Oracle® Solaris Studio 12.4: 数値計算ガイド

Part No: E57329
2015 年 1 月

ORACLE

目次

このドキュメントの使用法 11

1 概要 13
1.1 浮動小数点環境 13

2 IEEE 演算 15
2.1 IEEE 演算モデル 15
2.1.1 IEEE 演算について 15
2.2 IEEE 形式 17
2.2.1 格納形式 17
2.2.2 単精度形式 17
2.2.3 倍精度形式 20
2.2.4 4 倍精度形式 22
2.2.5 拡張倍精度形式 (x86) 25
2.2.6 10 進数表現の範囲と精度 28
2.2.7 Oracle Solaris 環境での基数変換 31
2.3 アンダーフロー 32
2.3.1 アンダーフローしきい値 33
2.3.2 IEEE 演算でのアンダーフローの処理方法 34
2.3.3 段階的アンダーフローを使用する理由 35
2.3.4 段階的アンダーフローの誤差の属性 35
2.3.5 段階的アンダーフローと突発的アンダーフローを比較した 2 つの例 38
2.3.6 アンダーフローは問題か 39
2.4 IEEE 標準 754-2008 40

3 数学ライブラリ 43
3.1 Oracle Solaris の数学ライブラリ 43
3.1.1 標準数学ライブラリ 43
3.1.2 ベクトル数学ライブラリ 45
3.2 Oracle Solaris Studio の数学ライブラリ 46

3.2.1 Oracle 数学ライブラリ 47
3.2.2 最適化されたライブラリ 48
3.3 単精度、倍精度、および拡張/4 倍精度 49
3.4 IEEE サポート関数 50
3.4.1 ieee_functions(3m) および ieee_sun(3m) 50
3.4.2 ieee_values(3m) 51
3.4.3 ieee_flags(3m) 54
3.4.4 ieee_retrospective(3m) 56
3.4.5 nonstandard_arithmetic(3m) 58
3.5 C99 浮動小数点環境関数 58
3.5.1 例外フラグ関数 59
3.5.2 丸めの制御 60
3.5.3 環境関数 60
3.6 libm および libsunmath の実装機能 62
3.6.1 アルゴリズムについて 62
3.6.2 三角関数の引数還元 62
3.6.3 データ変換ルーチン 63
3.6.4 乱数の機能 64

4 例外と例外処理 67
4.1 例外処理の目的 67
4.2 例外とは 68
4.2.1 表 4-1 の注 70
4.3 例外の検出 71
4.3.1 ieee_flags(3m) 72
4.3.2 C99 例外フラグ関数 74
4.4 例外の特定 75
4.4.1 デバッガを使用して例外を特定する 75
4.4.2 シグナルハンドラを使用して例外を特定する 82
4.4.3 libm の例外処理拡張機能を使用して例外を特定する 87
4.5 例外の処理 93
4.5.1 IEEE トラップされたアンダーフローおよびオーバーフローの置換 95

5 コンパイラコードの生成 107
5.1 サポートされているオペレーティングシステム、ハードウェア、およびメモリーモデル 107
5.2 コード生成オプション 108
5.3 デフォルトのアドレスモデルとコード生成 109
5.4 コンパイルオプション 110

5.5 再現可能な結果 111
5.5.1 超越関数 112
5.5.2 連想演算 112
5.5.3 不定の評価 113
5.5.4 移植できない型 113
5.5.5 高い暗黙的精度 113
5.6 独立した確認 114

A 例 115
A.1 IEEE 演算 115
A.2 数学ライブラリ 117
A.2.1 乱数ジェネレータ 117
A.2.2 IEEE が推奨する関数 119
A.2.3 IEEE の特殊な値 122
A.2.4 `ieee_flags` — 丸め方向 124
A.2.5 C99 浮動小数点環境関数 125
A.3 例外と例外処理 129
A.3.1 `ieee_flags` — 累積例外 129
A.3.2 `ieee_handler`: 例外のトラップ 132
A.3.3 `ieee_handler`: 例外での中止 138
A.3.4 `libm` の例外処理機能 138
A.3.5 Fortran プログラムでの `libm` 例外処理の使用 143
A.4 その他 146
A.4.1 `sigfpe`: 整数例外のトラップ 146
A.4.2 C からの Fortran の呼び出し 147
A.4.3 役に立つデバッグコマンド 150

B SPARC の動作と実装 155
B.1 浮動小数点ハードウェア 155
B.1.1 浮動小数点ステータスレジスタおよびキュー 157
B.1.2 ソフトウェアサポートを必要とする特殊な場合 159
B.2 *fpversion*(1) 関数: FPU に関する情報の検索 164

C x86 の動作と実装 167
C.1 サポートされているシステムのコード生成 167
C.2 SPARC との差異 168

D 『*浮動小数点演算について計算機科学者は何を知っておくべきか*』の付録 171

D.1 IEEE 754 実装間の相違 172
D.1.1 現在の IEEE 754 の実装 173
D.1.2 拡張ベースシステムでの計算の落とし穴 175
D.1.3 拡張精度におけるプログラミング言語のサポート 180
D.1.4 結論 185

E 標準規格への準拠 187
E.1 `libm` の特殊なケース 187
E.1.1 標準規格への準拠に影響を及ぼすその他のコンパイラフラグ 191
E.1.2 C99 への準拠に関するその他の注意事項 192
E.2 LIA-1 への準拠 193
E.2.1 a.データ型 (LIA 5.1): 193
E.2.2 b.パラメータ (LIA 5.1): 193
E.2.3 d.DIV/REM/MOD (LIA 5.1.3): 194
E.2.4 i.表記法 (LIA 5.1.3): 194
E.2.5 j.式評価: 195
E.2.6 k.パラメータの取得方法: 195
E.2.7 n.通知: 196
E.2.8 o.選択メカニズム: 196

F 参考資料 197
F.1 第 2 章:「IEEE 演算」..... 197
F.2 第 3 章:「数学ライブラリ」..... 198
F.3 第 4 章:「例外と例外処理」..... 199
F.4 標準規格 200
F.5 テストプログラム 200

用語集 201

索引 205

図目次

図 2-1 単精度格納形式 18
図 2-2 倍精度格納形式 20
図 2-3 4 倍精度形式 23
図 2-4 拡張倍精度形式 (x86) 26
図 2-5 デジタル表現と 2 進数表現で定義される数値のセットの比較 29
図 2-6 数直線 36
図 B-1 SPARC 浮動小数点ステータスレジスタ 158

表目次

表 2-1 IEEE 形式と言語の型 17
表 2-2 IEEE 単精度形式のビットパターンによって表される値 18
表 2-3 単精度格納形式のビットパターンとその IEEE 値 19
表 2-4 IEEE 倍精度形式のビットパターンによって表される値 21
表 2-5 倍精度格納形式のビットパターンとその IEEE 値 22
表 2-6 各ビットパターンによって表される値 23
表 2-7 4 倍精度形式のビットパターン 24
表 2-8 各ビットパターンによって表される値 (x86) 26
表 2-9 拡張倍精度形式のビットパターンとその値 (x86) 27
表 2-10 格納形式の範囲と精度 31
表 2-11 アンダーフローしきい値 33
表 2-12 4 つの異なる精度での ulp(1) 36
表 2-13 表現可能な単精度形式の浮動小数点の間隔 37
表 3-1 libm の内容 44
表 3-2 libmvec の内容 45
表 3-3 libsunmath の内容 47
表 3-4 単精度、倍精度、および拡張/4 倍精度関数の呼び出し 49
表 3-5 ieee_functions(3m) 50
表 3-6 ieee_sun(3m) 50
表 3-7 Fortran からの ieee_functions の呼び出し 51
表 3-8 Fortran からの ieee_sun の呼び出し 51
表 3-9 IEEE 値: 単精度 52
表 3-10 IEEE 値: 倍精度 52
表 3-11 IEEE 値: 4 倍精度 53
表 3-12 IEEE 値: 拡張倍精度 (x86) 53
表 3-13 ieee_flags のパラメータ値 54
表 3-14 丸め方向に関する ieee_flags の入力値 55
表 3-15 C99 規格の例外フラグ関数 59
表 3-16 libm 浮動小数点環境関数 60
表 3-17 1 回につき 1 つ値の乱数ジェネレータの間隔 64

表 4-1 IEEE 浮動小数点例外 69
表 4-2 例外ビット 73
表 4-3 算術例外のタイプ 85
表 4-4 fex_set_handling の例外コード 88
表 A-1 いくつかのデバッグコマンド (SPARC) 150
表 A-2 いくつかのデバッグコマンド (x86) 151
表 B-1 Oracle Solaris 11 以降でサポートされている SPARC システム 155
表 B-2 Oracle Solaris 10 Update 10 ではサポートされているが Oracle Solaris 11 ではサポートされていない UltraSPARC システム 156
表 B-3 浮動小数点ステータスレジスタフィールド 158
表 B-4 例外処理フィールド 159
表 E-1 特殊なケースと libm 関数 188
表 E-2 Solaris と C99/SUSv3 の相違点 193
表 E-3 LIA-1 への準拠 - 表記法 194

このドキュメントの使用法

- **概要** – Oracle Solaris オペレーティングシステムが稼働している SPARC ベースシステムおよび x86 ベースシステムにおいて、ソフトウェアおよびハードウェアによってサポートされる浮動小数点環境について説明します。
- **対象読者** – アプリケーション開発者、システム開発者、設計者、サポートエンジニア。
- **必要な知識** – プログラミングの実務、ソフトウェア開発のテスト、ソフトウェア製品のビルドおよびコンパイルの適性

製品ドキュメントライブラリ

この製品の最新情報や既知の問題は、ドキュメントライブラリ (http://docs.oracle.com/cd/E37069_01) に記載されています。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

♦ ♦ ♦ 第 1 章

1 概要

SPARC システムおよび Intel x86 システムで Oracle の浮動小数点環境を利用すると、高性能で堅牢な移植性のある数値計算アプリケーションを開発できます。浮動小数点環境は、ほかのユーザーが記述した数値プログラムの異常な動作を調査する場合にも役に立ちます。これらのシステムは、IEEE 標準 754 (2 進浮動小数点演算) で規定されている演算モデルを実装しています。このマニュアルでは、これらのシステム上で IEEE 標準によって提供されるオプションおよび柔軟性を利用する方法を説明しています。

1.1 浮動小数点環境

*浮動小数点環境*は、IEEE 標準 754 を実装したハードウェア、システムソフトウェア、およびソフトウェアライブラリによってアプリケーションプログラマが利用できるようになったデータ構造および演算で構成されています。IEEE 標準 754 を使用すると、数値計算アプリケーションの記述が簡単になります。これは綿密に考案されたコンピュータ演算の基礎であり、数値計算プログラミングの技術を進化させるものです。

たとえば、ハードウェアは IEEE データ形式に対応するストレージの形式、そのような形式のデータに対する演算、それらの演算が生成した結果の丸めの制御、IEEE 数値例外の発生を示すステータスフラグ、およびユーザー定義のハンドラがない場合にそのような例外が発生したときの IEEE の規定の結果を提供します。システムソフトウェアは IEEE 例外処理をサポートします。数学ライブラリ `libm` および `libsunmath` を含むソフトウェアライブラリは、例外の発生に関して IEEE 標準 754 に準拠した方法で、`exp(x)` や `sin(x)` などの関数を実装しています。浮動小数点の算術演算で十分に定義された結果を得られない場合、システムは*例外を発生させる*ことによって、その事実をユーザーに通知します。また、数学ライブラリは `Inf` (無限大) または `NaN` (非数) のような特殊な IEEE の値を処理する関数呼び出しも提供しています。

浮動小数点環境のこの 3 つの構成要素の相互作用はわずかなため、通常はアプリケーションプログラマがこの相互作用を意識することはありません。プログラマは、IEEE 標準によって規定または推奨されている計算メカニズムのみを意識します。全般的に、このマニュアルではプ

