos	[-1.0000000	, 1.0000000]	29	no	no	
anint	[-Inf	, Inf]	28	43		
asin	[-1.0000000	, 1.0000000]	21	19	no	
atan	[-Inf	, Inf]	16	26	no	
cos	[-3.1415925	, 0.]	4	9	72	38
cosh	[0.	, Inf]		22		
exp	[-Inf	, Inf]		41	no	
exp-1	[-Inf	, Inf]		62		
2**x	[-Inf	, Inf]		36		
10**x	[-Inf	, Inf]		44		
log	[0.	, Inf]	19	31	no	
log1+x	[-1.	, Inf]	22	46	no	
log2	[0.	, Inf]		38		
log10	[0.	, Inf]		27		
nint	[-0.21474836e+10	, 2.14748352e+9]	27	27		
sin	[-1.5707963	, 1.5707963]	4	11	no	45
sinh	[-Inf	, Inf]		44		
tan	[-1.5707963	, 1.5707963]	16	14	90	58
tanh	[-Inf	, Inf]		49		

The `-f68881 acos` monotonicity failure is in extended precision and is not detected if `-fstore` is in effect.

All tests were compiled with `-O` and without `-fstore`, and were made on Sun-3's except `-fsky`, made on a Sun-2. Timings are the entire execution time for the test program and consequently give only a rough comparison of function speed. As the execution times imply, these tests constitute an ongoing project.

4.4. Performance Benchmarks

The performance benchmarks described in this section, `Linpack` and `SPICE`, are commonly used to compare the relative performance of different manufacturer's computers.

`Linpack`

The most realistic of the frequently cited benchmarks of peak floating-point speed is the `Linpack` benchmark, a FORTRAN program that determines the time required to solve a 100x100 system of linear equations using the `Linpack` linear algebra subroutine library.

Sun's version of the `Linpack` benchmark program reports performance in thousands of floating-point operations per second (KFLOPS). Performance reported below is based upon measuring total user and system CPU time used. All results are based on compiling FORTRAN source code; `-fsoft`, `-fsky`, and `-fswitch` were compiled on Sun-2's; `-f68881` and `-ffpa` were compiled on Sun-3's.

All floating-point hardware runs at 16.7 MHz except as noted.

Sun-2 results (10 MHz MC68010 CPU):

Compiler Options	Single Precision KFLOPS	Double Precision KFLOPS	Comments
-O -fsoft	13	6	
-O -fswitch	32	18	using Sky FFP
-O -fsky	49	27	

Sun-3 results (15 MHz MC68020 CPU):

FPU MHz	Compiler Options	Single Precision KFLOPS	Double Precision KFLOPS	Comments
	-O -fsoft	28	12	
12.5	-O -fswitch	47	34	using MC68881
12.5	-O -f68881	78	73	
	-O -fswitch	55	38	using MC68881
16.7	-O -f68881	93	87	

Sun-3 results (16.7 MHz MC68020 CPU):

FPU MHz	Compiler Options	Single Precision KFLOPS	Double Precision KFLOPS	Comments
	-O -fsoft	34	15	
12.5	-O -fswitch	50	37	using MC68881
	-O -fswitch	65	45	using MC68881
12.5	-O -f68881	85	80	
16.7	-O -f68881	108	100	
	-O -fswitch	185	105	using FPA
	-O -ffpa	500	315	
16.7	-O -f68881	87	82	double rounding
16.7	-f68881	95	87	
16.7	-O -f68881 -fstore	99	90	
16.7	-O -f68881	109	101	rolled
	-ffpa	280	160	
	-P -ffpa	420	285	
	-O -ffpa	615	405	rolled

The comment "double rounding" means that the double rounding precision mode of the MC68881 was selected. The comment "rolled" means that the inner loops of the Basic Linear Algebra Subroutines were rewritten slightly. The distributed form of the `Linpack` benchmark program has the key inner loop written in the

"unrolled" form

```
      do 1 i = 1, n, 4
      x(i  ) = x(i  ) + c * y(i  )
      x(i+1) = x(i+1) + c * y(i+1)
      x(i+2) = x(i+2) + c * y(i+2)
1     x(i+3) = x(i+3) + c * y(i+3)
```

because that form was faster on certain mainframes common in the mid-1970's. However, the unrolling defeats many current vectorizing compilers, so supercomputer manufacturers usually measure the rolled speed by rewriting the loop in the form:

```
      do 1 i = 1, n
1     x(i  ) = x(i  ) + c * y(i  )
```

Dongarra's `Linpack` benchmark performance compilation, listed in the Preface, annotates such results as "Rolled BLAS." Further complicating the issue is that compilers on some systems do not generate optimum code for the inner loop whether rolled or unrolled, so hand-coded assembly language is faster yet; these results are annotated as "Coded BLAS." For the rolled form of the `Linpack` benchmark, the Sun-3 code generated with `-O` and either `-ffpa` or `-f68881` is truly optimal and cannot be improved by hand coding in assembly language.

`SPICE`

The `SPICE` program simulates integrated circuits and is an example of a complete application that is floating-point intensive but does not continually attain peak performance rates as does the `Linpack` benchmark. It is therefore more representative of relative performance on realistic computations that do not closely fit the model of a Linear Algebra computation.

For benchmarking purposes Sun uses a FORTRAN implementation of `SPICE` version 2G.6, and a C implementation of `SPICE` version 3A.7, both with an input data file called MOSAMP2, listed in Appendix E. The reported time is the total elapsed real time in seconds for the complete program, invoked with a command like

```
time spice.out < mosamp2.input >! mosamp2.output
```

In each case the program was run at least twice and the fastest result reported. Note that the global optimizer `iropt` is available in Sun's `f77`, but not in Sun's `cc`. `-fswitch` and `-fsoft` versions were compiled on Sun-2's and run on Sun-2s and Sun-3s. `-fsky` versions were compiled and run on Sun-2s. `-f68881` and `-ffpa` versions were compiled and run on Sun-3s. All floating-point hardware runs at 16.7 MHz except as noted.

Results are:

FPU MHz	Compiler Options	Real Time to Execute (Seconds) 2G.6 Time	3A.7 Time
Sun-2 results (10 MHz MC68010 CPU):			
	-O -fsoft	1372	1979
	-O -fswitch+Sky	483	674
	-O -fsky	441	605
Sun-3 results (15 MHz MC68020 CPU):			
	-O -fsoft	595	858
12.5	-O -fswitch+A79J	184	262
12.5	-O -f68881+A79J	102	128
16.7	-O -fswitch+A93N	166	237
16.7	-O -f68881+A93N	88	109
Sun-3 results (16.7 MHz MC68020 CPU):			
	-O -fsoft	482	696
12.5	-O -fswitch+A79J	165	233
12.5	-O -f68881+A79J	92	117
16.7	-O -fswitch+A93N	139	198
16.7	-O -f68881+A93N	78	96
	-O -fswitch+fpa	76	105
	-O -ffpa	45	60

4.5. Benchmarking Hazards

It is not easy to construct meaningful benchmark programs that are short and easy to understand and provide a sound basis for projecting results for realistic applications. The following examples illustrate how seemingly minor variations in hardware or in coding techniques can have surprising results.

MC68020 Cache

The MC68020 has a 256-byte direct-mapped instruction cache. "Direct-mapped" means that memory address `X` is mapped to cache address `X mod 256`, so memory locations `X` and `X+256*n` cannot be in the cache at the same time. This can be a problem for short loops or subroutines that call other short subroutines. If the difference in memory addresses of the caller and callee happens to be close to a multiple of 256, almost every instruction fetch misses the cache and performance is degraded overall. Note that this is a problem particularly of direct-mapped caches, and is more likely to be encountered when such caches are small, and is most noticeable when the caller and callee are very short, so that the instruction fetches from main memory add up to a significant fraction of the overall time required.

Programs coded in higher-level languages or even in separate assembly-language modules are usually written without thought to where they might be placed in memory, since that is often an accident of code generation or linking order which may not be visible to or controllable by the programmer.

Whetstone

The `Whetstone` benchmark was originally an Algol 60 program that duplicated the typical instruction stream processed by the Whetstone Algol interpreter during the 1960's. Performance is reported as thousands of Whetstone interpreter instructions per second.

Various FORTRAN translations of this program have been used to benchmark floating-point performance, but interpretation of the results has become increasingly difficult as new computer architectures and optimization techniques develop. The `Linpack` and `SPICE` benchmarks discussed above give better information about relative performance of computers and compilers. However, the `Whetstone` program is ideal for illustrating MC68020 cache effects.

The `Whetstone` program consists of several timing loops, but the most important, that consumes about half the overall time, is a loop that calls a subroutine `P3`. The loop is

```
        DO 90 I=1,N8
          CALL P3(X,Y,Z)
90      CONTINUE
```

and the subroutine is

```
        SUBROUTINE P3(X,Y,Z)
        COMMON T,T2
        X1 = X
        Y1 = Y
        X1 = T * (X1 + Y1)
        Y1 = T * (X1 + Y1)
        Z = (X1 + Y1) / T2
        RETURN
        END
```

This code is intended to model an important class of numerical codes that solve nonlinear problems: optimization and finding zeros of functions. These codes repeatedly call user-supplied subroutines to evaluate the nonlinear functions in questions. Usually, the calling routine is rather more complicated than a `DO` loop and the parameters vary; in those respects `P3` is a deficient abstraction.

When compiled in single precision with `-O` and `-ffpa`, the results obtained for this program vary markedly depending on the cache contention between `P3` and its calling loop. Single precision and `-ffpa` were chosen for illustration since cache contention is most noticeable when the timings of the caller and callee are as similar as possible. Worst-case performance is 2100 thousand Whetstone instructions per second (KWIPS), best case is 2400. Which result is obtained depends on the relative compiled addresses of the `do` loop and `P3`, which vary among compiler releases.

The performance with the MC68020's cache disabled is about 1750 KWIPS, so even the worst cache case is much better. A more complicated cache architecture on the MC68020 might show less variation but might be slower overall.

Assembly Language Inline Expansion

One way to avoid cache contention between callers and callees is to expand procedures inline in the calling code. There is a facility in Sun's compilers to facilitate such expansion at the assembly-language level; see the appendix "Assembly-Level Inline Expansion". This facility is intended to allow short, simple assembly-language library routines to be expanded inline to bypass the procedure-call overhead of saving and restoring registers and pushing arguments. To demonstrate the facility, we apply it to `P3` by first compiling `P3` separately with `-O -ffpa -S` to obtain this assembly language:

```
        .data
        .comm   __BLNK__,40
        .text
        .globl  _p3_
_p3_:
        link    a6,#-16
        fpmoved fpa4,a6@(-16)
        fpmoved fpa5,a6@(-8)
        movl    a6@(8),a0
        fpmoves a0@,fpa4
        movl    a6@(12),a0
        fpmoves a0@,fpa5
        fpams   fpa5,fpa4,__BLNK__,fpa4
        fpams   fpa5,fpa4,__BLNK__,fpa5
        fpadd3s fpa5,fpa4,fpa2
        fpdivs  __BLNK__+8,fpa2
        movl    a6@(16),a0
        fpmoves fpa2,a0@
        fpmoved a6@(-8),fpa5
        fpmoved a6@(-16),fpa4
        unlk    a6
        rts
```

which was converted manually according to the inline expansion rules to

```
        .data
        .comm   __BLNK__,40
        .text
        .inline _p3_,12
        movl    sp@+,a0
        fpmoves a0@,fpa0
        movl    sp@+,a0
        fpmoves a0@,fpa1
        fpams   fpa1,fpa0,__BLNK__,fpa0
        fpams   fpa1,fpa0,__BLNK__,fpa1
        fpadd3s fpa1,fpa0,fpa2
        fpdivs  __BLNK__+8,fpa2
        movl    sp@+,a0
        fpmoves fpa2,a0@
        .end
```

When the FORTRAN `Whetstone` source, with `P3` removed, was compiled with a `.il` inline expansion file containing the above, the resulting performance was 3300 KWIPS. The inner loop was:

```
L77069:
        movl    #VAR_SEG1+4,a0
        movl    d6,__BLNK__+36
        movl    d7,__BLNK__+32
        movl    d3,__BLNK__+28
        fpmoves a0@,fpa0
        movl    #VAR_SEG1+8,a0
        fpmoves a0@,fpa1
        fpams   fpa1,fpa0,__BLNK__,fpa0
        fpams   fpa1,fpa0,__BLNK__,fpa1
        fpadd3s fpa1,fpa0,fpa2
        fpdivs  __BLNK__+8,fpa2
        movl    #VAR_SEG1+12,a0
        fpmoves fpa2,a0@
        movl    __BLNK__+32,d7
        movl    __BLNK__+36,d6
        movl    __BLNK__+28,d3
        dbra    d5,L77069
```

Source Level Inline Expansion

Of course, someone optimizing by hand would normally expand FORTRAN source inline into the FORTRAN source of the caller! If the following source code is compiled:

```
        DO 90 I=1,N8
        X1 = X
        Y1 = Y
        X1 = T * (X1 + Y1)
        Y1 = T * (X1 + Y1)
        Z = (X1 + Y1) / T2
 90     CONTINUE
```

then resulting performance is 4100 KWIPS. Examination of the generated code shows the same number of floating-point instructions executed:

```
L77069:
        fpams@5     fpa5,fpa7,__BLNK__,fpa6
        fpams@5     fpa5,fpa6,__BLNK__,fpa4
        fpadd3s@5       fpa4,fpa6,fpa8
        fpdivs@5        __BLNK__+8,fpa8
        dbra        d5,L77069
```

Global Interprocedural Analysis

Interestingly enough, executing the `P3` loop only once would be logically sufficient. Inspection shows that the inputs `X`, `Y`, `T`, and `T2` are the same on each invocation, so the call could be moved outside the loop entirely. An optimizing compiler that did interprocedural global analysis might detect this. To measure the effect such analysis would have if performed by a hypothetical compiler, we change the source code to:

```
        CALL P3(X,Y,Z)
        DO 90 I=1,N8
90      CONTINUE
```

Resulting performance is 4800 KWIPS; the floating point instructions are only executed once.

There is no compiler that can optimize code as well as a person who understands what is required to be done and how to do it! The Whetstone benchmark example illustrates this point extremely well, simultaneously rendering suspect its own value as a general-purpose floating-point benchmark.

Performance, Source Coding, and Optimization

A benchmark program by Myron Ginsberg was adapted to further illustrate how source coding techniques and compiler optimization techniques interact to affect the throughput on floating-point intensive calculations. This adaptation measures the time to evaluate a specific fifth-degree polynomial `A` at 500 points `X(I)` using several different methods. The "explicit" method is written out *in extenso* specifically for fifth-degree polynomials:

```
    DO 10 I=1,500
10  Y(I) = A(1) + X(I) * (A(2) + X(I) * (A(3) + X(I) *
    (A(4) + X(I) * (A(5) + X(I) * A(6)))))
```

The "external" method relies on a conventional external subroutine for evaluating polynomials of arbitrary degree:

```
        DO 10 I=1,500
        Y(I) = PEVAL( A, M, X(I))
10      CONTINUE
    ...
        REAL FUNCTION PEVAL( A, M, X )
        REAL A(*),X
        INTEGER M,K
        REAL P
        P = A(M+1)
        DO 15 K = M,1,-1
        P = P * X + A(K)
 15     CONTINUE
        PEVAL = P
        END
```

The "inline" method effectively expands the "external" code inline at the source level:

```
      P = A(NCOEF)
      DO 10 I=1,500
      DO 20 J=M,1,-1
20    P = A(NCOEF-J) + X(I) * P
10    Y(I) = P
```

The "vectored" method is intended to be optimized into appropriate vector operations on supercomputers:

```
      DO 10 L=1,500
10    BR(L,NCOEF) = A(NCOEF)
      DO 20 J=1,M
      DO 20 I=1,500
20    BR(I,NCOEF-J) = A(NCOEF-J) + X(I) * BR(I,NCOEF-J+1)
      DO 30 LL=1,500
30    Y(LL) = BR(LL,1)
```

The results are reported in KFLOPS. Each polynomial evaluation requires ten floating-point operations. The number of floating-point operations executed is the same in every case:

Single-Precision KFLOPS

Options	explicit	inline	external	vectored
-O -fsky	62	49	43	23
-O -f68881	210	130	125	73
-O -ffpa	1500	710	295-335	175

Double-Precision KFLOPS

Options	explicit	inline	external	vectored
-O -fsky	33	26	24	13
-O -f68881	200	120	115	67
-O -ffpa	1060	480	265-290	130

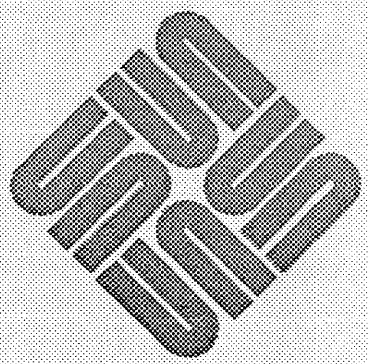
Note that the MC68020's internal cache causes the variable external case performance with `-ffpa`. The variation is insignificant with `-f68881`.

As the results indicate, the Sun FPA attains almost one third of the maximum theoretical throughput of the 1164/1165 chips - 4800 KFLOPS single, 3300 KFLOPS double. But this remarkably high efficiency is seldom achievable on realistic problems because performance is primarily limited by bus and memory bandwidth. The explicitly coded polynomial evaluation creates expressions so simple that Sun's FORTRAN compiler can allocate all the constants to FPA registers to minimize bus traffic. The vectorization-oriented evaluation, in contrast, is more difficult to optimize effectively for the FPA, since its complicated

expressions involve double subscripts and heavy bus traffic. These differences are most pronounced for high-performance floating-point hardware.

A

adb Changes



A

adb Changes

A.1. Changes in Release 3.1

Release of the Floating-Point Accelerator (FPA) required some changes to `adb`, in order to support assembly language debugging of programs that use the FPA. Here are the changes made to `adb` in Release 3.1:

1. The new debugger variables `A` through `Z` are reserved for special use by `adb`. They should not be used in `adb` scripts.
2. The FPA register names `fpa0` through `fpa31` are recognized and can be used or modified in debugger commands. (This extension only applies to machines with FPA's.)
3. The debugger variable `F` governs FPA disassembly. A value of 0 indicates that all FPA instructions are to be treated as move instructions. A nonzero value indicates that FPA instruction sequences are to be disassembled and single-stepped using FPA assembly language mnemonics. The default value is 1 on machines with FPA's; on other machines, the default value is 0.
4. The debugger variable `B` is used to designate an FPA base register. If FPA disassembly is disabled (the `F` flag = 0) its value is ignored. Otherwise, its value is interpreted as follows:

   0 through 7:
   Based-mode FPA instructions that use the corresponding address register in `[a0..a7]` to address the FPA are also disassembled using FPA assembler mnemonics. Note that this is independent of the actual run-time value of the register.

   otherwise:
   All based-mode FPA instructions are disassembled and single-stepped as move instructions.

   The default value of the FPA base register number is −1, which designates no FPA base register.
5. The command `$x` has been added to display the values of FPA registers `fpa0` through `fpa15`, along with FPA control registers and the current contents of the FPA instruction pipeline. All registers are displayed in the format:

```
<low word> <high word> <double precision> <single precision>
```

This verbose display is used because FPA registers are typeless; in particular, they may contain either single- or double-precision floating-point values. If a single-precision value is stored, it is always stored in the high-order word. Machines without an FPA display the message "no FPA".

1. The command `$X` is similar to `$x`, but displays the FPA registers `fpa16` through `fpa31` instead of `fpa0` through `fpa15`. This is done as a separate command because `adb` cannot display the contents of all FPA registers in a single standard-size window.
2. The command `$R` displays the contents of the data and control registers of the standard MC68881 floating-point coprocessor.

A.2. Examples of FPA Disassembly

As an example, consider the following assembly source fragment:

```
% cat foo.s
foo:
fpadds  d0,fpa0
fpadds@0    d0,fpa0
fpadds@5    d0,fpa0
%
```

On machines without an FPA, the default mode is to disassemble all FPA instructions as moves. For the example program, the following output is produced (except the parenthesized comments added here for explanation):

```
% as foo.s -o foo.o
% adb foo.o
<F=d
    0          (default value of 'F' on a machine without FPA)
foo?ia
foo:        movl    d0,0xe0000380    (normal disassembly)
```

FPA disassembly can be enabled by setting the debugger variable `F` to 1. For example:

```
% adb foo.o
1>F
<F=d
    1          (new value of 'F')
foo?ia
foo:        fpadds  d0,fpa0      (FPA disassembly)
```

On machines with an FPA, FPA disassembly is on by default, so the above output is produced without having to set the value of `F`.

FPA instructions may address the FPA using a base register in `[a0..a7]`. In practice, only `[a0..a5]` are used by the compilers.

`adb` does not know which register (if any) is being used to address the FPA in a given sequence of machine code. However, another debugger variable (`B`) may

be set by the user to designate a register as an FPA base register. By default, this variable has the value −1, which means that no register should be assumed to point to the FPA, so only instructions that access the FPA using absolute addressing are recognized as FPA instructions.