ログラマが IEEE メカニズムを完全および有効に活用し、アプリケーションソフトウェアを効果的に記述できるようになることを目的としています。

浮動小数点演算に関する質問の多くは、数値の基本的な演算に関連しています。次の質問について考えてみます。

- コンピュータシステムで無限に正確な結果を表現できない場合、演算の結果はどうなるか。
- 乗算や加算のような基本的な演算は交換可能か。

その他の種類の質問は、例外や例外処理に関連しています。たとえば、次のような操作を行なった場合の結果です。

- 非常に大きい 2 つの数値の乗算
- ゼロによる除算
- 負の数の正方根計算の試行

一部の演算では、最初の種類の質問が予期した結果にならなかったり、2 番目の種類でも例外のケースで同様のことが発生する場合があります。その場合、プログラムがその時点で停止します。一部の非常に古いマシンでは、計算は続行されますが、結果が意味のないものになります。

IEEE 標準 754 では、演算によって、予期した特性を備えた、数学的に期待される結果が生成されることを保証しています。また、ユーザーが特にほかに選択しなければ、例外のケースでも指定した結果が得られます。

このマニュアルでは、`NaN`または*非正規数*などの用語に触れています。浮動小数点演算に関する用語については、201 ページの用語集で定義しています。

♦ ♦ ♦ 第 2 章

2 IEEE 演算

この章では、2 進浮動小数点演算のための ANSI/IEEE 規格 754-1985 (「IEEE 規格」または略して「IEEE 754」) で指定されている演算モデルについて説明します。SPARC® および x86 プロセッサのすべてが IEEE 演算を使用しています。Oracle Solaris Studio コンパイラは、IEEE 演算の機能をサポートしています。この章では、次の内容について説明します。

- 15 ページの「IEEE 演算モデル」
- 17 ページの「IEEE 形式」
- 32 ページの「アンダーフロー」
- 40 ページの「IEEE 標準 754-2008」

2.1 IEEE 演算モデル

このセクションでは、IEEE 754-1985 の仕様について説明します。IEEE 規格は、2008 年に大幅に改訂されました。

2.1.1 IEEE 演算について

IEEE 754 は、次のものを指定しています。

- *単精度*と*倍精度*という 2 つの基本的な浮動小数点形式。

  IEEE 単精度形式は仮数の精度が 24 ビットであり、全体で 32 ビットを占有します。IEEE 倍精度形式は仮数の精度が 53 ビットであり、全体で 64 ビットを占有します。
- *拡張単精度*と*拡張倍精度*という拡張浮動小数点形式の 2 つのクラス。

  この規格では、これらの形式の正確な精度やサイズは規定されておらず、最小の精度とサイズが指定されています。たとえば、IEEE 拡張倍精度形式は仮数の精度が少なくとも 64 ビットであり、全体で少なくとも 79 ビットを占有する必要があります。

- 浮動小数点演算に関する正確性の要件: *加算*,*減算*,*乗算*,*除算*,*平方根*,*剰余*,*浮動小数点形式の数値の整数値への丸め*,*異なる浮動小数点形式間の変換*,*浮動小数点と整数形式の間の変換*,*および比較*。

  剰余演算と比較演算は正確である必要があります。その他の各演算は、その宛先に正確な結果を提供する必要があります。ただし、このような結果が存在しない場合や、その結果が宛先の形式に収まらない場合を除きます。後者の場合、その演算は、規定された丸めモード (下で説明します) の規則に従って正確な結果を最小限に変更し、そのように変更された結果を演算の宛先に提供する必要があります。

- 10 進数文字列と、両方の基本的な浮動小数点形式の 2 進浮動小数点数の間の変換に関する正確性、単調性、および同一性の要件。

  指定された範囲内に収まるオペランドの場合、これらの変換は正確な結果を生成するか (可能な場合)、または規定された丸めモードの規則に従ってこのような正確な結果を最小限に変更する必要があります。指定された範囲内に収まらないオペランドの場合、これらの変換は、丸めモードによって異なる指定された許容範囲を超えない程度に正確な結果と異なる結果を生成する必要があります。

- 5 つの種類の IEEE 浮動小数点例外と、これらの種類の例外の発生をユーザーに示すための条件。

  5 つの種類の浮動小数点例外は、*無効な演算*,*0 による除算*,*オーバーフロー*,*アンダーフロー*,および*不正確です*。

- 4 つの丸め方向: *もっとも近い表現可能な値に向けて* (もっとも近い表現可能な値が 2 つ存在する場合は常に「偶数」の値が優先されます)、*負の無限大に向けて* (下へ)、*正の無限大に向けて* (上へ)、および *0 に向けて* (切り捨て)。

- 丸め精度。たとえば、システムが結果を拡張倍精度形式で提供する場合は、ユーザーがこのような結果を単精度または倍精度形式のどちらかの精度に丸めるよう指定できるようにするべきです。

IEEE 規格ではまた、例外のユーザー処理のサポートも推奨されています。

IEEE 規格に必要な機能によって、区間演算、異常の遡及診断、`exp` や `cos` などの標準的な基本演算の効率的な実装、多倍精度演算、数値計算に役立つその他の多くのツールをサポートすることが可能になります。

IEEE 754 浮動小数点演算を使用すると、ユーザーは計算を、その他のどの種類の浮動小数点演算よりも詳細に制御できます。IEEE 規格は、実装の準拠に厳しい要件を課すことによってだけでなく、このような実装によるその規格自体の改良や拡張も可能にすることによって、数値的に複雑な、移植性のあるプログラムを記述するタスクを簡素化します。

2.2 IEEE 形式

このセクションでは、浮動小数点データがメモリー内にどのように格納されるかについて説明します。ここでは、IEEE のさまざまな格納形式の精度と範囲の要約を示します。

2.2.1 格納形式

浮動小数点形式は、浮動小数点数値を構成する各フィールド、これらのフィールドのレイアウト、およびその演算の解釈を指定するデータ構造です。浮動小数点の*格納*形式は、浮動小数点形式がメモリー内にどのように格納されるかを指定します。IEEE 規格は形式を定義しますが、それらの格納形式の選択は実装者に任されています。

アセンブリ言語のソフトウェアは、これらの格納形式の使用に依存している場合もありますが、より高水準な言語は通常、浮動小数点データ型の言語上の概念のみを扱います。これらの型は高水準言語ごとに異なる名前を持ち、表2-1「IEEE 形式と言語の型」に示すように IEEE 形式に対応しています。

表 2-1 IEEE 形式と言語の型

IEEE の精度	C、C++	Fortran
単精度	float	`REAL` または `REAL*4`
倍精度	double	`DOUBLE PRECISION` または `REAL*8`
拡張倍精度	long double (x86)	—
4 倍精度	long double (SPARC)	`REAL*16`

IEEE 754 は、単精度と倍精度の浮動小数点形式を正確に指定し、これらの 2 つの各基本形式に対して拡張形式のクラスを定義しています。表2-1「IEEE 形式と言語の型」に示されている `long double` および `REAL*16` 型は、IEEE 規格で定義されている拡張倍精度形式のクラスの 1 つを示しています。

以降のセクションでは、SPARC および x86 プラットフォーム上の IEEE 浮動小数点形式に使用される各格納形式について詳細に説明します。

2.2.2 単精度形式

IEEE 単精度形式は、23 ビットの小数部 `f`、8 ビットのバイアス付き指数 `e`、および 1 ビットの符号 `s` の 3 つのフィールドで構成されています。これらのフィールドは、次の図に示すように、1

つの 32 ビットワードに連続して格納されます。ビット 0:22 には 23 ビットの小数部 f が含まれ (ビット 0 が小数部の最下位ビット、ビット 22 が最上位ビット)、ビット 23:30 には 8 ビットのバイアス付き指数 e が含まれ (ビット 23 がバイアス付き指数の最下位ビット、ビット 30 が最上位ビット)、最上位ビット 31 には符号ビット s が含まれています。

図 2-1 単精度格納形式

表2-2「IEEE 単精度形式のビットパターンによって表される値」は、一方にある 3 つの構成フィールド s、e、および f の値と、もう一方にある単精度形式のビットパターンによって表される値の間の対応関係を示しています。*u* は、示されているフィールドの値が、単精度形式の特定のビットパターンの値の決定には関係しないことを示しています。

表 2-2 IEEE 単精度形式のビットパターンによって表される値

単精度形式のビットパターン	値
0 < e < 255	$(-1)^s \times 2^{e-127} \times 1.f$ (正規数)
e = 0; f ≠ 0	$(-1)^s \times 2^{-126} \times 0.f$ (非正規数)
(f 内の少なくとも 1 ビットは 0 以外)	
e = 0; f = 0	$(-1)^s \times 0.0$ (符号付き 0)
(f 内のすべてのビットが 0)	
s = 0; e = 255; f = 0 (f 内のすべてのビットが 0)	+INF (正の無限大)
s = 1; e = 255; f = 0 (f 内のすべてのビットが 0)	–INF (負の無限大)
s = u; e = 255; f ≠ 0	NaN (非数)
(f 内の少なくとも 1 ビットは 0 以外)	

e < 255 の場合、単精度形式のビットパターンに割り当てられる値は、小数部の最上位ビットのすぐ左側に 2 進基数点を挿入し、2 進小数点のすぐ左側に暗黙ビットを挿入することによって形成されます。それにより、混在した数値 (整数と小数部。ここで、0 ≤ 小数部 < 1) が 2 進定位置表記で表されます。

このように形成された混在した数値は、*単精度形式の仮数*と呼ばれます。暗黙ビットにこの名前が付けられているのは、その値が単精度形式のビットパターンでは明示的に指定されておらず、バイアス付き指数フィールドの値によって暗に示されているためです。

単精度形式の場合、正規数と非正規数の違いは、正規数の仮数の先行ビット (2 進小数点の左側のビット) が 1 であるのに対して、非正規数の仮数の先行ビットは 0 である点です。単精度形式の非正規数は、IEEE 規格 754 では単精度形式の非正規化数という名前になりました。

23 ビットの小数部が暗黙的先行仮数ビットと組み合わされて、単精度形式の正規数の 24 ビットの精度が実現されます。

単精度格納形式での重要なビットパターンの例を表2-3「単精度格納形式のビットパターンとその IEEE 値」に示します。最大の正の正規数は、IEEE 単精度形式で表すことができる最大の有限数です。最小の正の非正規数は、IEEE 単精度形式で表すことができる最小の正の数値です。最小の正の正規数は多くの場合、アンダーフローしきい値と呼ばれます。(最大と最小の正規数および非正規数に対する 10 進数値は近似値であり、示されている桁数に関して正確です。)

表 2-3 単精度格納形式のビットパターンとその IEEE 値

共通名	ビットパターン (16 進数)	10 進数値
+0	`00000000`	0.0
–0	`80000000`	–0.0
1	`3f800000`	1.0
2	`40000000`	2.0
最大の正規数	`7f7fffff`	3.40282347e+38
最小の正の正規数	`00800000`	1.17549435e–38
最大の非正規数	`007fffff`	1.17549421e–38
最小の正の非正規数	`00000001`	1.40129846e–45
+∞	`7f800000`	無限大
–∞	`ff800000`	負の無限大
非数	`7fc00000`	NaN

NaN (非数) は、NaN の定義を満足させる多くのビットパターンのいずれかで表すことができます。表2-3「単精度格納形式のビットパターンとその IEEE 値」に示されている NaN の 16 進数値は、NaN を表すために使用できる多くのビットパターンの 1 つにすぎません。

2.2.3 倍精度形式

IEEE 倍精度形式は、52 ビットの小数部 f、11 ビットのバイアス付き指数 e、および 1 ビットの符号 s の 3 つのフィールドで構成されています。これらのフィールドは、次の図に示すように、アドレスが連続した 2 つの 32 ビットワードに連続して格納されます。

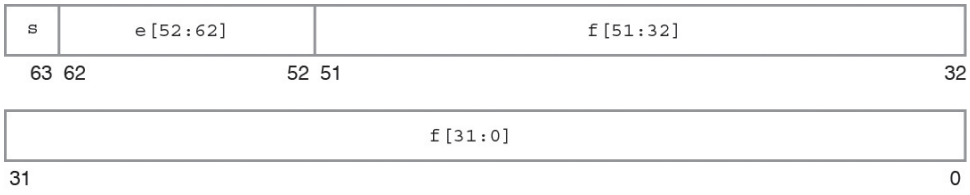
SPARC アーキテクチャーでは、上位アドレスの 32 ビットワードに小数部の最下位 32 ビットが含まれるのに対して、x86 アーキテクチャーでは、下位アドレスの 32 ビットワードに小数部の最下位 32 ビットが含まれます。