For the example program, a machine with an FPA produces the following output:

```
% adb foo.o
<F=d
    1          (default value of 'F' on a machine with FPA)
<B=d
   -1          (default value of 'B')
foo,3?ia
foo:           fpadds   d0,fpa0          (FPA disassembly)
0x6:           movl     d0,a0@(0x380)    (normal disassembly)
0xa:           movl     d0,a5@(0x380)    (normal disassembly)
0xe:
```

Note that the second and third instructions are still disassembled as moves, since `adb` cannot assume that they access the FPA. Continuing this example, if the FPA base register number is set to 5, the following output is produced:

```
% adb foo.o
5>B
<B=d
    5
foo,3?ia
foo:           fpadds   d0,fpa0          (FPA disassembly)
0x6:           movl  d0,a0@(0x380)       (normal disassembly)
0xa:           fpadds@5  d0,fpa0         (FPA disassembly)
0xe:
```

Note that the second instruction is still disassembled as a move, since `a5`, the register designated as the FPA base, is not used in it.

A.3. Examples of FPA Register Use

FPA data registers can be displayed using a syntax similar to that used for the MC68881 coprocessor registers. Note that unlike the MC68881 registers, FPA registers may contain either single-precision (32-bit) or double-precision (64-bit) values; MC68881 registers always contain extended-precision (96-bit) values.

For example, if `fpa0` contains the value 2.718282, we may display it as follows:

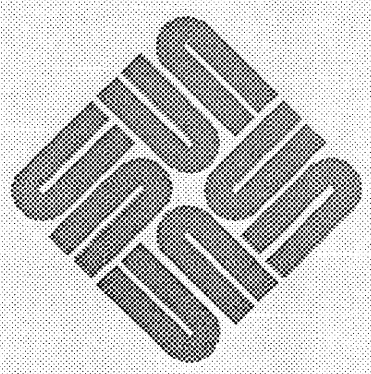
```
<fpa0=f
          fpa0      0x402df855     +2.718282e+00
```

Note that the value is displayed in hexadecimal as well as in floating-point notation. Unfortunately, an FPA register can only be set to a hexadecimal value. To set `fpa0` to 1.0, for example, you must know that this is represented as `0x3f800000` in IEEE single-precision format:

```
0x3f800000>fpa0
<fpa0=X
        3f800000
<fpa0=f
        +1.0000000e+00
```

B

dbx and dbxtool Changes



B

dbx and dbxtool Changes

B.1. Changes in Release 3.1

Release of the Floating-Point Accelerator (FPA) required some changes to `dbx` and `dbxtool`, in order to support debugging of programs using the FPA. Here are the changes made to `dbx` in Release 3.1:

1. There is a new `fpaasm` debugger variable to control disassembly of FPA instructions. This variable may be set or displayed using the `dbxenv` command, for which the syntax is:

```
dbxenv fpaasm [on|off]
```

All FPA instructions are disassembled as moves if the value of `fpaasm` is `off`. FPA instructions are disassembled with FPA assembly language mnemonics if the value is `on`. Defaults: on a machine with an FPA, `fpaasm` is initially set to `on`; on machines without an FPA, it is initially set to `off`.

1. The `fpabase` debugger variable has been added. It designates an MC68020 address register for FPA instructions that use base+short displacement addressing to address the FPA. The syntax is:

```
dbxenv fpabase [a[0-7]|off]
```

If FPA disassembly is disabled (if `fpaasm` is `off`) its value is ignored. Otherwise, its value is interpreted as follows:

value in `a0..a7`:
: Long move instructions that use the designated address register in base+short displacement mode address the FPA, and are disassembled using FPA assembler mnemonics. Note that this is independent of the actual run-time value of the register.

value = `off`:
: All based-mode FPA instructions are disassembled and single-stepped as move instructions.

The default value of `fpabase` is `off`, which designates no FPA base register.

1. The FPA registers `$fpa0..$fpa31` are recognized and can be used in arithmetic expressions or modified in `set` commands. This extension only

applies to machines with FPA's. Note that if an FPA register is used in an expression or assignment, its type is assumed to be double precision.

2. FPA registers can be displayed in single precision using the `/f` display format. Double-precision values are displayed using the `/F` display format.

B.2. Example of FPA Disassembly

Consider the following FORTRAN program:

```
program example
print *,f(1.0,1.0)
end

function f(x,y)
f = atan(x/y)
return
end
```

Assume that this program has been compiled with the `-g` option into the file `example`. On a machine with an FPA, we could disassemble the function `f` as shown below. Note that the FORTRAN intrinsic `ATAN` is directly supported by the FPA instruction set and compiler.

```
% dbx a.out
(dbx) stop in f
(1) stop in f
(dbx) run
Running: a.out
stopped in f at line 5 in file "example.f"
    5           f = atan(x/y)
(dbx) &$pc/8i
f+0x12:         movl    a6@(0xc),a0
f+0x16:         fpmoves a0@,fpa0
f+0x1c:         movl    a6@(0x8),a0
f+0x20:         fprdivs a0@,fpa0
f+0x26:         fpmoves fpa0,a6@(-0xc)
f+0x2e:         fpmoves a6@(-0xc),fpa1
f+0x36:         fpatans fpa1,fpa1
f+0x40:         fpmoves fpa1,a6@(-0x8)
        ...
```

FPA disassembly can be disabled by setting the debugger variable `fpaasm` to `off`. This causes `dbx` to disassemble FPA instructions as long moves to addresses on the FPA page:

```
(dbx) dbxenv fpaasm off
(dbx) &f+0x12/10i
f+0x12:          movl    a6@(0xc),a0
f+0x16:          movl    a0@,0xe0000c00:l
f+0x1c:          movl    a6@(0x8),a0
f+0x20:          movl    a0@,0xe0000600:l
f+0x26:          movl    0xe0000e00:l,a6@(-0xc)
f+0x2e:          movl    a6@(-0xc),0xe0000c08:l
f+0x36:          movl    #0x41,0xe0000818:l
f+0x40:          movl    0xe0000e08:l,a6@(-0x8)
```

When tracing a more complex program, one may occasionally want to step into a routine that has been compiled with optimization on. In such routines, the compiled code often addresses the FPA page by using base+short offset addressing. Such code can be difficult to recognize unless it is known ahead of time that a particular address register is being used to address the FPA. This situation can be identified by the presence of an instruction that loads the address of the FPA page (0xe0000000) into an address register before doing any floating-point arithmetic.

For example, here is a disassembly of the beginning of an optimized FORTRAN routine compiled with the `-O` and `-ffpa` options:

```
(dbx) &ddot_/7i
ddot_:       link    a6,#-0x2a0
ddot_+0x4:   moveml  #<d2,d3,d4,d5,d6,d7,a2,a3,a4,a5>,sp@
ddot_+0x8:   lea     e0000000:l,a2
ddot_+0xe:   movl    a2@(0xe20),a6@(-0x278)
ddot_+0x14:  movl    a2@(0xe24),a6@(-0x274)
ddot_+0x1a:  movl    a2@(0xe28),a6@(-0x270)
ddot_+0x20:  movl    a2@(0xe2c),a6@(-0x26c)
```

`dbx` does not know which register (if any) is being used to address the FPA in a given sequence of machine code. However, you may set the `dbxenv` variable `fpabase` to designate an MC68020 address register as the FPA base register. In this example, we note that the compiler has loaded the address of the FPA page into register `a2`, and we then designate `a2` as the FPA base register to obtain the following:

```
(dbx) dbxenv fpabase a2
(dbx) &ddot_/7i
ddot_:       link    a6,#-0x2a0
ddot_+0x4:   moveml  #<d2,d3,d4,d5,d6,d7,a2,a3,a4,a5>,sp@
ddot_+0x8:   lea     e0000000:l,a2
ddot_+0xe:   fpmoved@2       fpa4,a6@(-0x278)
ddot_+0x1a:  fpmoved@2       fpa5,a6@(-0x270)
ddot_+0x26:  fpmoved@2       204ce:l,fpa5
ddot_+0x36:  fpmoved@2       204ce:l,fpa4
```

B.3. Examples of FPA Register Use

FPA data registers can be displayed using a syntax similar to that used for the MC68881 coprocessor registers. Note that unlike the MC68881 registers, FPA registers may contain either single-precision (32-bit) or double-precision (64-bit) values; MC68881 registers always contain extended-precision (96-bit) values.

For example, if `fpa0` contains the single-precision value 2.718282, we may display it as follows:

```
(dbx) &$fpa0/f
fpa3      0x402df855     +2.718282e+00
```

Note that the value is displayed in hexadecimal as well as in floating-point notation.

A double-precision value may be displayed using the `/F` format. For example, if `fpa0` contains the double-precision value 2.718281828, we may display it as follows:

```
(dbx) &$fpa0/F
fpa0      0x4005bf0a 0x8b04919b     +2.71828182800000e+00
```

Note that it is important to use the correct display format; attempting to display a double-precision value in single precision will usually produce meaningless results.

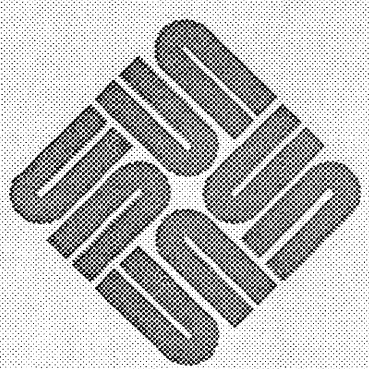
FPA registers can also be used in `set` commands and in arithmetic expressions. Since `dbx` cannot tell whether the value in an FPA register is single- or double-precision, `dbx` provides two sets of names to refer to FPA registers. The names `$fpa0..$fpa31` always cause the contents of the register to be interpreted as a double-precision value; the names `$fpa0s..$fpa31s` cause interpretation as a single-precision value. Thus, the commands

```
(dbx) set $fpa0s = 1.0
(dbx) set $fpa0 = 1.0
```

store different values in `fpa0`.

C

FPA Assembler Syntax



C

FPA Assembler Syntax

This appendix describes the Floating-Point Accelerator (FPA) support extensions to `as` included in Sun software release 3.1.

The extensions to `as` are described in general, with discussions of two-, three-, and four-operand instruction examples. Some instructions covered separately don't follow the formats described at the beginning of the appendix. The appendix includes restrictions and potential errors, followed by a summary of supported floating-point instructions.

C.1. Instruction Syntax

The general format for floating-point instructions is

```
fpopt@A     operands
```

where

`fp` indicates an FPA instruction.

op is the opcode name.

t is the operand type, either single (s) or double (d).

The *@A* part of the instruction is optional. When present, *A* specifies the address register which contains the base address for the FPA and can be in the range 0..7. If this form is used, a previous instruction must load the FPA address (0xe0000000) into the specified address register.

If *@A* is not present, then absolute long addressing is used to refer to the FPA. This form is more efficient for short routines.

Depending on the instruction, there may be from zero to four operands specified. The operands can be any of the following forms:

- Any MC68020 effective address, with the exception that absolute short addresses are not allowed for double-precision values.
- If either of the data register or the address register is used to hold a double-precision value, then the value will be in a register pair and both registers, separated by a colon, must be specified in the instruction. For example:

```
fpaddd   d0:d1, fpa0
```

The only exception to this rule is the `fpltod` instruction (convert integer to double-precision value).

- In some instructions (command register type) it is possible to specify that the register is in constant RAM. The syntax used for this case is `%n`, where `n` is a register number in the range 0 to 511.

C.2. Register Syntax

The 32 floating-point data registers are designated `fpa0, fpa1, ..., fpa31`. The supported control registers are:

Hardware	*Software*
MODE3_0	fpamode
WSTATUS	fpastatus

C.3. Operand Types

`as` supports three floating-point operand types:

- `s` for single-precision operands.
- `d` for double-precision operands.
- `l` for 32-bit integer operands, used for integer to floating-point conversions.

C.4. Two-Operand Instructions

Opcodes such as add, subtract, multiply, divide, negate, absolute value, square root, conversion from integer to floating-point, conversion from single to double (and vice versa) are all represented as:

```
fpopt  X, fpan
```

where *t*= `s` or `d`, and *X* is any valid MC68020 effective address for an operand or is an FPA data register.

If *X* is an FPA register which is in the constant RAM, then it can be in the range 0 to 511. If it is not in constant RAM, then it is one of the 32 FPA data registers. When *X* is an FPA register, then `fpa`*n* is one of the 32 floating-point data registers. If *X* is an effective address, then `fpa`*n* is one of the FPA registers in the range 0 to 15. The following are examples of such instructions:

```
            Instruction                          Computes

fpnegs    <effective address>, fpa1
fpsqrd    <effective address>, fpa2
fpsubs    fpa1, fpa2                    fpa2 ← fpa2 - fpa1
fprsubs   fpa1, fpa2                    fpa2 ← fpa1 - fpa2
fpdivs    d0, fpa2                      fpa2 ← fpa2 / d0
fprdivs   d0, fpa2                      fpa2 ← d0 / fpa2
```

In the above examples `fprsubs` and `fprdivs` are the reverse subtract and reverse divide operators, respectively.

The opcodes for `sine, cosine, atan, e^x, e^x -1, ln(x), ln(1+x)` and `sincos(x)` are all supported as command register type instructions:

```
fpopt  fpax, fpan
```

where *t* = `s` or `d`.

`fpax` is either a floating-point register or a register in the constant RAM (which is specified as `%`number). For the `sincos` instruction, the destination operand is actually a register pair:

```
fpsincost  fpax, fpac:fpas
```

where `fpac` is the cosine's destination and `fpas` is the sine's destination.

C.5. Three-Operand Instructions

The opcodes +, -, *, / are supported in extended and command register forms as

```
fpop3t    X, fpam, fpan
```

where *t* = `s` or `d` and *X* is an <effective address>for an extended instruction or a floating-point register for a command register type of instruction.

In the *command register form*, *X* and `fpam` can indicate a register number in the constant RAM. That is, they can either be in the range 0 to 511 or in the range 0 to 31. In the *extended instruction form*, `fpam` and `fpan` must be in the range 0 to 15. In the above format the position of *X* and `fpam` can be exchanged for the commutative operators add and multiply (the result of the operation remains the same).

For example,

```
fpa2 ← <effective address> + fpa1
```

can be represented by either of the following forms:

```
    fpadd3s  <effective address>, fpa1, fpa2
    fpadd3s  fpa1, <effective address>, fpa2
```

The same rule applies to subtract and divide operations. However, they are not commutative, so different answers result from each order. For example,

```
fpa2 ← fpa1 - <effective address>
```

must be coded as:

```
    fpsub3s  <effective address>, fpa1, fpa2
```

whereas

```
fpa2 ← <effective address> - fpa1
```

must be coded as:

```
    fpsub3s  fpa1, <effective address>, fpa2
```

Following the same format,

```
fpa3 ← fpa2 - fpa1
```

must be coded as:

```
    fpsub3s  fpa1, fpa2, fpa3
```

C.6. Four-Operand Instructions

In the extended and command register formats there are pivot instructions of the form:

```
fpopt  X, fpax, fpay, fpan
```

where `fpa`*n* is the destination floating-point data register and *t* = `s` or `d`, and *X* is an effective address or a floating-point register.

In the extended form, the positions of *X* and `fpa`*y* can be exchanged for both single- and double-precision types of instructions. In single-precision extended form, it is possible for two of the four operands to be effective addresses. This is in general either the first and third or the second and third operands.

In the command register form, `fpa`*x* and `fpa`*y* can be replaced by `%`*x* and `%`*y* indicating register numbers *x* and *y* in the constant RAM.

For four-operand instructions, `fpa`*x*, `fpa`*y* and `fpa`*n* can each be in the range 0 to 15, when *X* is an effective address. If *X* is an FPA register, then *X* and `fpa`*n* must be in the range 0 to 31 and `fpa`*x* and `fpa`*y* can either be in the range 0 to 511 (designating a location in constant RAM) or else in the range 0 to 31.

These pivot instructions are rather complicated and will be dealt with completely. The following shows the forms of each operation, the assembly code equivalent to each form, a generalization of the assembly instruction and the sequence of operations equivalent to the pivot instruction.

Instruction		Meaning
fpma{s,d}	<effective address>, reg2, reg3, reg1	reg1 ← reg3 + (reg2 * operand)
fpma{s,d}	reg2, reg3, <effective address>, reg1	reg1 ← operand + (reg3 * reg2)
fpma{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 + (reg2 * reg4)
fpmas	<ea1>, reg2, <ea2>, reg1	reg1 ← operand2 + (reg2 * operand1)

The fpma instruction, where m stands for multiply, and a stands for add, can be generalized as

```
fpmat X, fpax, fpay, fpan
```

where *t* is s or d, and *X* is an <effective address>or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and fpa*y* can be exchanged. Also, for single precision either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpmas   d0, fpa1, fpa2, fpa3
```

is equivalent to the following sequence of instructions

```
fpmul3s     d0, fpa1, temp
fpadd3s     temp, fpa2, temp
fpmoves     temp, fpa3
```

where temp is a temporary register.

Instruction		Meaning
fpms{s,d}	<effective address>, reg2, reg3, reg1	reg1 ← reg3 - (reg2 * operand)
fpms{s,d}	reg2, reg3, <effective address>, reg1	reg1 ← operand - (reg3 * reg2)
fpms{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 - (reg2 * reg4)
fpmss	<ea1>, reg2, <ea2>, reg1	reg1 ← operand2 - (reg2 * operand1)

The fpms instruction, where m stands for multiply, and s stands for subtract, can be generalized as

```
fpmst   X, fpax, fpay, fpan
```

where *t* is s or d, and *X* is an <effective address> or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and fpa*y* can be exchanged. Also, in single-precision two-memory instructions, either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpmss  fpa1, fpa2, d0, fpa3
```

is equivalent to the following sequence of instructions

```
fpmul3s     fpa1, fpa2, temp
fpsub3s     temp, d0, temp
fpmoves     temp, fpa3
```

The `fpmr` instruction, where `m` stands for multiply, and `r` stands for reverse subtract, can be generalized as

```
fpmrt    X, fpax, fpay, fpan
```

where *t* is `s` or `d`, and *X* is an <effective address> or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and `fpay` can be exchanged.

Instruction	Meaning
`fpmr{s,d}  <effective address>, reg2, reg3, reg1`	reg1 ← (-reg3) + (reg2 * operand)
`fpmr{s,d}  reg2, reg3, <effective address>, reg1`	reg1 ← (-operand) + (reg3 * reg2)
`fpmr{s,d}  reg4, reg2, reg3, reg1`	reg1 ← (-reg3) + (reg2 * reg4)
`fpmrs  <ea1>, reg2, <ea2>, reg1`	reg1 ← (-operand2) + (reg2 * operand1)

In single-precision extended form either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpmrs   d0, fpa1, fpa2, fpa3
```

is equivalent to the following sequence of instructions:

```
fpmul3s     d0, fpa1, temp
fpsub3s     fpa2, temp, temp
fpmoves     temp, fpa3
```

The `fpam` instruction, where `a` stands for add, and `m` stands for multiply, can be generalized as

```
fpamt    X, fpax, fpay, fpan
```

where *t* is `s` or `d`, and *X* is an <effective address> or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and `fpay` can be exchanged.

Instruction		Meaning
fpam{s,d}	<effective address>, reg2, reg3, reg1	reg1 ← reg3 * (reg2 + operand)
fpam{s,d}	reg2, reg3, <effective address>, reg1	reg1 ← operand * (reg3 + reg2)
fpam{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 * (reg2 + reg4)
fpams	<ea1>, reg2, <ea2>, reg1	reg1 ← operand2 * (reg2 + operand1)

In single-precision two-memory instructions, either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpams    fpa1, fpa2, fpa3, fpa4
```

is equivalent to the following sequence of instructions:

```
fpadd3s     fpa1, fpa2, temp
fpmul3s     temp, fpa3, temp
fpmoves     temp, fpa4
```

The `fpsm` instruction, where `s` stands for subtract, and `m` stands for multiply, can be generalized as

```
fpsmt    X, fpax, fpay, fpan
```

where *t* is `s` or `d`, and *X* is an effective address or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and `fpay` can be exchanged. The special cases for single-precision instructions are that either the first and third operands or the second and third operands can be effective addresses.

Instruction		Meaning
fpsm{s,d}	<effective address>, reg2, reg3, reg1	reg1 ← reg3 * (reg2 - operand)
fpsm{s,d}	reg2, reg3, <effective address>, reg1	reg1 ← operand * (reg3 - reg2)
fpsm{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 * (reg2 - reg4)
fpsm{s,d}	reg2, <effective address>, reg3, reg1	reg1 ← reg3 * (-reg2 + operand)
fpsm{s,d}	reg2, reg4, reg3, reg1	reg1 ← reg3 * (-reg2 + reg4)
fpsms	<ea1>, reg2, <ea2>, reg1	reg1 ← operand2 * (reg2 - operand1)
fpsms	reg2, <ea1>, <ea2>, reg1	reg1 ← operand2 * (-reg2 + operand1)

Note that, for example,

```
fpsms  d0, fpa1, fpa2, fpa3
```

is equivalent to the following sequence of instructions:

```
fpsub3s  d0, fpa1, temp
fpmul3s  temp, fpa2, temp
fpmoves  temp, fpa3
```

C.7. Other Instructions

Other special instructions are listed below. In each of them the last operand is also the destination, except for `tst`, `cmp` and `mcmp` where `fpastatus` is the implied destination. *X* is either an effective address or an FPA data register and *t* is either `s` or `d` for all instructions except `fpmove`*t*, where *t* can be `s`, `d`, or `l`.