`f[31:0]` が小数部の最下位 32 ビットを示している場合は、ビット 0 が小数部全体の最下位ビットであり、ビット 31 は小数部の最下位 32 ビットの最上位ビットです。

もう一方の 32 ビットワードでは、ビット 0:19 には小数部の最上位 20 ビット `f[51:32]` が含まれ (ビット 0 がこれらの小数部の最上位 20 ビットの最下位ビット、ビット 19 が小数部全体の最上位ビット)、ビット 20:30 には 11 ビットのバイアス付き指数 e が含まれ (ビット 20 がバイアス付き指数の最下位ビット、ビット 30 が最上位ビット)、最上位ビット 31 には符号ビット s が含まれています。

次の図では、2 つの連続した 32 ビットワードが 1 つの 64 ビットワードであるかのように各ビットに番号を付けています。そこでは、ビット 0:51 に 52 ビットの小数部 f が格納され、ビット 52:62 に 11 ビットのバイアス付き指数 e が格納され、ビット 63 に符号ビット s が格納されています。

図 2-2 倍精度格納形式



これらの 3 つのフィールド内のビットパターンの値によって、ビットパターン全体で表される値が決定されます。

表2-4「IEEE 倍精度形式のビットパターンによって表される値」は、一方にある 3 つの構成フィールド内のビットの値と、もう一方にある倍精度形式のビットパターンによって表される値の

間の対応関係を示しています。*u* は、示されているフィールドの値が、倍精度形式の特定のビットパターンの値の決定には関係しないことを示しています。

表 2-4 IEEE 倍精度形式のビットパターンによって表される値

倍精度形式のビットパターン	値
0 < e < 2047	$(-1)^s \times 2^{e-1023} \times 1.f$ (正規数)
e = 0; f ≠ 0 (f 内の少なくとも 1 ビットは 0 以外)	$(-1)^s \times 2^{-1022} \times 0.f$ (非正規数)
e = 0; f = 0 (f 内のすべてのビットが 0)	$(-1)^s \times 0.0$ (符号付き 0)
s = 0; e = 2047; f = 0 (f 内のすべてのビットが 0)	+INF (正の無限大)
s = 1; e = 2047; f = 0 (f 内のすべてのビットが 0)	–INF (負の無限大)
s = u; e = 2047; f ≠ 0 (f 内の少なくとも 1 ビットは 0 以外)	NaN (非数)

e < 2047 の場合、倍精度形式のビットパターンに割り当てられる値は、小数部の最上位ビットのすぐ左側に 2 進基数点を挿入し、2 進小数点のすぐ左側に暗黙ビットを挿入することによって形成されます。このように形成された数値は、*仮数*と呼ばれます。暗黙ビットにこの名前が付けられているのは、その値が倍精度形式のビットパターンでは明示的に指定されておらず、バイアス付き指数フィールドの値によって暗に示されているためです。

倍精度形式の場合、正規数と非正規数の違いは、正規数の仮数の先行ビット (2 進小数点の左側のビット) が 1 であるのに対して、非正規数の仮数の先行ビットは 0 である点です。倍精度形式の非正規数は、IEEE 規格 754 では倍精度形式の非正規化数という名前になりました。

52 ビットの小数部が暗黙的先行仮数ビットと組み合わされて、倍精度形式の正規数の 53 ビットの精度が実現されます。

倍精度格納形式での重要なビットパターンの例を表2-5「倍精度格納形式のビットパターンとその IEEE 値」に示します。2 番目の列のビットパターンは、2 つの 8 桁 16 進数として示されています。SPARC アーキテクチャーでは、左側が下位アドレスの 32 ビットワードの値であり、右側が上位アドレスの 32 ビットワードの値であるのに対して、x86 アーキテクチャーでは、左側は上位アドレスのワードであり、右側は下位アドレスのワードです。最大の正の正規数は、IEEE

倍精度形式で表すことができる最大の有限数です。最小の正の非正規数は、IEEE 倍精度形式で表すことができる最小の正の数値です。最小の正の正規数は多くの場合、アンダーフローしきい値と呼ばれます。(最大と最小の正規数および非正規数に対する 10 進数値は近似値であり、示されている桁数に関して正確です。)

表 2-5 倍精度格納形式のビットパターンとその IEEE 値

共通名	ビットパターン (16 進数)	10 進数値
+ 0	`00000000 00000000`	0.0
– 0	`80000000 00000000`	–0.0
1	`3ff00000 00000000`	1.0
2	`40000000 00000000`	2.0
最大の正規数	`7fefffff ffffffff`	1.7976931348623157e+308
最小の正の正規数	`00100000 00000000`	2.2250738585072014e–308
最大の非正規数	`000fffff ffffffff`	2.2250738585072009e–308
最小の正の非正規数	`00000000 00000001`	4.9406564584124654e–324
+∞	`7ff00000 00000000`	無限大
–∞	`fff00000 00000000`	負の無限大
非数	`7ff80000 00000000`	NaN

NaN (非数) は、NaN の定義を満足させる多くのビットパターンのいずれかで表すことができます。表2-5「倍精度格納形式のビットパターンとその IEEE 値」に示されている NaN の 16 進数値は、NaN を表すために使用できる多くのビットパターンの 1 つにすぎません。

2.2.4 4 倍精度形式

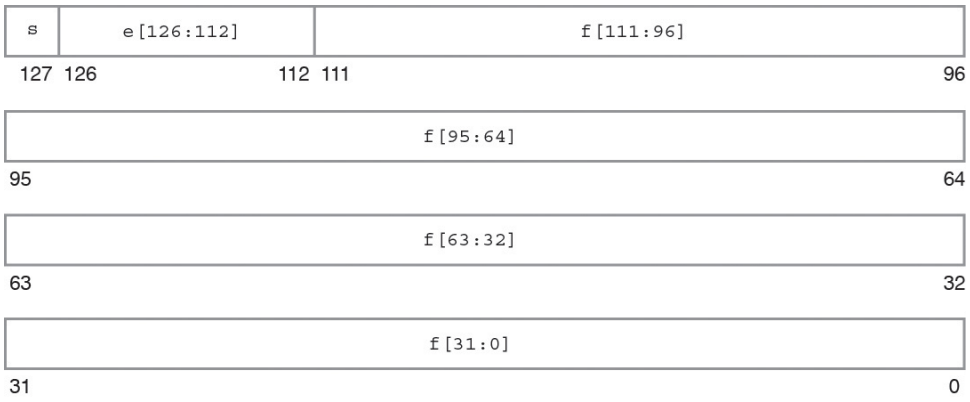
浮動小数点環境の 4 倍精度形式もまた、拡張倍精度形式の IEEE 定義に準拠しています。この形式は、x86 用の Oracle Solaris Studio C/C++ コンパイラには存在しません。4 倍精度形式は 4 つの 32 ビットワードを占有し、112 ビットの小数部 `f`、15 ビットのバイアス付き指数 `e`、および 1 ビットの符号 `s` の 3 つのフィールドで構成されています。これらは、次の図に示すように連続して格納されます。

最上位アドレスの 32 ビットワードには、f[31:0] で示される小数部の最下位 32 ビットが含まれています。次の 2 つの 32 ビットワードには、それぞれ `f[63:32]` と `f[95:64]` が含まれています。次のワードのビット 0:15 には、小数部の最上位 16 ビット `f[111:96]` が含まれています (ビット 0 がこれらの 16 ビットの最下位ビット、ビット 15 が小数部全体の最上位ビット)。ビッ

ト 16:30 には 15 ビットのバイアス付き指数 e が含まれ (ビット 16 がバイアス付き指数の最下位ビット、ビット 30 が最上位ビット)、ビット 31 には符号ビット s が含まれています。

次の図では、4 つの連続した 32 ビットワードが 1 つの 128 ビットワードであるかのように各ビットに番号を付けています。そこでは、ビット 0:111 に小数部 f が格納され、ビット 112:126 に 15 ビットのバイアス付き指数 e が格納され、ビット 127 に符号ビット s が格納されています。

図 2-3 4 倍精度形式



3 つのフィールド f、e、および s 内のビットパターンの値によって、ビットパターン全体で表される値が決定されます。

表2-6「各ビットパターンによって表される値」は、3 つの構成フィールドの値と、4 倍精度形式のビットパターンによって表される値の間の対応関係を示しています。u は、示されているフィールドの値が特定のビットパターンの値の決定には関係しないため、考慮が必要ないものです。

表 2-6 各ビットパターンによって表される値

4 倍精度のビットパターン	値
0 < e < 32767	$(-1)^s \times 2^{e-16383} \times 1.f$ (正規数)
e = 0, f ≠ 0 (f 内の少なくとも 1 ビットは 0 以外)	$(-1)^s \times 2^{-16382} \times 0.f$ (非正規数)
e = 0, f = 0 (f 内のすべてのビットが 0)	$(-1)^s \times 0.0$ (符号付き 0)

4 倍精度のビットパターン	値
s = 0, e = 32767, f = 0 (f 内のすべてのビットが 0)	+INF (正の無限大)
s = 1, e = 32767; f = 0 (f 内のすべてのビットが 0)	-INF (負の無限大)
s = u, e = 32767, f ≠ 0 (f 内の少なくとも 1 ビットは 0 以外)	NaN (非数)

4 倍精度の拡張倍精度格納形式での重要なビットパターンの例を表2-7「4 倍精度形式のビットパターン」に示します。2 番目の列のビットパターンは、4 つの 8 桁 16 進数として示されています。いちばん左の数値は最下位アドレスの 32 ビットワードの値であり、いちばん右の数値は最上位アドレスの 32 ビットワードの値です。最大の正の正規数は、4 倍精度形式で表すことができる最大の有限数です。最小の正の非正規数は、4 倍精度形式で表すことができる最小の正の数値です。最小の正の正規数は多くの場合、アンダーフローしきい値と呼ばれます。(最大と最小の正規数および非正規数に対する 10 進数値は近似値であり、示されている桁数に関して正確です。)

表 2-7 4 倍精度形式のビットパターン

共通名	ビットパターン (SPARC)	10 進数値
+0	`00000000 00000000 00000000 00000000`	0.0
−0	`80000000 00000000 00000000 00000000`	−0.0
1	`3fff0000 00000000 00000000 00000000`	1.0
2	`40000000 00000000 00000000 00000000`	2.0
最大の正規数	`7ffeffff ffffffff ffffffff ffffffff`	1.189731495357231765085759326628007 0e+4932
最小の正規数	`00010000 00000000 00000000 00000000`	3.362103143112093506262677817321752 6e−4932
最大の非正規数	`0000ffff ffffffff far-off ffffffff`	3.362103143112093506262677817321752 0e−4932
最小の正の非正規数	`00000000 00000000 00000000 00000001`	6.475175119438025110924438958227646 6e−4966
+∞	`7fff0000 00000000 00000000 00000000`	+∞
−∞	`ffff0000 00000000 00000000 00000000`	−∞
非数	`7fff8000 00000000 00000000 00000000`	NaN

表2-7「4 倍精度形式のビットパターン」に示されている NaN の 16 進数値は、NaN を表すために使用できる多くのビットパターンの 1 つにすぎません。

2.2.5 拡張倍精度形式 (x86)

この浮動小数点環境の拡張倍精度形式は、拡張倍精度形式の IEEE 定義に準拠しています。これは、63 ビットの小数部 `f`、1 ビットの明示的先行仮数ビット `j`、15 ビットのバイアス付き指数 `e`、および 1 ビットの符号 `s` の 4 つのフィールドで構成されています。この形式は、SPARC 用の Oracle Solaris Studio Fortran または C/C++ の言語の型としては使用できません。

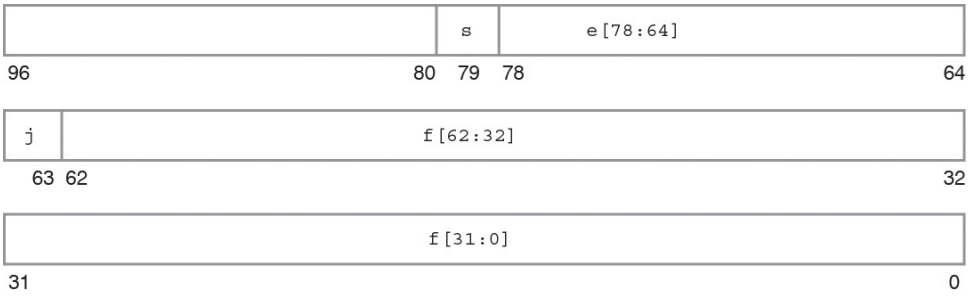
x86 アーキテクチャーファミリでは、これらのフィールドは、アドレスが連続した 10 個の 8 ビットバイトに連続して格納されます。ただし、UNIX System V Application Binary Interface Intel 386 Processor Supplement (Intel ABI) では、次の図に示すように、拡張倍精度のパラメータや結果がスタック内でアドレスが連続した 3 つの 32 ビットワード (最上位アドレスのワードの最上位 16 ビットは未使用) を占有する必要があります。

最下位アドレスの 32 ビットワードには、小数部 `f[31:0]` の最下位 32 ビットが含まれています (ビット 0 が小数部全体の最下位ビット、ビット 31 が小数部の最下位 32 ビットの最上位ビット)。中央のアドレスの 32 ビットワードでは、ビット 0:30 には小数部の最上位 31 ビット `f[62:32]` が含まれ (ビット 0 がこれらの小数部の最上位 31 ビットの最下位ビット、ビット 30 が小数部全体の最上位ビット)、この中央のアドレスの 32 ビットワードのビット 31 には明示的先行仮数ビット `j` が含まれています。

最上位アドレスの 32 ビットワードでは、ビット 0:14 には 15 ビットのバイアス付き指数 `e` が含まれ (ビット 0 がバイアス付き指数の最下位ビット、ビット 14 が最上位ビット)、ビット 15 には符号ビット `s` が含まれています。この最上位アドレスの 32 ビットワードの最上位 16 ビットは x86 アーキテクチャーファミリでは未使用ですが、上で示したように、それらの存在は Intel ABI への準拠のために不可欠です。

次の図では、3 つの連続した 32 ビットワードが 1 つの 96 ビットワードであるかのように各ビットに番号を付けています。そこでは、ビット 0:62 に 63 ビットの小数部 `f` が格納され、ビット 63 に明示的先行仮数ビット `j` が格納され、ビット 64:78 に 15 ビットのバイアス付き指数 `e` が格納され、ビット 79 に符号ビット `s` が格納されています。

図 2-4 拡張倍精度形式 (x86)



4 つのフィールド f、j、e、および s 内のビットパターンの値によって、ビットパターン全体で表される値が決定されます。

表2-8「各ビットパターンによって表される値 (x86)」は、4 つの構成フィールドの 16 進数表現と、各ビットパターンによって表される値の間の対応関係を示しています。*u* は、示されているフィールドの値が特定のビットパターンの値の決定には関係しないことを示しています。

表 2-8 各ビットパターンによって表される値 (x86)

拡張倍精度のビットパターン (x86)	値
j = 0, 0 < e <32767	未サポート
j = 1, 0 < e < 32767	$(-1)^s \times 2^{e-16383} \times 1.f$ (正規数)
j = 0, e = 0; f ≠ 0 (f 内の少なくとも 1 ビットは 0 以外)	$(-1)^s \times 2^{-16382} \times 0.f$ (非正規数)
j = 1, e = 0	$(-1)^s \times 2^{-16382} \times 1.f$ (擬似非正規数)
j = 0, e = 0, f = 0 (f 内のすべてのビットが 0)	$(-1)^s \times 0.0$ (符号付き 0)
j = 1; s = 0; e = 32767; f = 0 (f 内のすべてのビットが 0)	+INF (正の無限大)
j = 1; s = 1; e = 32767; f = 0 (f 内のすべてのビットが 0)	–INF (負の無限大)
j = 1; s = *u*; e = 32767; f = .1*uuu* — *uu*	QNaN (シグナルを発生しない NaN)
j = 1; s = *u*; e = 32767; f = .0*uuu* — *uu* ≠ 0 (f 内の *u* の少なくとも 1 つは 0 以外)	SNaN (シグナルを発生する NaN)

拡張倍精度形式のビットパターンには暗黙的先行仮数ビットが含まれて*いない*ことに注意してください。拡張倍精度形式では、先行仮数ビットは個別のフィールド j として明示的に指定されます。ただし、e ≠ 0 の場合、j = 0 を含むビットパターンはすべて、このようなビットパターンを浮動小数点演算のオペランドとして使用すると無効な演算の例外が発生するという意味でサポートされていません。

拡張倍精度形式内の互いに素なフィールド j と f の和集合は、*仮数*と呼ばれます。e < 32767 かつ j = 1 の場合、または e = 0 かつ j = 0 の場合、この仮数は、先行仮数ビット j と小数部の最上位ビットの間に 2 進基数点を挿入することによって形成されます。

x86 拡張倍精度形式では、先行仮数ビット j が 0 で、かつバイアス付き指数フィールド e も 0 であるビットパターンが非正規数を表すのに対して、先行仮数ビット j が 1 で、かつバイアス付き指数フィールド e が 0 以外であるビットパターンは正規数を表します。先行仮数ビットは指数の値から推測されるのではなく、明示的に表されるため、この形式では (非正規数のように) バイアス付き指数は 0 だが、先行仮数ビットが 1 であるビットパターンも認められます。このような各ビットパターンは、実際にはバイアス付き指数フィールドが 1 である対応するビットパターン (つまり、正規数) と同じ値を表すため、これらのビットパターンは*擬似非正規数*と呼ばれます。非正規数は、IEEE 規格 754-1985 では非正規化数という名前になりました。擬似非正規数は単に、x86 拡張倍精度形式のエンコーディングのアーティファクトです。オペランドとして現れると対応する正規数に暗黙的に変換され、結果として生成されることはありません。

表 2-9 拡張倍精度形式のビットパターンとその値 (x86)

共通名	ビットパターン (x86)	10 進数値
+0	`0000 00000000 00000000`	0.0
–0	`8000 00000000 00000000`	–0.0
1	`3fff 80000000 00000000`	1.0
2	`4000 80000000 00000000`	2.0
最大の正規数	`7ffe ffffffff ffffffff`	1.189731495357231765e+4932
最小の正の正規数	`0001 80000000 00000000`	3.362103143112093506e–4932
最大の非正規数	`0000 7fffffff ffffffff`	3.362103143112093506e–4932
最小の正の非正規数	`0000 00000000 00000001`	3.645199531882474602e–4951
+∞	`7fff 80000000 00000000`	+∞
–∞	`ffff 80000000 00000000`	–∞
最大の小数部を含む、シグナルを発生しない NaN	`7fff ffffffff ffffffff`	QNaN

共通名	ビットパターン (x86)	10 進数値
最小の小数部を含む、シグナルを発生しない NaN	7fff c0000000 00000000	QNaN
最大の小数部を含むシグナルを発生する NaN	7fff bfffffff ffffffff	SNaN
最小の小数部を含む、シグナルを発生する NaN	7fff 80000000 00000001	SNaN

拡張倍精度格納形式の重要なビットパターンの例が前の表に示されています。2 番目の列のビットパターンは、1 つの 4 桁 16 進数と、それに続く 2 つの 8 桁 16 進数として示されています。前者は、最上位アドレスの 32 ビットワードの最下位 16 ビットの値です (この最上位アドレスの 32 ビットワードの最上位 16 ビットは未使用であるため、それらの値は示されていません)。後者は、左側が中央のアドレスの 32 ビットワードの値であり、右側が最下位アドレスの 32 ビットワードの値です。最大の正の正規数は、x86 拡張倍精度形式で表すことができる最大の有限数です。最小の正の非正規数は、拡張倍精度形式で表すことができる最小の正の数値です。最小の正の正規数は多くの場合、アンダーフローしきい値と呼ばれます。最大と最小の正規数および非正規数に対する 10 進数値は近似値であり、示されている桁数に関して正確です。

NaN (非数) は、NaN の定義を満足させる多くのビットパターンのいずれかで表すことができます。前の表に示されている NaN の 16 進数値は、小数部フィールドの先行 (最上位) ビットによって、NaN がシグナルを発生しない NaN (先行小数ビット = 1) またはシグナルを発生する NaN (先行小数ビット = 0) のどちらであるかが決定されることを示しています。

2.2.6 10 進数表現の範囲と精度

このセクションでは、特定の格納形式での範囲と精度の概念について説明します。ここでは、IEEE の単精度、倍精度、4 倍精度の各形式、および x86 アーキテクチャー上での IEEE 拡張倍精度形式の実装に対応する範囲と精度について説明します。範囲と精度の概念を定義する場合の具体的な例については、IEEE 単精度形式を参照してください。

IEEE 規格では、単精度形式の浮動小数点数を表すために 32 ビットを使用することが規定されています。32 個の 0 と 1 の組み合わせは有限数しか存在しないため、32 ビットでは有限数の数値しか表すことができません。

この特定の形式で表すことができる最大と最小の正の数値の 10 進数表現は何かと尋ねるのは自然なことです。

範囲の概念を導入すると、その質問を言い換えて、代わりに IEEE 単精度形式で表すことができる数値の範囲 (10 進数表記) は何かと尋ねることができます。

IEEE 単精度形式の正確な定義を考慮に入れ、正の正規化数に限定すると、IEEE 単精度形式で表すことができる浮動小数点数の範囲は次のようになると証明できます。

1.175...× (10^{-38}) から 3.402... ×(10^{+38}) まで

2 番目の質問は、特定の形式で表される数値の精度に関するものです。これらの概念は、いくつかの状況と例を調べることによって説明できます。

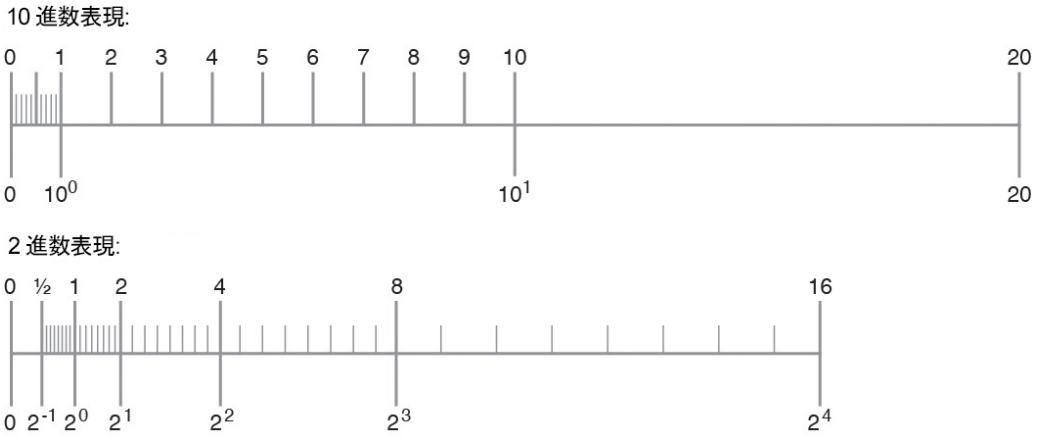
2 進浮動小数点演算のための IEEE 規格では、単精度形式で表すことができる数値のセットが規定されています。この数値のセットは、2 進浮動小数点数のセットとして説明されます。IEEE 単精度形式の仮数は 23 ビットあり、これが暗黙的先行ビットとともに 24 桁 (ビット) の (2 進数の) 精度を生み出します。