Table C-1 *Other Instructions*

Mnemonic	*Syntax*	*Operation Name*
fpnop		nop
fptst*t*	X	operand compare with zero
fpcmp*t* fpmcmp*t*	X, fpa*m* X, fpa*m*	register *m* compare with operand register *m* compare magnitude with operand
fpmove*t* fpmove2*t* fpmove3*t* fpmove4*t*	fpa*m*, fpa*n* fpa*m*, fpa*n* fpa*m*, fpa*n* fpa*m*, fpa*n*	move floating-point registers 2x2 matrix move 3x3 matrix move 4x4 matrix move
fpdot2*t*	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*) * (fpa*y*+*1*)
fpdot3*t*	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*) * (fpa*y*+*1*) + (fpa*x*+*2*) * (fpa*y*+*2*)
fpdot4*t*	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*)*(fpa*y*+*1*) + (fpa*x*+*2*)*(fpa*y*+*2*) + (fpa*x*+*3*)*(fpa*y*+*3*)
fptran2*t* fptran3*t* fptran4*t*	fpa*m*, fpa*n* fpa*m*, fpa*n* fpa*m*, fpa*n*	transpose 2x2 matrix transpose 3x3 matrix transpose 4x4 matrix
fpmove fpmove fpmove fpmove fpmove*t* fpmove*t*	fpamode, <ea> <ea>, fpamode fpastatus, <ea> <ea>, fpastatus fpa*m*, <ea> <ea>, fpa*n*	read mode register write to mode register read status register write to status register read a floating-point data register write to a floating-point data register

C.8. Restrictions and Errors

`as` reports an invalid operand error in double-precision instructions when absolute short addressing or a single data or address register is used.

`as` reports a register out of range error for the dot product and matrix move and transpose instructions when the register specified does not fall within the specified range.

For most instructions where one operand is an effective address, the register range is 0 to 15. If all operands are FPA registers, then the register range is 0 to 31. For constant RAM registers, the range is 0 to 511. `as` reports an invalid operand error when any of these registers are not within the permitted range.

C.9. Instruction Set Summary

In the following table, `X` is any valid FPA register or MC68020 effective address (the form `(xxx):w` is not allowed for double). In some three- or four-address instructions the position of the `X` and one of the FPA registers can be exchanged as shown in the fourth column of the following table.

Table C-2 *Floating-Point Instructions*

Instruction	*Syntax*	*Operation*	*Alternative*
fpnegs	X, fp*a*n	negate single	
fpnegd	X, fp*a*n	negate double	
fpabss	X, fp*a*n	absolute value single	
fpabsd	X, fp*a*n	absolute value double	
fpltos	X, fp*a*n	convert integer to single	
fpltod	X, fp*a*n	convert integer to double	
fpstol	X, fp*a*n	convert single to integer	
fpdtol	X, fp*a*n	convert double to integer	
fpstod	X, fp*a*n	convert single to double	
fpdtos	X, fp*a*n	convert double to single	
fpsqrs	X, fp*a*n	square single	
fpsqrd	X, fp*a*n	square double	
fpadds	X, fp*a*n	add single	
fpadd3s	X, fp*a*m, fp*a*n	add single	fp*a*m, X, fp*a*n
fpaddd	X, fp*a*n	add double	
fpadd3d	X, fp*a*m, fp*a*n	add double	fp*a*m, X, fp*a*n
fpsubs	X, fp*a*n	subtract single	
fpsub3s	X, fp*a*m, fp*a*n	subtract single	fp*a*m, X, fp*a*n
fprsubs	<ea>, fp*a*n	reverse subtract single	
fpsubd	X, fp*a*n	subtract double	
fpsub3d	X, fp*a*m, fp*a*n	subtract double	fp*a*m, X, fp*a*n

Table C-2 *Floating-Point Instructions— Continued*

Instruction	*Syntax*	*Operation*	*Alternative*
fprsubd	<ea>, fpa*n*	reverse subtract double	
fpmuls	X, fpa*n*	multiply single	
fpmul3s	X, fpa*m*, fpa*n*	multiply single	fpa*m*, X, fpa*n*
fpmuld	X, fpa*n*	multiply double	
fpmul3d	X, fpa*m*, fpa*n*	multiply double	fpa*m*, X, fpa*n*
fpdivs	X, fpa*n*	divide single	
fpdiv3s	X, fpa*m*, fpa*n*	divide single	fpa*m*, X, fpa*n*
fprdivs	<ea>, fpa*n*	reverse divide single	
fpdivd	X, fpa*n*	divide double	
fpdiv3d	X, fpa*m*, fpa*n*	divide double	fpa*m*, X, fpa*n*
fprdivd	<ea>, fpa*n*	reverse divide double	
fpnop		nop	
fptsts	X	single compare with 0	
fptstd	X	double compare with 0	
fpcmps	X, fpa*m*	single compare	
fpcmpd	X, fpa*m*	double compare	
fpmcmps	X, fpa*m*	single magnitude compare	
fpmcmpd	X, fpa*m*	double magnitude compare	
fpsins	fpa*x*, fpa*n*	sine single	
fpsind	fpa*x*, fpa*n*	sine double	
fpcoss	fpa*x*, fpa*n*	cosine single	
fpcosd	fpa*x*, fpa*n*	cosine double	
fpatans	fpa*x*, fpa*n*	atan single	
fpatand	fpa*x*, fpa*n*	atan double	
fpetoxs	fpa*x*, fpa*n*	e^x single	
fpetoxd	fpa*x*, fpa*n*	e^x double	
fpetoxm1s	fpa*x*, fpa*n*	e^x-1 single	
fpetoxm1d	fpa*x*, fpa*n*	e^x-1 double	
fplogns	fpa*x*, fpa*n*	ln(x) single	
fplognd	fpa*x*, fpa*n*	ln(x) double	
fplognp1s	fpa*x*, fpa*n*	ln(1+x) single	
fplognp1d	fpa*x*, fpa*n*	ln(1+x) double	
fpsincoss	fpa*x*, fpa*c*:fpa*s*	fpa*c* ← cosine(x), fpa*s* ← sine(x)	
fpsincosd	fpa*x*, fpa*c*:fpa*s*	fpa*c* ← cosine(x), fpa*s* ← sine(x)	
fpmas	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* * X) + fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n* X, fpa*x*, X, fpa*n* fpa*x*, X, X, fpa*n*
fpmad	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* * X) + fpay	

Table C-2 *Floating-Point Instructions— Continued*

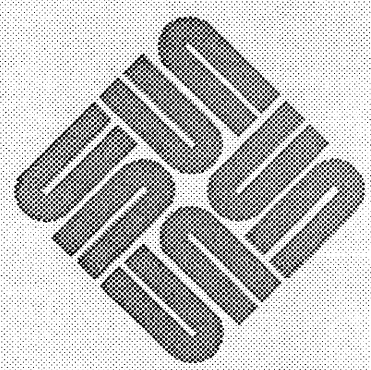
Instruction	*Syntax*	*Operation*	*Alternative*
fpmss	X, fpa*x*, fpay, fpa*n*	fpa*n* ← fpay - (fpa*x* * X)	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n*
fpmsd	X, fpa*x*, fpay, fpa*n*	fpa*n* ← fpay - (fpa*x* * X)	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n* X, fpa*x*, X, fpa*n* fpa*x*, X, X, fpa*n*
fpmrs	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* * X) - fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n*
fpmrd	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* * X) - fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n* X, fpa*x*, X, fpa*n* fpa*x*, X, X, fpa*n*
fpams	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* + X) * fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n*
fpamd	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* + X) * fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n* X, fpa*x*, X, fpa*n* fpa*x*, X, X, fpa*n*
fpsms	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* - X) * fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n*
fpsmd	X, fpa*x*, fpay, fpa*n*	fpa*n* ← (fpa*x* - X) * fpay	fpa*x*, X, fpay, fpa*n* fpay, fpa*x*, X, fpa*n* X, fpa*x*, X, fpa*n* fpa*x*, X, X, fpa*n*
fpmoves	<ea>, fpa*n*	write to a register, single	
fpmoved	<ea>, fpa*n*	write to a register, double	
fpmovel	<ea>, fpa*n*	write to a register, integer	
fpmoves	fpa*m*, <ea>	read a register, single	
fpmoved	fpa*m*, <ea>	read a register, double	
fpmove2s	fpa*m*, fpa*n*	2x2 matrix move, single	
fpmove2d	fpa*m*, fpa*n*	2x2 matrix move, double	
fpmove3s	fpa*m*, fpa*n*	3x3 matrix move, single	
fpmove3d	fpa*m*, fpa*n*	3x3 matrix move, double	
fpmove4s	fpa*m*, fpa*n*	4x4 matrix move, single	
fpmove4d	fpa*m*, fpa*n*	4x4 matrix move, double	

Table C-2 *Floating-Point Instructions— Continued*

Instruction	*Syntax*	*Operation*	*Alternative*
fpdot2s	fpa*x*, fpay, fpa*n*	fpa*n* ← fpa*x**fpay + (fpa*x+1*) * (fpay+*1*)	
fpdot2d	fpa*x*, fpay, fpa*n*	fpa*n* ← fpa*x**fpay + (fpa*x+1*) * (fpay+*1*)	
fpdot3s	fpa*x*, fpay, fpa*n*	fpa*n* ← fpa*x**fpay + (fpa*x+1*) * (fpay+*1*) + (fpa*x+2*) * (fpay+*2*)	
fpdot3d	fpa*x*, fpay, fpa*n*	fpa*n* ← fpa*x**fpay + (fpa*x+1*) * (fpay+*1*) + (fpa*x+2*) * (fpay+*2*)	
fpdot4s	fpa*x*, fpay, fpa*n*	fpa*n* ← fpa*x**fpay + (fpa*x+1*)*(fpay+*1*) + (fpa*x+2*)*(fpay+*2*) + (fpa*x+3*)*(fpay+*3*)	
fpdot4d	fpa*x*, fpay, fpa*n*	fpa*n* ← fpa*x**fpay + (fpa*x+1*)*(fpay+*1*) + (fpa*x+2*)*(fpay+*2*) + (fpa*x+3*)*(fpay+*3*)	
fptran2s	fpa*m*, fpa*n*	transpose 2x2 matrix, single	
fptran2d	fpa*m*,fpa*n*	transpose 2x2 matrix, double	
fptran3s	fpa*m*, fpa*n*	transpose 3x3 matrix, single	
fptran3d	fpa*m*,fpa*n*	transpose 3x3 matrix, double	
fptran4s	fpa*m*, fpa*n*	transpose 4x4 matrix, single	
fptran4d	fpa*m*,fpa*n*	transpose 4x4 matrix, double	
fpmove	fpamode, <ea>	read the mode register	
fpmove	<ea>, fpamode	write on mode register	
fpmove	fpastatus, <ea>	read the status register	
fpmove	<ea>, fpastatus	write to status register	

D

IEEE Appendix Functions



D

IEEE Appendix Functions

The IEEE Standard for Binary Floating-Point Arithmetic includes an appendix of useful functions which are not usually available in higher-level languages. The IEEE test vectors measure compliance with the specification of some of those functions. The following paragraphs give examples of double-precision FORTRAN and assembly language routines that implement those functions in a way that passes the IEEE test vectors under conditions described under "Conformance Benchmarks" in Chapter 4. The assembly language examples are coded to use switched floating-point. These functions, especially the assembly language ones that call V... and F... entry points in `libc`, may not work in future software releases. Efficiency was not a coding consideration.