次の数値 (10 進数の仮数 q 桁で表現できます) を数直線上にマークすることによって、異なる数値のセットが取得されます。

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

次の図は、この状況を示しています。

図 2-5 デジタル表現と 2 進数表現で定義される数値のセットの比較



この 2 つのセットは異なることに注意してください。そのため、2 進数の有効桁数 24 桁に対応する 10 進数の有効桁数を推定するには、問題を再公式化する必要があります。

2 進数表現 (コンピュータによって使用される内部形式) と 10 進数形式 (ユーザーが通常関心を持つ形式) の間で浮動小数点数を変換するという観点から問題を再公式化します。実際に、10 進数から 2 進数に変換したあと 10 進数に戻したり、2 進数から 10 進数に変換したあと 2 進数に戻したりすることがあります。

数値のセットが異なるため、変換は一般に不正確である点に注意することが重要です。正しく実行された場合、あるセット内の数値からもう一方のセット内の数値への変換では、2 番目のセットから 2 つの隣接する数値のうちの 1 つが選択されます (具体的にどちらを選択するかは丸めに関連した問題です)。

いくつかの例を考察します。IEEE 単精度形式の次の 10 進数表現で数値を表そうとしているとします。

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

IEEE 単精度形式で正確に表すことができる実数は有限数しか存在せず、またその中に上の形式のすべての数値が含まれているわけではないため、一般に、このような数値を正確に表すことは不可能です。たとえば、

$y = 838861.2, z = 1.3$

として、次の Fortran プログラムを実行します。

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

このプログラムからの出力は、次のようになるはずです。

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

y に割り当てられた値 8.388612×10^5 と出力された値の差は 0.000000125 です。これは、y より 10 進数の 7 桁小さい値です。y を IEEE 単精度形式で表したときの正確性は、有効桁数が約 6 から 7 です。あるいは、y は、IEEE 単精度形式で表された場合、*有効桁数*が約 *6* です。

同様に、*z* に割り当てられた値 1.3 と出力された値の差は 0.00000004768 です。これは、*z* より 10 進数の 8 桁小さい値です。*z* を IEEE 単精度形式で表したときの正確性は、有効桁数が約 7 から 8 です。あるいは、*z* は、IEEE 単精度形式で表された場合、*有効桁数*が約 7 です。

10 進数の浮動小数点数 *a* を IEEE 単精度形式の 2 進数表現 *b* に変換したあと、*b* を 10 進数 *c* に戻すとします。*a* と *a - c* の間には何桁あるでしょうか。

この質問を次のように言い換えます。

IEEE 単精度形式表現の *a* の *10 進数の有効桁数*は何桁でしょうか。あるいは、*x* を IEEE 単精度形式で表したとき、正確であるとして信頼できる 10 進数の桁数は何桁でしょうか。

10 進数の有効桁数は常に 6 から 9 まで (つまり、少なくとも 6 桁) であり、9 桁を超えて正確であることはありません (変換が正確である場合、つまり無限数の桁を正確にできる場合を除く)。

逆に、IEEE 単精度形式の 2 進数を 10 進数に変換したあと 2 進数に戻す場合は、一般に、これらの 2 つの変換のあとに最初の数値を確実に取得できるようにするために少なくとも 10 進数 9 桁を使用する必要があります。

全体のまとめを表2-10「格納形式の範囲と精度」に示します。

表 2-10 格納形式の範囲と精度

形式	有効桁数 (2 進数)	最小の正の正規数	最大の正の数値	有効桁数 (10 進数)
単精度	24	$1.175... 10^{-38}$	$3.402... 10^{+38}$	6-9
倍精度	53	$2.225... 10^{-308}$	$1.797... 10^{+308}$	15-17
4 倍精度	113	$3.362... 10^{-4932}$	$1.189... 10^{+4932}$	33-36
拡張倍精度 (x86)	64	$3.362... 10^{-4932}$	$1.189... 10^{+4932}$	18-21

2.2.7 Oracle Solaris 環境での基数変換

基数変換とは、ある基数で表された数値を別の基数で表された数値に変換することを指します。C の `printf`、`scanf` や、Fortran の `read`、`write`、`print` などの I/O ルーチンでは、基数 2 と基数 10 で表された数値間の基数変換が必要になります。

- 基数 10 から基数 2 への基数変換は、従来の 10 進数表記の数値を読み込み、それを内部の 2 進数形式で格納する場合に発生します。

- 基数 2 から基数 10 への基数変換は、内部の 2 進数値を 10 進数の ASCII 文字列として出力する場合に発生します。

Oracle Solaris 環境では、すべての言語での基数変換のための基本的なルーチンが標準 C ライブラリ `libc` に含まれています。これらのルーチンは、関係する 10 進数の文字列の長さに対するわずかな制限に従って、すべての入力および出力形式間の正しく丸められた変換を生み出すテーブル駆動アルゴリズムを使用しています。テーブル駆動アルゴリズムでは、その正確性に加えて、正しく丸められた基数変換のための最悪条件での時間が削減されます。

1985 IEEE 規格では、10−44 から 10+44 までの範囲の大きさを持つ標準的な数値での正しい丸めが必要ですが、それを超える指数の場合は多少間違った丸めが許可されます。IEEE 規格 754 のセクション 5.6 を参照してください。`libc` のテーブル駆動アルゴリズムは、改訂された 754-2008 の要求どおり、単精度、倍精度、および拡張倍精度形式の範囲全体にわたって正しく丸めます。

C では、10 進数文字列と 2 進浮動小数点値の間の変換は、IEEE 754 に従って常に正しく丸められます。変換された結果は、現在の丸めモードによって指定された方向で元の値にもっとも近い、結果の形式で表すことができる数値です。丸めモードがもっとも近い値への丸めであり、かつ元の値が結果の形式の 2 つの表現可能な数値の間に正確に存在する場合、変換された結果は最下位の桁が偶数である方になります。これらの規則は、コンパイラによって実行されるソースコード内の定数の変換や、標準のライブラリルーチンを使用してプログラムによって実行されるデータの変換に適用されます。

Fortran では、デフォルトでは、10 進数文字列と 2 進浮動小数点値の間の変換は C と同じ規則に従って正しく丸められます。I/O 変換の場合、もっとも近い値への丸めモードの「偶数に結び付ける丸め」の規則は、プログラムで `ROUNDING=` 指定子を使用するか、または `-iorounding` フラグでコンパイルすることによってオーバーライドできます。詳細は、『Oracle Solaris Studio 12.4: Fortran ユーザーズガイド 』および `f95`(1) のマニュアルページを参照してください。

基数変換に関するリファレンス (特に Coonen 氏の論文や Sterbenz 氏の書籍) については、付録F 参考資料 を参照してください。

2.3 アンダーフロー

アンダーフローは、大まかに述べると、算術演算の結果が小さすぎるために、その目的の宛先形式に格納しようとすると通常より大きい丸め誤差を必ず引き起こしてしまう場合に発生します。

2.3.1 アンダーフローしきい値

表2-11「アンダーフローしきい値」は、単精度、倍精度、および拡張倍精度でのアンダーフローしきい値を示しています。

表 2-11 アンダーフローしきい値

宛先の精度	アンダーフローしきい値	
単精度	最小の正規数	1.17549435e-38
	最大の非正規数	1.17549421e-38
倍精度	最小の正規数	2.2250738585072014e-308
	最大の非正規数	2.2250738585072009e-308
4 倍精度	最小の正規数	3.362103143112093506262677817321752 6e-4932
	最大の非正規数	3.362103143112093506262677817321752 0e-4932
拡張倍精度 (x86)	最小の正規数	3.362103143112093506 26e-4932
	最大の非正規数	3.362103143112093505 90e-4932

正の非正規数は、最小の正規数と 0 の間にある数値です。最小の正規数に近い 2 つの (正の) 非常に小さい数値を引くと、非正規数が生成される可能性があります。または、最小の正の正規数を 2 で割ると、非正規数の結果が生成されます。

非正規数の存在によって、小さい数値が関係する浮動小数点演算により高い精度が提供されますが、非正規数自体の精度のビット数は正規数より少なくなります。数学的に正しい結果が最小の正の正規数より小さい大きさを持つときに (0 の答えを返すのではなく) 非正規数を生成する方法は、段階的アンダーフローと呼ばれます。

このような*アンダーフロー*の結果を処理するための方法は、ほかにもいくつかあります。以前は一般的であった 1 つの方法として、これらの結果を 0 にフラッシュする方法があります。この方法は*突発的アンダーフロー*と呼ばれ、IEEE 規格が現れる前はほとんどのメインフレーム上でデフォルトでした。

IEEE 規格 754 の草案を作成した数学者やコンピュータ設計者は、数学的に堅牢な解決方法への望みと、効率的に実装できる標準を作成する必要性のバランスをとりながら、いくつかの代替方法を検討しました。

2.3.2 IEEE 演算でのアンダーフローの処理方法

IEEE 規格 754 では、アンダーフローの結果を処理するための望ましい方法として段階的アンダーフローが選択されています。この方法では、格納された値のための 2 つの表現として、正規と非正規を定義します。

正規の浮動小数点数の IEEE 形式は、次のように示されます。

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

ここで、s は符号ビット、e はバイアス付き指数、f は小数部です。数値を完全に指定するために格納する必要があるのは、s、e、および f だけです。仮数の暗黙的先行ビットは正規数では 1 であると定義されているため、格納する必要はありません。

そのため、格納できる最小の正の正規数は、最大の大きさの負の指数とすべてが 0 の小数部を持っています。先行ビットを 1 ではなく 0 であると見なすことによって、さらに小さい数値にも対応できます。倍精度形式では、小数部の長さが 52 ビット (10 進数では約 16 桁) であるため、これによって最小の指数が 10-308 から実質的に 10-324 に拡張されます。これらが*非正規数*であり、アンダーフローした結果を 0 にフラッシュするのではなく、非正規数を返す方法が*段階的アンダーフロー*です。

明らかに、非正規数が小さくなるほど、その小数部に含まれる 0 以外のビットは少なくなります。非正規数の結果を生成する計算には、相対的な丸め誤差に関して、正規数のオペランドに対する計算と同じ制限は適用されません。ただし、段階的アンダーフローに関して重要な点は、その使用によって次のことが暗黙的に示されることです。

- アンダーフローした結果が、通常の丸め誤差からの結果を超える、大きな正確性の損失を招くことはありません。
- 加算、減算、比較、および剰余は、その結果が非常に小さい場合でも常に正確です。

非正規の浮動小数点数の IEEE 形式は、次のように示されます。

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

ここで、s は符号ビット、バイアス付き指数 e は 0、f は小数部です。暗黙的な 2 のべき乗のバイアスが正規数の形式でのバイアスより 1 大きく、小数部の暗黙的先行ビットが 0 であることに注意してください。

段階的アンダーフローを使用すると、表現可能な数値の小さい方の範囲を拡張できます。ある値を疑わしくするのは小ささではなく、それに関連する誤差です。非正規数を活用するアルゴリ

ズムでは、誤差制限がほかのシステムより小さくなります。次のセクションでは、段階的アンダーフローに対する数学的なある程度の正当化を示します。

2.3.3 段階的アンダーフローを使用する理由

非正規数の目的は、ほかの一部の演算モデルとは異なり、アンダーフローやオーバーフローを完全に回避することではありません。代わりに、非正規数は、さまざまな計算 (通常、乗算のあとの加算) での問題の原因としてのアンダーフローを排除します。詳細は、James Demmel 著『*Underflow and the Reliability of Numerical Software*』および S. Linnainmaa 著『*Combating the Effects of Underflow and Overflow in Determining Real Roots of Polynomials*』を参照してください。

演算に非正規数が存在するため、以降の加算や減算で、正確性の損失を招きかねない、トラップされないアンダーフローが発生することはなくなります。x と y が 2 倍以内に収まっている場合は、$x - y$ で誤差は発生しません。これは、アルゴリズム内の重要な場所で、有効な精度を実質的に高くしている多くのアルゴリズムにとって重要な点です。

さらに、段階的アンダーフローでは、アンダーフローによる誤差が通常の丸め誤差より悪化することはありません。これは、アンダーフローを処理するためのほかのどのような方法に関して行われるよりはるかに強い表明であり、この点が段階的アンダーフローに対するもっとも適切な正当化の 1 つです。