Copysign:

```
        .globl  _copyd_         | Works with any kind of floating point!
_copyd_:
        movel   sp@(4),a0
        moveml  a0@,d0/d1       | d0 := x.
        movel   sp@(8),a0       | Address of y.
        tstb    a0@
        bmis    1f
        bclr    #31,d0
        bras    2f
1:
        bset    #31,d0
2:
        rts
```

Logb:

```
        .globl  _logbd_
_logbd_:
        movel   sp@(4),a0
        movel   a0@,d0
        jsr     Fexpod
        cmpw    #-0x3ff,d0
        beqs    3f
        cmpw    #0x400,d0
        beqs    2f
```

```
1:
        jsr     Vfltd
        rts
2:
        moveml  a0@,d0/d1
        bclr    #31,d0
        lea     dzero,a0
        jsr     Vaddd               | Add zero to catch signalling NaN.
        rts
3:
        moveml  a0@,d0/d1
        bclr    #31,d0
        tstl    d0
        bnes    5f
        tstl    d1
        beqs    4f
5:
        movel   #-0x3ff,d0
        bras    1b
4:
        moveml  dmone,d0/d1
        lea     dzero,a0
        jsr     Vdivd               | Set up -1/0 to generate divide by zero signal.
        rts

dzero:  .double 0r0
dmone:  .double 0r-1
```

Scalb:

```
        real*8 function scalbd ( x, y)
        real*8 x, y, scalid
        scalbd = scalid( x, int(y))
        end

        .globl  _scalid_
_scalid_:
        movel   sp@(4),a0
        moveml  a0@,d0/d1               | x.
        movel   sp@(8),a0               | Address of y.
        jsr     Vscaleid
        rts
```

Fraction part: The fraction part or significand function, not in the Standard, is nonetheless tested by an IEEE test vector.

```
        real*8 function signifd ( x )
        real*8 x, logbd, scalbd, s
        s = scalbd(x,-logbd(x))
        if ((abs(s) .gt. 0.0d0 ) .and. (abs(s) .lt. 1.0d0)) then
 1              s = s + s
                if (abs(s) .lt. 1.0d0) goto 1
        endif
        signifd = s
        end
```

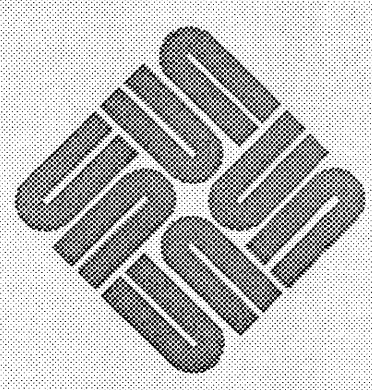
Nextafter:

```
    real*8 function nextd ( x, y)
    real*8 x, y, t, minnorm, maxnorm
    integer i(2)
    equivalence (i,t)
    parameter ( minnorm = 2.225073858507201383d-308)
    parameter ( maxnorm = 1.7976931348623157d+308)

    if ((y .ne. y) .or. (x .ne. x)) then
c       handle nans; convert signalling nans
        nextd = x + y
        return
    end if

    t = x
    if ( x .lt. y) then
             if (x .lt. 0) then
            i(2) = i(2) - 1
            if (i(2) .eq. -1) i(1) = i(1) - 1
        else if (x .gt. 0) then
            i(2) = i(2) + 1
            if (i(2) .eq. 0) i(1) = i(1) + 1
        else if (x .eq. 0) then
            i(2) = 1
            i(1) = 0
        endif
    else if ( x .gt. y) then
             if (x .lt. 0) then
            i(2) = i(2) + 1
            if (i(2) .eq. 0) i(1) = i(1) + 1
        else if (x .gt. 0) then
            i(2) = i(2) - 1
            if (i(2) .eq. -1) i(1) = i(1) - 1
        else if (x .eq. 0) then
            i(2) = 1
            i(1) = 0
            t = -t
```

```
        end if
    end if

c   It would have been preferable to use the following statements to
c   generate underflow and overflow signals, but optimization with -O
c   eliminates the statements as dead code.
c        if (x .ne. t) then
c                if (abs(t) .lt. minnorm) then
c       signal underflow without changing t
c                        z = minnorm * 0.1d0
c                else if (abs(t) .gt. maxnorm) then
c       signal overflow without changing t
c                        z = maxnorm * 2.0d0
c                end if

c   The following code accomplishes the same thing with more trouble,
c   but fails with -f68881 -O because of extended register allocation.
c   if (x .ne. t) then
c       if (abs(t) .lt. minnorm) then
c   signal underflow without changing t
c                        t = copyd(min(abs(t),4.0d0 * (minnorm/3.0d0)),t)
c                else if (abs(t) .gt. maxnorm) then
c       signal overflow without changing t
c                        t = copyd(max(abs(t),maxnorm + maxnorm),t)
c       end if

    if (x .ne. t) then
        if (abs(t) .lt. minnorm) then
c   signal underflow without changing t
                        call dummyd(minnorm * minnorm)
                else if (abs(t) .gt. maxnorm) then
c       signal overflow without changing t
                        call dummyd(maxnorm * maxnorm)
        end if
    end if
    nextd = t
    end

    subroutine dummyd( x )
    real*8 x
    end
```

E

SPICE Input Files



E

SPICE Input Files

E.1. tdo

The following input file models a tunnel diode oscillator; the transient analysis output calculation is extremely ill-conditioned, as discussed under "Different Numerical Results" in Chapter 2:

```
TDO - TUNNEL DIODE OSCILLATOR
 .WIDTH IN=72
VBIAS 0 2 -120MV
LS    2 1  2.5UH
CS    1 0  100PF
GTD   1 0  POLY(1)  1 0
+ -3.95510115972848E-17   +1.80727308405845E-01   -2.93646217292003E+00
+ +4.12669748472374E+01   -6.09649516869413E+02   +6.08207899870511E+03
+ -3.73459336478768E+04   +1.44146702315112E+05   -3.53021176453665E+05
+ +5.34093436084762E+05   -4.56234076434067E+05   +1.68527934888894E+05
 .DC VBIAS 0 -600MV -5MV
 .PLOT DC I(VBIAS) (0,5MA)
 .TRAN 5NS 500NS 0 5NS
 .PLOT TRAN V(1)
 .OPT ACCT LIST NODE LVLCOD=2
 .END
```

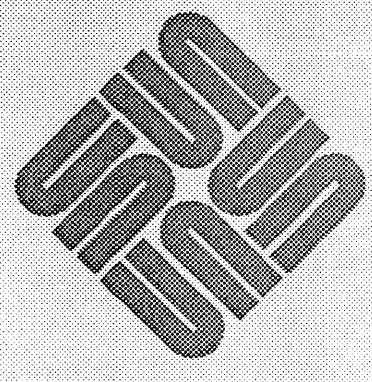
E.2. mosamp2

The following input file models an MOS amplifier, and is used to obtain the benchmark results discussed under "Performance Benchmarks" in Chapter 4:

```
mosamp2 - mos amplifier - transient
 .WIDTH OUT=80
 .OPTIONS ACCT ABSTOL=10N  VNTOL=10N
 .TRAN 0.1US 10US
M1  15 15  1 32 M W=88.9U  L=25.4U
M2   1  1  2 32 M W=12.7U  L=266.7U
M3   2  2 30 32 M W=88.9U  L=25.4U
M4  15  5  4 32 M W=12.7U  L=106.7U
M5   4  4 30 32 M W=88.9U  L=12.7U
M6  15 15  5 32 M W=44.5U  L=25.4U
M7   5 20  8 32 M W=482.6U L=12.7U
M8   8  2 30 32 M W=88.9U  L=25.4U
M9  15 15  6 32 M W=44.5U  L=25.4U
M10  6 21  8 32 M W=482.6U L=12.7U
M11 15  6  7 32 M W=12.7U  L=106.7U
M12  7  4 30 32 M W=88.9U  L=12.7U
M13 15 10  9 32 M W=139.7U L=12.7U
M14  9 11 30 32 M W=139.7U L=12.7U
M15 15 15 12 32 M W=12.7U  L=207.8U
M16 12 12 11 32 M W=54.1U  L=12.7U
M17 11 11 30 32 M W=54.1U  L=12.7U
M18 15 15 10 32 M W=12.7U  L=45.2U
M19 10 12 13 32 M W=270.5U L=12.7U
M20 13  7 30 32 M W=270.5U L=12.7U
M21 15 10 14 32 M W=254U   L=12.7U
M22 14 11 30 32 M W=241.3U L=12.7U
M23 15 20 16 32 M W=19U    L=38.1U
M24 16 14 30 32 M W=406.4U L=12.7U
M25 15 15 20 32 M W=38.1U  L=42.7U
M26 20 16 30 32 M W=381U   L=25.4U
M27 20 15 66 32 M W=22.9U  L=7.6U
CC 7 9 40PF
CL 66 0 70PF
VIN 21 0 PULSE(0 5 1NS 1NS 1NS 5US 10US)
VCCP 15 0 DC +15
VDDN 30 0 DC -15
VB 32 0 DC -20
 .MODEL M NMOS(NSUB=2.2E15 UO=575 UCRIT=49K UEXP=0.1 TOX=0.11U XJ=2.95U
+   LEVEL=2 CGSO=1.5N CGDO=1.5N CBD=4.5F CBS=4.5F LD=2.4485U NSS=3.2E10
+   KP=2E-5 PHI=0.6 )
 .PRINT TRAN V(20) V(66)
 .PLOT  TRAN V(20) V(66)
 .END
```

F

MC68881 Mask Differences



F

MC68881 Mask Differences

This appendix describes differences between the MC68881's A79J and later A93N masks. The A93N mask reflects the MC68881 described in Motorola's definitive *MC68881 Floating-Point Coprocessor User's Manual.*

You can run `mc68881version (8)` on a Sun-3 to determine the installed MC68881's mask and approximate clock rate.

Some of the problems listed below are worked around in software in Sun's Release 3.0 and later, sometimes at a cost in accuracy and performance, relative to A93N hardware and software without workarounds. Since one of the problems described below has *no* workaround, and future releases of Sun software may have other workarounds removed to improve accuracy and performance, it is a good idea for you to upgrade to the A93N-version MC68881 when it is available.

`fmove[sd] fpm,<ea>`

`fmove[sd] fpm,<ea>` to single- or double-precision destinations produces an incorrect result when the result underflows but rounds back up to the smallest normalized number. The result returned is zero instead of the smallest normalized number.

This problem might occasionally confound programs that depend on monotonicity: there are single- and double-precision `x > 0` such that the computed result of `x/2` is 0 but the computed result of `x/4` is > 0. There is no software workaround for this problem, which may, but probably does not, adversely affect any program in which underflow occurs. Underflow can be detected by enabling the underflow bit in the MC68881 Exception Enable Byte.

`fmovex fpm,<ea>`

`fmovex fpm,<ea>` to an extended-precision destination produces an incorrect result when the source operand is an extended-precision denormalized number. The result returned is zero, an incorrect denormalized number, or an incorrect, tiny normalized number.

The problem can be demonstrated by constructing an appropriate sequence of operations on double-precision operands but it is unlikely to be encountered unintentionally. There is no software workaround for this problem, but it is unlikely to affect any Sun-supplied software since the extended data type is not directly supported by Sun's compilers.

`flognt, flog2t, flog10t`

The `flognt, flog2t,` and `flog10t` instruction produces an excessive amount of error in the results for operands $1\pm\varepsilon$. As ε approaches 0, the results become totally dominated by error.

Sun's software releases 3.0 and later contain workarounds that avoid this problem for programs written in higher-level languages, as the assembly language entry `Mlogd` illustrates:

```
    fmoved  sp@,fp0        |sp@ contains the argument x.
    fcmps   #0r0.5,fp0
    fjule   1f             |Branch if x<=0.5 or x is Nan.
    fsubl   #1,fp0
    flognplx  fp0,fp0      |This is more accurate for x>0.5.
    bras  2f
1:
    flognx fp0,fp0         |This is more accurate for x<0.5.
2:
    fmoved  fp0,sp@        |sp@ receives the computed logn.
```

`fsqrtx fp`*m*`,fp`*n* (*m*<>*n*)

`fsqrtx fp`*m*`,fp`*n* (*m*<>*n*) can produce an incorrect result; the failure is dependent upon the contents of `fp`*n* before the `fsqrtx` executes. Therefore, this problem can be avoided in software by preceding the `fsqrtx` with an instruction which loads `fp`*n* with an innocuous value such as 2.0: precede the `fsqrtx` with `fmoveb #2,fp`*n* or an equivalent instruction.

`fsqrtp <ea>,fp`*n*

`fsqrtp <ea>,fp`*n* may cause the same problem, and may be avoided in the same way. Note that this problem can't occur for either register-to-register `fsqrtx fp`*m*`,fp`*m* or memory-to-register `fsqrt[sdxbwl] <ea>,fp`*n*.

Sun's software releases 3.0 and later contain workarounds that avoid this problem for programs written in higher-level languages.

Binary-to-Decimal Conversion

a. When the result is an exact power of 10, the packed BCD mantissa is $C.000....000 instead of $1.000...000 (note non-BCD digit).

b. The result will have the incorrect exponent sign when the decimal exponent is greater than +999.

c. When the "K" parameter is in the range 0 thru 17, the OPERR (IOP) exception is incorrectly signalled.

d. When the magnitude of the decimal exponent is greater than 999, the OVFL exception is incorrectly signalled (OPERR should be signalled).

e. When the source operand is an extended-precision denormalized number, the numeric value of the result is correct but the decimal rounding boundary is incorrect.

f. The inexact exception is set even when conversion is exact.

g. Conversions with a "K" parameter of 0 do not function as specified, the result is the same as K=+1.

Sun's releases 3.0 and later use the MC68881's binary-to-ASCII conversion instructions only in the `adb` and `dbx` debuggers. These problems are worked around there.

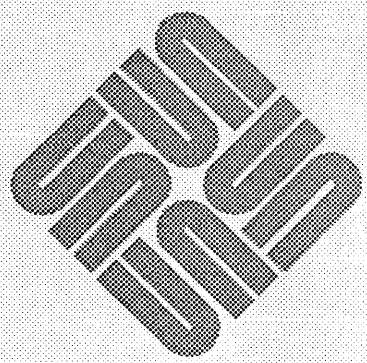
Decimal-to-Binary Conversion

`fopp <ea>, fp`*m* of some exact powers of ten contains an error of 1 bit.

These instructions are not used in Sun's releases 3.0 and later.

G

Assembly-Level In-line Expansion



G

Assembly-Level In-line Expansion

G.1. Introduction

A simple in-line expansion facility lets you integrate assembly routines into higher-level routines written in C, Pascal, or FORTRAN.

The peephole optimizer `c2` has been modified to optimize code sequences containing inline-expanded routines. In many cases, the resulting code is comparable in quality to that compiled for constructs directly supported in the common code generator.

In-line expansion of procedure calls is an important optimization strategy, primarily because it exposes opportunities for other optimizations. It is also useful as a "software Swiss Army knife" in applications that depend on routines written in assembly language, either for performance reasons or for access to machine instructions not otherwise available in high-level languages. Several such applications are described in the following sections.

Language-Specific Constructs

The constructs of a given language frequently cannot be supported in a common backend with a reasonably small operator set in multilingual environments based on a common code generator. For example, Pascal sets and strings, and FORTRAN complex numbers are not naturally supported in a compiler system based on the Portable C Compiler (`pcc`), which includes none of these types. Attempts to support all such language-specific constructs in a single code generator typically result in large, unwieldy code generators containing large amounts of code intended only for a specific language. In-line expansion of language-specific library routines presents an attractive alternative.

Access to Special Instructions

A programmer frequently requires access in system software to instructions not normally produced by the code generator. The usual technique is to call a hand-coded assembly routine which uses the desired instruction. Unfortunately, the procedure call and return overhead may be unacceptable if this occurs in a region of high execution frequency.

Register Allocation

As noted earlier, in-line expansion is an important optimization strategy, primarily becauses it exposes opportunities for improved register allocation and other optimizations. This is important in the Sun Workstation environment, for several reasons.

First, the Sun MC68000 family calling sequence requires that all argument values be passed on the stack. This incurs memory traffic that becomes unnecessary when the body of a called procedure is integrated into the caller's text.

Second, the calling sequence requires that all function results be returned in the main processor registers `d0` and `d1`. This is particularly annoying for functions returning floating-point values, since the calling routine cannot normally assume that the called routine uses a specific floating-point processor. Consequently, the calling sequence typically incurs a significant amount of traffic between the main processor registers and the floating-point processor registers. For simple operations, the cost of moving data in and out of external registers can easily exceed the cost of the operation itself. Experience shows that such operations are used frequently.

G.2. User Interface

In the C, FORTRAN, and Pascal compilers, filenames ending with the suffix "`.il`" are assumed to contain inline-expandable assembly routines. In addition, the Pascal compiler uses the same mechanism to expand standard procedures and functions.

Implementation

In-line expansion is divided into two tasks. The expansion itself is a straightforward text substitution, with little knowledge of the details of the assembly language. Most of the work of reducing calling sequence overhead is done by the peephole optimizer `c2`. This partitioning of tasks is convenient, since most of the information required is already present in the program representation built by `c2` for other optimizations.

For illustrative purposes, consider the FORTRAN function `cexp()` or complex exponential. Assuming the real functions `sin()`, `cos()`, and `exp()` as primitives, the `cexp()` function may be implemented by the following C routine:

```
typedef struct {
    double real,imag;
} dcomplex;

void c_exp(r, z)
    dcomplex *r, *z;
{
    register double expx;
    double exp(), cos(), sin();
    expx = exp(z->real);
    r->real = expx * cos(z->imag);
    r->imag = expx * sin(z->imag);
}
```

Straightforward compilation with MC68881 code generation (`-f68881`) enabled produces the following translation:

```
        .text
|#PROC# 04
        .globl  _c_exp
_c_exp:
        link    a6,#0
        addl    #-LF12,sp
        moveml  #LS12,sp@
        fmovem  #LSS12,a6@(-LFF12)
```

```
        movl    a6@(0xc),a0
        movl    a0@(0x4),sp@-
        movl    a0@,sp@-
        jbsr    _exp
        addqw   #0x8,sp
        movl    d1,sp@-
        movl    d0,sp@-
        fmoved  sp@+,fp7
        movl    a6@(0xc),a0
        movl    a0@(0xc),sp@-
        movl    a0@(0x8),sp@-
        jbsr    _cos
        addqw   #0x8,sp
        movl    d1,sp@-
        movl    d0,sp@-
        fmoved  sp@+,fp0
        fmulx   fp7,fp0
        movl    a6@(0x8),a0
        fmoved  fp0,a0@
        movl    a6@(0xc),a0
        movl    a0@(0xc),sp@-
        movl    a0@(0x8),sp@-
        jbsr    _sin
        addqw   #0x8,sp
        movl    d1,sp@-
        movl    d0,sp@-
        fmoved  sp@+,fp0
        fmulx   fp7,fp0
        movl    a6@(0x8),a0
        fmoved  fp0,a0@(0x8)
LE12:
        fmovem  a6@(-0xc),#0x1
        unlk    a6
        rts
        LF12 = 0
        LS12 = 0x0
        LFF12 = 12
        LSS12 = 0x1
        LP12 =  0x10
        .data
```

Note that most of this code is occupied with passing arguments on the stack, and moving function results from `d0/d1` to `fp0`. Even the latter involves stack traffic, since the MC68881 does not support direct moves between floating-point registers and register pairs on the main processor. Additional overhead (not shown) exists for similar reasons in the called library routines. In-line expansion can do much to alleviate such problems, as will be shown in the following sections.

G.3. In-line Expansion Pass

`Inline`, the program that actually does in-line expansion, is little more than a glorified *sed* script. It knows nothing about the syntax or semantics of the target machine assembly language, other than the form of a call instruction. `Inline` is invoked by the `cc`, `pc`, and `f77` compilers as

```
/usr/lib/inline [sourcefile] [-o outputfile] [-i inlinefile]
```

`Inline` expands call instructions in *sourcefile* using routine definitions from one or more *inlinefiles*. If no *sourcefile* is specified, the default source is `stdin`. If no *outputfile* is specified, the default output is `stdout`. If no *inlinefile* is specified, `inline` behaves essentially like */bin/cat*. Note that nested expansions are not supported, but may be implemented using pipes.

Each *inlinefile* contains one or more labeled assembly language routines of the form:

```
        .inline name,argsize
        ...
        instructions
        ...
        .end
```

where the *instructions* constitute an in-line expansion of the named routine, and *argsize* is the number of bytes of arguments expected. The routine must observe the following restrictions:

[1] Registers `a0-a1/d0-d1/fp0-fp1/fpa0-fpa3` may be used freely.

[2] Other registers must be saved on entry and restored on exit.

[3] Results are returned in `d0` or `d0/d1`.

[4] Arguments must be explicitly deleted from the stack. In general, this should be done using autoincrement addressing.