2.3.4 段階的アンダーフローの誤差の属性

浮動小数点の結果は、ほとんどの場合は丸められます。

計算された結果 = 正しい結果 + 丸め

丸めがどれだけ大きくなる場合があるかを示す便利な尺度の 1 つとして、*最終桁単位* (「unit in the last place」を略して *ulp*) と呼ばれるものがあります。浮動小数点数のその標準の表現における小数部の最下位ビットは、その*最終桁*です。このビットによって表される値 (たとえば、このビット以外は表現が同じである 2 つの数値の絶対的な違い) は、その数値の*最終桁単位*です。計算された結果が正しい結果をもっとも近い表現可能な数値に丸めることによって取得された場合は、明らかに、丸め誤差が計算された結果の最終桁単位の半分より大きくなることはありません。言い換えると、もっとも近い値への丸めモードの IEEE 演算では、次の計算結果になります。

$0 \le |\text{丸め}| \le \frac{1}{2}\text{ulp}$

ulp が相対的な量であることに注意してください。非常に大きな数値の ulp はそれ自体が非常に大きいのに対して、非常に小さい数値の ulp はそれ自体が非常に小さくなります。この関係は、ulp を関数として表すことによって明示的にすることができます。*ulp(x)* は、浮動小数点数 *x* の最終桁単位を示します。

さらに、浮動小数点数の ulp は、その数値が表される精度によって異なります。たとえば、表 2-12「4 つの異なる精度での ulp(1)」は、前に説明した 4 つの各浮動小数点形式での ulp(1) の値を示しています。

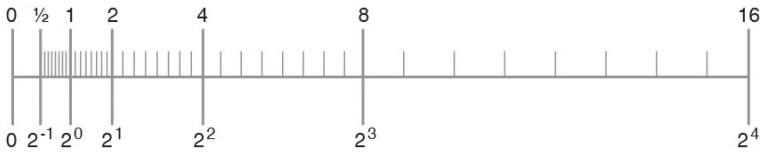
表 2-12 4 つの異なる精度での ulp(1)

精度	値
単精度	ulp(1) = 2^-23 ~ 1.192093e-07
倍精度	ulp(1) = 2^-52 ~ 2.220446e-16
拡張倍精度 (x86)	ulp(1) = 2^-63 ~ 1.084202e-19
4 倍精度	ulp(1) = 2^-112 ~ 1.925930e-34

前に説明したように、どのコンピュータ演算でも、正確に表すことができる数値のセットは有限数しか存在しません。数値の大きさが小さくなり、0 に近づくにつれて、隣接する表現可能な数値の間隔は狭くなります。逆に、数値の大きさが大きくなるにつれて、隣接する表現可能な数値の間隔は広がります。

たとえば、3 ビットの精度しかない 2 進演算を使用しているとします。この場合は、次の図に示すように、任意の 2 つの 2 のべき乗の間には $2^3 = 8$ 個の表現可能な数値があります。

図 2-6 数直線



この数直線は、指数が 1 増えるごとに数値の間隔が 2 倍になる様子を示しています。

IEEE 単精度形式では、2 つの最小の正の非正規数の大きさの違いが約 10^{-45} であるのに対して、2 つの最大の有限数の大きさの違いは実に約 10^{31} もあります。

表2-13「表現可能な単精度形式の浮動小数点の間隔」で、nextafter(x,+∞) は、数直線に沿って +∞ に向かって移動したとき、x のあとの次の表現可能な数値を示しています。

表 2-13 表現可能な単精度形式の浮動小数点の間隔

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

表現可能な浮動小数点数の従来のどのセットも、1 つの不正確な結果の最悪の影響によって、計算された結果の表現可能な近傍のいずれかとの間隔より悪化することはない誤差が導入されるという属性を備えています。表現可能なセットに非正規数が追加されたときに、段階的アンダーフローが実装されている場合は、1 つの不正確な、*またはアンダーフローした*結果の最悪の影響によって、計算された結果の表現可能な近傍のいずれかとの間隔を超えることはない誤差が導入されます。

特に、0 と最小の*正規数*の間の領域では、任意の 2 つの隣接する数値の間隔は 0 と最小の*非正規数*の間隔に等しくなります。非正規数の存在によって、もっとも近い表現可能な数値との間隔を超える丸め誤差が導入される可能性が排除されます。

どの計算も、計算された結果の表現可能な近傍のいずれかとの間隔を超える丸め誤差を招くことはないため、堅牢な演算環境の多くの重要な属性が保持されます。これには、次の 3 つが含まれます。

- $x - y \neq 0$ の場合のみ $x \neq y$
- x と y のうちの大きい方の丸め誤差以内まで、$(x - y) + y \simeq x$
- x が正規化数のとき、1/(1/x) ≃ x。つまり、最大の正規化数 x についても 1/$x \neq 0$

代替のアンダーフロースキームは、アンダーフローの結果を 0 にフラッシュする突発的アンダーフローです。突発的アンダーフローは、x – y がアンダーフローした場合は常に、最初の属性と 2 番目の属性に違反します。突発的アンダーフローはまた、1/x がアンダーフローした場合は常に、3 番目の属性にも違反します。

λが最小の正の正規化数 (アンダーフローしきい値とも呼ばれます) を表すものとします。この場合、段階的アンダーフローと突発的アンダーフローの誤差の属性はλの観点から比較できます。

段階的アンダーフロー: |誤差| < λの ½*ulp*

突発的アンダーフロー: |誤差| ≃ λ

λの最終桁単位の半分とλ自体には大きな違いがあります。

2.3.5 段階的アンダーフローと突発的アンダーフローを比較した 2 つの例

次に、2 つのよく知られた数学的な例を示します。最初の例は、内積を計算するコードです。

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

段階的アンダーフローでは、結果は丸めによって許可される範囲で正確です。突発的アンダーフローでは、もっともらしく見えるが、はるかに不適切な、0 以外の小さい総計が提供される可能性があります。ただし、こうした問題を回避するために、賢いプログラマなら、数値が小さいために正確性が低下する場所を予測できる場合は計算をスケーリングするという点を認める必要があります。

複素数の商を導き出している 2 番目の例は、スケーリングには適していません。

$$a + i \cdot b = \frac{p+iq}{r+is} \quad |r / s| \le 1 \text{ と仮定する}$$

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

丸めにもかかわらず、計算された複素数の結果が正確な結果と異なり、その差は $p + i \cdot q$ と $r + i \cdot s$ がそれぞれ数 *ulp* 程度を超えない不安定さだったとしたら正確な結果になっていた値を超えない程度であったことを示すことができます。この誤差分析は、a と b の両方がアンダーフローした場合は、誤差が $|a + i \cdot b|$ の数 *ulp* で制限される点を除き、アンダーフローが発生した

場合にも保持されます。アンダーフローが 0 にフラッシュされた場合は、どちらの結論も当てはまりません。

複素数の商を計算するためのこのアルゴリズムは堅牢であり、段階的アンダーフローが存在する場合の誤差分析に適しています。突発的アンダーフローが発生した場合に複素数の商を計算するための、同様に堅牢で、分析が容易であり、かつ効率的なアルゴリズムは*存在しません*。突発的アンダーフローでは、低レベルの複雑な詳細事項を気にすることによる負担は、浮動小数点環境の実装者からそのユーザーに移ります。

段階的アンダーフローが存在する場合は成功するが、突発的アンダーフローでは失敗する問題のクラスは、突発的アンダーフローのユーザーが認識しているより広範囲です。次に示すような、よく使用される数値技法の多くはこのクラスに分類されます。

- 線形方程式の解法
- 多項式の解法
- 数値積分
- 収束加速法
- 複素数の除算

2.3.6 アンダーフローは問題か

これらの例にもかかわらず、アンダーフローはめったに問題にならないと議論される場合もあります。問題にならないなら、悩む必要がありません。しかし、この議論は堂々巡りを招くものです。

段階的アンダーフローが存在しない場合、ユーザープログラムは、暗黙的な不正確性のしきい値に敏感になる必要があります。たとえば、単精度で計算の一部でアンダーフローが発生し、突発的アンダーフローを使用してアンダーフローした結果が `0` に置き換えられた場合は、単精度の指数での通常の低い範囲である 10^{-38} ではなく、約 10^{-31} までしか正確性を保証できません。

つまり、プログラマは、この不正確性のしきい値に近づいている時期を検出するための独自の方法を実装するか、そうでなければ自分のアルゴリズムの堅牢で、安定した実装に対する追求を諦める必要があります。

一部のアルゴリズムは、0 に近い制限された領域では計算が実行されないようにスケーリングできます。ただし、アルゴリズムのスケーリングや不正確性のしきい値の検出は、ほとんどのデータには必要がなくても、困難で、しかも時間がかかる場合があります。

2.4 IEEE 標準 754-2008

このセクションでは、754-1985 とその後継である 754-2008 との相違点について説明します。Oracle は、ほかのシステム実装者と同様に、754-2008 の推奨事項に徐々に準拠していく予定です。それは、これらの推奨事項がプログラミング言語の標準の中で定義されているためです。

2008 年、IEEE は、IEEE Standard for Floating-Point Arithmetic の改訂版を採用しました。これは、以前の 754-1985 IEEE Standard for Binary Floating-Point Arithmetic および 854-1987 IEEE Standard for Radix-Independent Floating-Point Arithmetic より優先されるものです。

ハードウェアの観点からは、新しい標準規格は古い標準規格と上位互換性があります。ソフトウェアによる拡張が十分であれば、既存のハードウェアに新しい標準規格との互換性を持たせることができます。ただし、新しい標準のすべての推奨事項を完全に実装するには、言語定義の大量の開発が必要になります。これは、数年の期間にわたって実行されており、そのあと、これらの定義の商用の実装が予定されています。改訂された標準で推奨されている新機能が Oracle Solaris Studio C、C++、および Fortran コンパイラで使用可能になったら、Oracle Solaris Studio 用のこの数値計算ガイドの将来のバージョンでそれらの機能について説明します。

Oracle Solaris Studio 12.4 にすでに存在する重要な変更のいくつかを次に示します。

- 754-2008 では、128 ビットの 2 進 (および 10 進) 浮動小数点形式が指定されています。128 ビットの 2 進形式は Studio Fortran `REAL*16` および SPARC の Studio C および C++ `long double` に対応します。
- 754-2008 では、正しい丸めには 2 進および 10 進形式とキャラクタシーケンスとの変換が必要ですが、754-1985 では、指数が非常に大きい、あるいは非常に小さい数値に対するエラーバウンドはそれよりわずかに大きい、あるいは小さいものでした。
- 754-2008 では、2 進形式と 16 進キャラクタシーケンスとの変換操作を推奨しています。
- 754-2008 では、754-1985 ではオプションであったいくつかの演算が必要です。これらは、Oracle Solaris Studio コンパイラにはすでに存在します。
- 54-2008 では、積和演算 (FMA) が規定されています。これらは、Oracle Solaris Studio 13 によってサポートされる最新の SPARC サーバー内のハードウェアで使用できます。

Oracle Solaris Studio 12.4 には存在しない重要な変更のいくつかを次に示します。

- 754-2008 では、多数の新しい演算が推奨されています。

- 754-2008 では、多数の初等超越関数で、丸めが正しいバージョンを使用可能にすることが推奨されています。
- 754-2008 では、式の評価属性が推奨されています。
- 754-2008 では、浮動小数点トラップ処理は指定されなくなりました。代わりに、マシンに依存しない方法で使用できる、高レベルの代替の例外処理属性が推奨されています。

♦ ♦ ♦ 第 3 章

3 数学ライブラリ

この章では、Oracle Solaris OS および Oracle Solaris Studio ソフトウェアで提供される数学ライブラリについて説明します。各ライブラリとその内容の一覧表示に加えて、この章では、IEEE サポート関数、乱数ジェネレータ、IEEE 形式と IEEE 以外の形式の間でデータを変換する関数を含め、コンパイラコレクションで提供されている数学ライブラリによってサポートされる機能のいくつかについて説明します。

`libm` および `libsunmath` ライブラリの内容は、*Intro*(3M) のマニュアルページにも記載されています。

この章のトピックは、次の各セクションに分かれています。

- 43 ページの「Oracle Solaris の数学ライブラリ」
- 46 ページの「Oracle Solaris Studio の数学ライブラリ」
- 49 ページの「単精度、倍精度、および拡張/4 倍精度」
- 50 ページの「IEEE サポート関数」
- 58 ページの「C99 浮動小数点環境関数」
- 62 ページの「`libm` および `libsunmath` の実装機能」

3.1 Oracle Solaris の数学ライブラリ

このセクションでは、Oracle Solaris 10 OS にバンドルされている数学ライブラリについて説明します。これらのライブラリは共有オブジェクトとして提供され、Oracle Solaris ライブラリの標準の場所にインストールされます。

3.1.1 標準数学ライブラリ

Oracle Solaris の標準数学ライブラリ `libm` には、Oracle Solaris オペレーティング環境が準拠しているさまざまな標準に必要な基本数学関数とサポートルーチンが含まれています。

Oracle Solaris 10 OS には、libm の 2 つのバージョンである libm.so.1 と libm.so.2 が含まれています。libm.so.1 は、Oracle Solaris 9 OS 以前のバージョンによってサポートされる標準に必要な関数を提供します。libm.so.2 は、Oracle Solaris 10 OS (C99 を含む) によってサポートされる標準に必要な関数を提供します。libm.so.1 は、Oracle Solaris 9 OS 以前のシステム上でコンパイルおよびリンクされたプログラムが引き続き変更なしで動作するように、下位互換性のために提供されています。libm.so.1 の内容は、これらのシステム上のセクション 3M のマニュアルページで説明されています。この章の残りの部分では、libm.so.2 について説明します。動的リンクや、プログラムが実行されるときにどの共有オブジェクトがロードされるかを決定するオプションと環境変数の詳細については、*ld*(1) とコンパイラのマニュアルページを参照してください。

表3-1「libm の内容 」は、libm 内の関数のリストを示しています。各数学関数について、この表では、倍精度バージョンの関数の名前のみを示しています。このライブラリには、同じ名前のあとに f が付いた単精度バージョンと、同じ名前のあとに l が付いた拡張/4 倍精度バージョンも含まれています。

表 3-1 libm の内容

種類	関数名
代数関数	cbrt、fdim、fma、fmax、fmin、hypot、sqrt
基本超越関数	asin、acos、atan、atan2、asinh、acosh、atanh, exp、exp2、expm1、pow、log、log1p、log10、log2、sin、cos、sincos、tan、sinh、cosh、tanh
高級超越関数	j0、j1、jn、y0、y1、yn、erf、erfc、gamma、lgamma、gamma_r、lgamma_r、tgamma
整数丸め関数	ceil、floor、llrint、llround、lrint、lround、modf、nearbyint、rint、round、trunc
IEEE 規格で推奨される関数	copysign、fmod、ilogb、nextafter、remainder、scalbn、fabs
IEEE 分類関数	isnan
旧式の浮動小数点関数	frexp、ldexp、logb、scalb、significand
エラー処理ルーチン (ユーザー定義)	matherr
複素関数	cabs、cacos、cacosh、carg、casin、casinh、catan、catanh、ccos、ccosh、cexp、cimag、clog、conj、cpow、cproj、creal、csin、csinh、csqrt、ctan、ctanh
C99 浮動小数点環境関数	feclearexcept、fegetenv、fegetexceptflag、fegetprec、fegetround、feholdexcept、feraiseexcept、fesetenv、fesetexceptflag、fesetprec、fesetround、fetestexcept、feupdateenv

種類	関数名
浮動小数点例外処理関数	fex_getexcepthandler、fex_get_handling、fex_get_log、fex_get_log_depth、fex_log_entry、fex_merge_flags、fex_setexcepthandler、fex_set_handling、fex_set_log、fex_set_log_depth
その他の C99 関数	nan、nexttoward、remquo、scalbln

表3-1「libm の内容 」について、次の点に注意してください。

1. 関数 gamma_r と lgamma_r は、gamma と lgamma の再入可能なバージョンです。

2. 関数 fegetprec と fesetprec は、x86 システム上でのみ使用できます。これらの関数は、C99 規格では規定されていません。

3. libm 内の超越関数に関する誤差制限と監視される誤差は、*libm*(3LIB) のマニュアルページで表に示されています。

3.1.2 ベクトル数学ライブラリ

ライブラリ libmvec は、引数のベクトル全体に対して共通数学関数を評価するルーチンを提供します。アプリケーションが libmvec 内のこれらのルーチンを明示的に呼び出すか、または -xvector フラグが使用されたときにコンパイラがこれらのルーチンを呼び出すことがあります。

libmvec は、プライマリ共有オブジェクト libmvec.so.1、およびベクトル関数の一部またはすべての代替バージョンを提供する複数の補助共有オブジェクトとして実装されます。libmvec でリンクされたプログラムが実行されると、実行時リンカーは、ホストプラットフォーム上で最高のパフォーマンスを提供するバージョンを自動的に選択します。このため、libmvec 内の関数を使用するプログラムは、別のシステム上で実行されると若干異なる結果を示すことがあります。

表3-2「libmvec の内容」は、libmvec 内の関数のリストを示しています。

表 3-2 libmvec の内容

種類	関数名
代数関数	vhypot_、vhypotf_、vrhypot_、vrhypotf_、vrsqrt_、vrsqrtf_、vsqrt_, vsqrtf_
指数関数および関連する関数	vexp_、vexpf_、vlog_、vlogf_、vpow_、vpowf_

種類	関数名
三角関数	vatan_、vatanf_、vatan2_、vatan2f_、vcos_、vcosf_、vsin_、vsinf_、vsincos_、vsincosf_
複素関数	vc_abs_、vc_exp_、vc_log_、vc_pow_、vz_abs_、vz_exp_、vz_log_、vz_pow_

3.2 Oracle Solaris Studio の数学ライブラリ

このセクションでは、Oracle Solaris Studio コンパイラに含まれている数学ライブラリについて説明します。

Oracle Solaris Studio 12.4 のデフォルトのベースインストールディレクトリは、`/opt/solarisstudio12.4` です。

32 ビットの静的アーカイブは、デフォルトではディレクトリ `/opt/solarisstudio12.4/lib/compilers/` にインストールされます。対応する 64 ビットの静的アーカイブは、SPARC と x86 で、それぞれ `/opt/solarisstudio12.4/lib/compilers/sparcv9/` または `/opt/solarisstudio12.4/lib/compilers/amd64/` にインストールされます。

特定の *-xarch* プロセッサ用に最適化された 32 ビットの静的アーカイブは、形式 `/opt/solarisstudio12.4/lib/compilers/`*xarch*`/` のサブディレクトリにインストールされます。

対応する 64 ビットの静的アーカイブは、`/opt/solarisstudio12.4/lib/compilers/`*xarch*`/sparcv9` または `/opt/solarisstudio12.4/lib/compilers/`*xarch*`/amd64` にインストールされます。

変数 *xarch* は、場合に応じて変更されるサポートされているバリアントのリストにある命令セットバリアントの名前です。SPARC 用の Oracle Solaris Studio 12.4 ディレクトリには、次のものが含まれています。

- `sparc`
- `sparcv9`
- `sparcvis`
- `sparcvis2`
- `sparcvis3`
- `sparcfmaf`
- `sparc4`

x86 用の Oracle Solaris Studio 12.4 ディレクトリには、次のものが含まれています。

- 386
- amd64
- sse2
- sse4_1
- sse4_2

ディレクトリ /opt/solarisstudio12.4/lib/ には、32 ビット共有オブジェクトとして提供される Oracle Solaris Studio の数学ライブラリが含まれています。

対応する 64 ビット共有オブジェクトは、/opt/solarisstudio12.4/lib/sparcv9/ または /opt/solarisstudio12.4/lib/amd64/ にインストールされます。

Oracle Solaris Studio の数学ライブラリのヘッダーファイルは、ディレクトリ /opt/solarisstudio12.4/lib/compilers/include/cc/ にインストールされます。Oracle Solaris Studio 12.4 では、以前の Oracle Solaris Studio リリースと比較して、静的アーカイブ、共有オブジェクト、およびインクルードファイル用の /opt/solarisstudio12.4/ サブディレクトリが変更されていることに注意してください。

3.2.1 Oracle 数学ライブラリ

libsunmath 数学ライブラリには、どの標準でも規定されていないが、数値ソフトウェアで役立つ関数が含まれています。また、libm.so.2 に含まれているが、libm.so.1 には含まれていない関数の多くも含まれています。libsunmath は、共有オブジェクトと静的アーカイブの両方として提供されます。

表3-3「libsunmath の内容」は、libm.so.2 には含まれていない libsunmath 内の関数のリストを示しています。各数学関数について、この表では、一般に C プログラムから呼び出される倍精度バージョンの関数の名前のみを示しています。

表 3-3 libsunmath の内容

種類	関数名
基本超越関数	exp10
度単位の三角関数	asind、acosd、atand、atan2d、sind、cosd、sincosd、tand

種類	関数名
πでスケーリングされた三角関数	asinpi、acospi、atanpi、atan2pi、sinpi、cospi、sincospi、tanpi
倍精度のπの引数還元を使用した三角関数	asinp、acosp、atanp,sinp、cosp、sincosp、tanp
財務関数	annuity、compound
整数丸め関数	aint、anint、irint、nint
IEEE 規格で推奨される関数	signbit
IEEE 分類関数	fp_class、isinf、isnormal、issubnormal、iszero
役立つ IEEE 値を提供する関数	min_subnormal、max_subnormal、min_normal、max_normal、infinity、signaling_nan、quiet_nan
加法的乱数ジェネレータ	i_addran_、i_addrans_、i_init_addrans_、i_get_addrans_、i_set_addrans_、r_addran_、r_addrans_、r_init_addrans_、r_get_addrans_、r_set_addrans_、d_addran_、d_addrans_、d_init_addrans_、d_get_addrans_、d_set_addrans_、u_addrans_
線形合同乱数ジェネレータ	i_lcran_、i_lcrans_、i_init_lcrans_、i_get_lcrans_、i_set_lcrans_、r_lcran_、r_lcrans_、d_lcran_、d_lcrans_、u_lcrans_
キャリー付き乗算乱数ジェネレータ	i_mwcran_、i_mwcrans_、i_init_mwcrans_、i_get_mwcrans_、i_set_mwcrans、i_lmwcran_、i_lmwcrans_、i_llmwcran_、i_llmwcrans_、u_mwcran_、u_mwcrans_、u_lmwcran_、u_lmwcrans、u_llmwcran_、u_llmwcrans_、r_mwcran_、r_mwcrans_、d_mwcran_、d_mwcrans_、smwcran_
乱数のシャッフル	i_shufrans_、r_shufrans_、d_shufrans_、u_shufrans_
データ変換	convert_external
丸めモードと浮動小数点例外フラグの制御	ieee_flags
浮動小数点トラップの処理	ieee_handler、sigfpe
ステータスの表示	ieee_retrospective
非標準の演算の有効化/無効化	standard_arithmetic、nonstandard_arithmetic

3.2.2 最適化されたライブラリ

libmopt ライブラリは、libm および libsunmath 内の一部の関数のより高速なバージョンを提供します。libmopt は、静的アーカイブとしてのみ提供されます。libmopt に含まれているルーチンは、libm 内の対応するルーチンを置き換えます。通常は、libmopt バージョンの方がはるかに高速です。ただし、例外的なケースの ANSI/POSIX®、SVID、X/Open、C99/IEEE のいずれの方法の処理もサポートしている libm バージョンとは異なり、libmopt ルーチンは、これらのケースの C99/IEEE の方法の処理しかサポートしていません。(付録E 標準規格への準拠を