The optimizations performed in `c2` assume that in-line routines are coded in a way that makes the lifetimes of stack temporaries explicit. On the MC68000 family of processors, this can be done by using autoincrement addressing to pop incoming arguments from the stack. For the preceding example, on the MC68881 you could code double-precision versions of `sin()`, `cos()`, and `exp()` as follows:

```
        .inline _cos,8
        fcosd   sp@+,fp0
        fmoved  fp0,sp@-
        movl    sp@+,d0
        movl    sp@+,d1
        .end

        .inline _sin,8
        fsind   sp@+,fp0
        fmoved  fp0,sp@-
```

```
        movl    sp@+,d0
        movl    sp@+,d1
        .end

        .inline _exp,8
        fetoxd  sp@+,fp0
        fmoved  fp0,sp@-
        movl    sp@+,d0
        movl    sp@+,d1
        .end
```

When `inline` is run on the preceding compiled code using these routines, the result is:

```
        .text
        .globl  _c_exp
_c_exp:
        link    a6,#0
        addl    #-LF12,sp
        moveml  #LS12,sp@
        fmovem  #LSS12,a6@(-LFF12)
        movl    a6@(0xc),a0
        movl    a0@(0x4),sp@-
        movl    a0@,sp@-
        fetoxd  sp@+,fp0
        fmoved  fp0,sp@-
        movl    sp@+,d0
        movl    sp@+,d1
        subql   #8,sp
        addqw   #0x8,sp
        movl    d1,sp@-
        movl    d0,sp@-
        fmoved  sp@+,fp7
        movl    a6@(0xc),a0
        movl    a0@(0xc),sp@-
        movl    a0@(0x8),sp@-
        fcosd   sp@+,fp0
        fmoved  fp0,sp@-
        movl    sp@+,d0
        movl    sp@+,d1
        subql   #8,sp
        addqw   #0x8,sp
        movl    d1,sp@-
        movl    d0,sp@-
        fmoved  sp@+,fp0
        fmulx   fp7,fp0
        movl    a6@(0x8),a0
        fmoved  fp0,a0@
        movl    a6@(0xc),a0
        movl    a0@(0xc),sp@-
        movl    a0@(0x8),sp@-
        fsind   sp@+,fp0
```

```
        fmoved  fp0,sp@-
        movl    sp@+,d0
        movl    sp@+,d1
        subql   #8,sp
        addqw   #0x8,sp
        movl    d1,sp@-
        movl    d0,sp@-
        fmoved  sp@+,fp0
        fmulx   fp7,fp0
        movl    a6@(0x8),a0
        fmoved  fp0,a0@(0x8)
LE12:
        fmovem  a6@(-0xc),#0x1
        unlk    a6
        rts
```

This code looks terrible, but it has several desirable attributes.

1. It will execute correctly in its present form, and may even be slightly faster than the original code, although it is considerably larger.
2. All stack traffic is explicit, rather than being hidden in the semantics of the procedure call.
3. The patterns of stack overhead are regular and may be readily optimized by local transformations. In particular, no elaborate analysis is necessary to determine lifetimes of stack temporaries.

G.4. Peephole Optimizations

The symmetric use of autoincrement and autodecrement addressing is taken by `c2` to indicate the lifetime of the stack copy of an argument. In particular, if there are no other uses of the stack copy, the copy need not be created, and the argument may instead be loaded into a register or used directly in the source field of an instruction in the expanded routine. This is typical of a class of optimizations in `c2` which are similar to copy propagation, and generally attempt to reduce the height of the runtime stack. These optimizations depend on a program representation in which uses, definitions, and lifetimes of registers are explicit. For example, consider the following code pattern:

```
        (1) move  x,sp@-
        ...
        (2) <op>  sp@+,...
```

If possible, this pattern is rewritten as

```
        (1) (deleted)
        ...
        (2) <op>  x,...
```

This optimization is feasible only if *x* is not modified between (1) and (2). Other requirements include: side effects of (1) (including condition codes) must not be used, the value on the top of the stack must not be used, and the stack pointer

itself must not be used or set. If any of these restrictions are not met, the code is left unchanged.

Another similar pattern is the following:

```
(1) move  x,sp@-
    ....
(2) move  sp@+,rn
```

If possible, this pattern is rewritten as

```
(1) move  x,rn
...
(2) (deleted)
```

This optimization requires that the register *rn* be neither used nor set in the interim. Care must also be taken to leave the code unchanged if the condition codes are live after either instruction (1) or instruction (2).

Applying these and other `c2` optimizations to the example of the previous section yields the following:

```
        .text
        .globl  _c_exp
_c_exp:
        link    a6,#-12
        fmovex  fp7,a6@(-12)
        movl    a6@(12),a0
        fetoxd  a0@,fp7
        fcosd   a0@(8),fp0
        fmulx   fp7,fp0
        movl    a6@(8),a0
        fmoved  fp0,a0@
        movl    a6@(12),a0
        fsind   a0@(8),fp0
        fmulx   fp7,fp0
        movl    a6@(8),a0
        fmoved  fp0,a0@(8)
        fmovex  a6@(-12),fp7
        unlk    a6
        rts
```

Although this code is not optimal, the improvement over the initial version is substantial. In addition, the final code is smaller than that which would be obtained by optimizing the original unexpanded code. Thus a gain in speed is obtained without incurring an increase in code size.

G.5. Using Sun's Predefined `.il` Files

Starting with Release 3.1, Sun provides five inline expansion template files:

```
/usr/lib/fswitch.il
/usr/lib/fsoft.il
/usr/lib/fsky.il
/usr/lib/f68881.il
/usr/lib/ffpa.il
```

These files may be optionally used to replace calls to procedures in `libF77.a` and `libm.a` with code that will either execute faster or provide smaller executable files.

The procedure is simple; replace a command like

```
f77 -O -ffpa -c cx.f
```

with

```
f77 -O -ffpa -c cx.f /usr/lib/ffpa.il
```

Then any code generated to call routines redefined in `/usr/lib/ffpa.il` will be appropriately expanded inline. More than one `.il` file can be specified; in that case, the first definition encountered of an inline-expanded procedure will supersede any following definitions of the same procedure.

None of Sun's compilers use the `f*.il` files from `/usr/lib` except when specifically requested. `pc` always automatically uses another file `pc2.il` which doesn't affect floating-point subroutines.

Faster Execution

The `/usr/lib/f*.il` files' primary application is to accelerate calculations involving the complex and doublecomplex data types in FORTRAN. For `-ffpa`, `-f68881`, and `-fsky`, intensive complex arithmetic may be twice as fast with inline expansion.

Smaller Executable Files

Although inline expansion normally increases an executable file's size, it can decrease executable size by avoiding the need to link in all or most of switched floating-point support.

With `f77`, using `x**y`, `mod`, `atan2`, `cabs`, `nint`, and complex or doublecomplex arithmetic, can all provoke linking-in switched floating point even when `-fsky`, `-f68881`, or `-ffpa` is specified. Using `fsky.il`, `f68881.il`, or `ffpa.il` causes these calls to switched floating point to be replaced by inline code or calls to appropriate unswitched routines.

With `cc`, use of almost any of the functions defined in `<math.h>` invokes switched floating point, but the appropriate `.il` file undoes the damage. With `SPICE 3A.7`, for instance, the version compiled with

```
-O -ffpa /usr/lib/ffpa.il
```

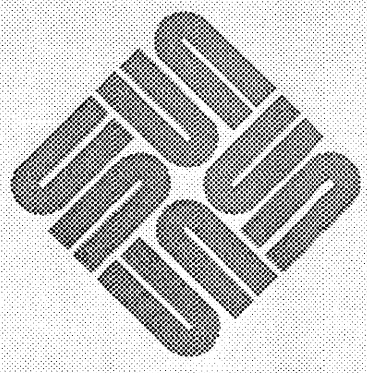
was 40,000 bytes smaller than the version compiled just with

```
-O -ffpa
```

Using the `.il` file also avoids certain overheads required for System V compatibility.

H

System V Interface Compliance



H

System V Interface Compliance

A joint goal of Sun and AT&T is to develop a version of UNIX that incorporates the best features of Berkeley BSD 4.2 and System V. Accordingly, Sun's Software Release 3.2 mathematical library `libm.a` and related files have been modified so that C programs compiled with the `-fsoft` floating point option or compiled with `-fswitch` and running without floating-point hardware better comply with the System V Interface Definition (SVID). Fortran and Pascal programs, and C programs running with floating-point hardware, are usually not affected. The differences primarily involve exception handling.

H.1. 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 PDP-11's and then ported to VAXes and IBM and Honeywell mainframe computers. These various 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.

H.2. IEEE 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. High performance conforming processors such as the MC68881 compute the common elementary transcendental functions in single instructions that look, from a programmer's viewpoint, just like the instructions for +, -, * and /.

H.3. SVID Future Directions

The current SVID 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 would be infinity on IEEE systems. `HUGE_VAL` would, for instance, be returned as the result of floating-point overflows. In this respect, Sun's implementation has already arrived at SVID's future direction, since the constant `HUGE` in `/usr/include/math.h` is defined to be a constant that compiles into IEEE infinity.

H.4. Sun Implementation

In Release 3.2, the following C language `libc` and `libm` functions provide operand or result checking corresponding to SVID, when called from C programs compiled with `-fsoft`, or compiled with `-fswitch` and run without floating-point hardware:

- `exp`
- `log` and `log10`
- `pow`
- `sqrt`
- `hypot`
- `cabs`
- `sinh` and `cosh`
- `sin`, `cos` and `tan`
- `asin`, `acos`, `atan` and `atan2`

When exceptional conditions are detected, the SVID function `matherr()` is invoked. The default `matherr()` in `libm` returns a default result value and, for `EDOM` errors, prints an error message prior to continuing. Because SVID encompasses machines without infinities or NaNs, the default results specified are finite values and therefore sometimes misleading. Users may provide their own `matherr()` function to obtain alternative processing.

For efficiency, programs compiled with inline hardware floating-point or with `-fswitch` and run with floating-point hardware do not do the extra checking

required to set EDOM or ERANGE or call matherr(). Usually NaN is returned for the function value in situations where EDOM might otherwise be set, and Infinity is returned for the function value where ERANGE might be set to indicate overflow. Where ERANGE might be set to indicate underflow, these functions return subnormal numbers or zero.

Note that if inline expansion files are used to expand libm functions, the SVID exception handling may be bypassed, even for -fsoft.

SIGNAL Notes

The SVID defines SIGFPE as "floating-point exception". On Suns, SIGFPE is also generated by MC68010 and MC68020 integer division by zero, CHK and TRAPV instructions, and by FPA-related non-numerical exceptions.

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 Sun algorithms used with -fsoft, -f68881, and -ffpa do not suffer such abrupt declines, and so do not signal PLOSS or TLOSS, nor do the Bessel functions which call the trigonometric functions.

Instead, Sun's sin, cos, and tan treat the essential singularity at infinity like other essential singularities by returning a NaN and setting EDOM for infinite arguments. The Bessel functions of the first kind, j0, j1, and jn, return zero for infinite arguments, while the Bessel functions of the second kind, y0, y1, and yn, return zero for positive infinite arguments, and return NaN and set EDOM for negative arguments.

Likewise SVID specifies that fmod(x,y) should be zero if x/y would overflow, but Sun's fmod, derived from the IEEE remainder function, does not compute x/y explicitly and hence always delivers a correctly rounded result.

Notes

Notes

Notes