参照してください。) また、libm 内のすべての数学関数が、浮動小数点の丸め方向モードには関係なく妥当な正確性を持つ結果を示すのに対して、libmopt 内のどの関数も、もっとも近い値への丸め以外の丸め方向を使用して呼び出した結果は未定義です。libmopt を使用するプログラムは、いずれかの標準数学関数が呼び出される場合は常に、デフォルトのもっとも近い値への丸めモードが有効になっていることを確認する必要があります。libmopt を使用してプログラムをリンクするには、-xlibmopt フラグを使用します。

3.3 単精度、倍精度、および拡張/4 倍精度

ほとんどの数値関数は、単精度、倍精度、および拡張 (x86) または 4 倍精度で使用できます。さまざまな関数の異なる精度バージョンを各言語から呼び出した例を表3-4「単精度、倍精度、および拡張/4 倍精度関数の呼び出し」に示します。

表 3-4 単精度、倍精度、および拡張/4 倍精度関数の呼び出し

言語	単精度	倍精度	拡張/4 倍精度
C、C++	`#include <math.h> float x,y, z; x = sinf(y); x = fmodf(y,z); #include <sunmath.h> float x; x = max_normalf(); x = r_addran_ ();`	`#include <math.h> double x,y, z; x = sin(y); x = fmod(y,z); #include <sunmath.h> double x; x = max_normal(); x = d_addran_();`	`#include <math.h> long double x, y,z; x = sinl(y); x = fmodl(y, z); #include <sunmath.h> long double x; x = max_normall();`
Fortran	`REAL x,y,z x = sin(y) x = r_ fmod(y,z) x = r_max_normal() x = r_addran()`	`REAL*8 x,y,z x = sin(y) x = d_ fmod(y,z) x = d_max_normal() x = d_addran()`	`REAL*16 x,y,z x = sin(y) x = q_ fmod(y,z) x = q_max_normal()`

C では、単精度関数の名前は倍精度関数の名前に f を追加することによって形成され、拡張または 4 倍精度関数の名前は l を追加することによって形成されます。Fortran の呼び出し規約は異なるため、libsunmath は、単精度、倍精度、および 4 倍精度のために、それぞれ r_…、d_…、および q_…関数を提供します。Fortran の組み込み関数は、3 つのすべての精度の汎用名で呼び出すことができます。

すべての関数に q_… のバージョンがあるわけではありません。libm および libsunmath 関数の名前と定義については、math.h および sunmath.h を参照してください。

Fortran プログラムでは、r_… 関数を real として、d_… 関数を倍精度として、および q_… 関数を REAL*16 として宣言することを忘れないでください。そうしないと、型の不一致が発生することがあります。

注記 - Oracle Solaris Studio Fortran は、拡張倍精度をサポートしていません。

3.4 IEEE サポート関数

このセクションでは、IEEE で推奨される関数、役立つ値を提供する関数、ieee_flags、ieee_retrospective、および standard_arithmetic と nonstandard_arithmetic について説明します。関数 ieee_flags と ieee_handler の詳細は、第4章「例外と例外処理」を参照してください。

3.4.1 ieee_functions(3m) および ieee_sun(3m)

ieee_functions(3m) および ieee_sun(3m) で説明される関数は、IEEE 規格に必要な機能か、またはその付録で推奨されている機能を提供します。これらは、効率的なビットマスク演算として実装されます。

表 3-5 ieee_functions(3m)

関数	説明
math.h	ヘッダーファイル
copysign(x,y)	y の符号ビットを持つ x
fabs(x)	x の絶対値
fmod(x, y)	y を基準にした x の剰余
ilogb(x)	x の基数 2 のバイアスなし指数 (整数形式)
nextafter(x,y)	方向 y への x のあとの次の表現可能な数値
remainder(x,y)	y を基準にした x の剰余
scalbn(x,n)	$x \times 2^n$

表 3-6 ieee_sun(3m)

関数	説明
sunmath.h	ヘッダーファイル
fp_class(x)	分類関数
isinf(x)	分類関数
isnormal(x)	分類関数

関数	説明
issubnormal(x)	分類関数
iszero(x)	分類関数
signbit(x)	分類関数
nonstandard_arithmetic(void)	非標準モードの有効化
standard_arithmetic(void)	標準モードの有効化
ieee_retrospective(*f)	n/a

remainder(x,y) は、IEEE 規格 754-1985 で指定された演算です。remainder(x,y) と fmod(x,y) の違いは、remainder(x,y) によって返される結果の符号が x または y のどちらかの符号と一致しない可能性があるのに対して、fmod(x,y) は常に、符号が x と一致する結果を返す点にあります。どちらの関数も正確な結果を返し、不正確の例外を生成しません。

表 3-7 Fortran からの ieee_functions の呼び出し

IEEE 関数	単精度	倍精度	4 倍精度
copysign(x,y)	t=r_copysign(x,y)	z=d_copysign(x,y)	z=q_copysign(x,y)
ilogb(x)	i=ir_ilogb(x)	i=id_ilogb(x)	i=iq_ilogb(x)
nextafter(x,y)	t=r_nextafter(x,y)	z=d_nextafter(x,y)	z=q_nextafter(x,y)
scalbn(x,n)	t=r_scalbn(x,n)	z=d_scalbn(x,n)	z=q_scalbn(x,n)
signbit(x)	i=ir_signbit(x)	i=id_signbit(x)	i=iq_signbit(x)

表 3-8 Fortran からの ieee_sun の呼び出し

IEEE 関数	単精度	倍精度	4 倍精度
signbit(x)	i=ir_signbit(x)	i=id_signbit(x)	i=iq_signbit(x)

注記 - 次の関数を使用する Fortran プログラムでは、d_*function* を倍精度として、q_*function* を REAL*16 として宣言する必要があります。

3.4.2 ieee_values(3m)

無限大、NaN、最大と最小の正の浮動小数点数などの IEEE 値は、ieee_values(3m) のマニュアルページで説明されている関数によって提供されます。表3-9「IEEE 値: 単精度」、表

3-10「IEEE 値: 倍精度」、表3-11「IEEE 値: 4 倍精度」、および表3-12「IEEE 値: 拡張倍精度 (x86)」は、ieee_values(3m) 関数によって提供される値の 10 進数値と 16 進数の IEEE 表現を示しています。

表 3-9 IEEE 値: 単精度

IEEE 値	10 進数値の 16 進表現	C、C++、Fortran
最大の正規数	3.40282347e+38 7f7fffff	r = max_normalf(); r = r_max_normal()
最小の正規数	1.17549435e-38 00800000	r = min_normalf(); r = r_min_normal()
最大の非正規数	1.17549421e-38 007fffff	r = max_subnormalf(); r = r_max_subnormal()
最小の非正規数	1.40129846e-45 00000001	r = min_subnormalf(); r = r_min_subnormal()
∞	Infinity 7f800000	r = infinityf(); r = r_infinity()
シグナルを発生しない NaN	NaN 7fffffff	r = quiet_nanf(0); r = r_quiet_nan(0)
シグナルを発生する NaN	NaN 7f800001	r = signaling_nanf(0); r = r_signaling_nan(0)

表 3-10 IEEE 値: 倍精度

IEEE 値	10 進数値の 16 進表現	C、C++、Fortran
最大の正規数	1.7976931348623157e+308 7fefffff ffffffff	d = max_normal(); d = d_max_normal()
最小の正規数	2.2250738585072014e-308 00100000 00000000	d = min_normal(); d = d_min_normal()
最大の非正規数	2.2250738585072009e-308 000fffff ffffffff	d = max_subnormal(); d = d_max_subnormal()
最小の非正規数	4.9406564584124654e-324 00000000 00000001	d = min_subnormal(); d = d_min_subnormal()
∞	Infinity 7ff00000 00000000	d = infinity(); d = d_infinity()
シグナルを発生しない NaN	NaN 7fffffff ffffffff	d = quiet_nan(0); d = d_quiet_nan(0)
シグナルを発生する NaN	NaN	d = signaling_nan(0); d = d_signaling_nan(0)

IEEE 値	10 進数値の 16 進表現	C、C++、Fortran
	7ff00000 00000001	

表 3-11 IEEE 値: 4 倍精度

IEEE 値	10 進数値の 16 進表現	C、C++ (SPARC) Fortran (すべて)
最大の正規数	1.189731495357231765085759326628007 0e+4932 7ffeffff ffffffff ffffffff ffffffff	q = max_normall(); q = q_max_normal()
最小の正規数	3.362103143112093506262677817321752 6e–4932 00010000 00000000 00000000 00000000	q = min_normall(); q = q_min_normal()
最大の非正規数	3.362103143112093506262677817321752 0e–4932 0000ffff ffffffff ffffffff ffffffff	q = max_subnormall(); q = q_max_subnormal()
最小の非正規数	6.475175119438025110924438958227646 6e–4966 00000000 00000000 00000000 00000001	q = min_subnormall(); q = q_min_subnormal()
∞	Infinity 7fff0000 00000000 00000000 00000000	q = infinityl(); q = q_infinity()
シグナルを発生しない NaN	NaN 7fff8000 00000000 00000000 00000000	q = quiet_nanl(0); q = q_quiet_nan(0)
シグナルを発生する NaN	NaN 7fff0000 00000000 00000000 00000001	q = signaling_nanl(0); q = q_signaling_nan(0)

表 3-12 IEEE 値: 拡張倍精度 (x86)

IEEE 値	10 進数値の 16 進表現 (80 ビット)	C、C++
最大の正規数	1.189731495357231765 05e+4932 7ffe ffffffff ffffffff	x = max_normall();
最小の正の正規数	3.362103143112093506 26e–4932 0001 80000000 00000000	x = min_normall();
最大の非正規数	3.362103143112093506 08e–4932 0000 7fffffff ffffffff	x = max_subnormall();
最小の正の非正規数	1.822599765941237301 26e–4951 0000 00000000 00000001	x = min_subnormall();

IEEE 値	10 進数値の 16 進表現 (80 ビット)	C、C++
∞	Infinity 7fff 80000000 00000000	x = infinityl();
シグナルを発生しない NaN	NaN 7fff c0000000 00000000	x = q
シグナルを発生する NaN	NaN 7fff 80000000 00000001	x = signaling_nanl(0);

3.4.3 ieee_flags(3m)

ieee_flags (3m) は、次のための Oracle インタフェースです。

- 丸め方向モードの問い合わせまたは設定
- 丸め精度モードの問い合わせまたは設定
- 例外発生フラグの検査、クリア、または設定

ieee_flags(3m) を呼び出すための構文は次のとおりです。

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

パラメータに指定できる値の ASCII 文字列を表3-13「ieee_flags のパラメータ値」に示します。

表 3-13 ieee_flags のパラメータ値

パラメータ	C または C++ の型	指定できるすべての値
action	char *	get、set、clear、clearall
mode	char *	direction、precision、exception
in	char *	nearest、tozero、negative、positive、extended、double、single、inexact、division、underflow、overflow、invalid、all、common
out	char **	nearest、tozero、negative、positive、extended、double、single、inexact、division、underflow、overflow、invalid、all、common

ieee_flags(3m) のマニュアルページでは、これらのパラメータが詳細に説明されています。

ieee_flags を使用して変更できる演算機能のいくつかについては、次の段落で説明します。第 4 章には、ieee_flags と IEEE 例外フラグに関する詳細情報が含まれています。

mode が `direction` である場合は、指定されたアクションが現在の丸め方向に適用されます。指定できる丸め方向は、もっとも近い値に向けた丸め、0 に向けた丸め、+· に向けた丸め、または –· に向けた丸めです。IEEE のデフォルトの丸め方向は、*もっとも近い値に向けた丸め*です。つまり、演算の数学的な結果が 2 つの隣接する表現可能な数値の間に厳密に存在する場合、数学的な結果にもっとも近い数値が提供されます。(数学的な結果が 2 つのもっとも近い表現可能な数値の正確に中央に存在する場合、提供される結果は最下位ビットが 0 である数値です。この点を強調するために、*もっとも近い値に向けた丸め*モードは、*もっとも近い偶数値への丸め*と呼ばれる場合があります。)

0 に向けた丸めは、IEEE が現れる以前の多くのコンピュータの動作方法であり、数学的には結果の切り捨てに対応しています。たとえば、2/3 が 10 進数 6 桁に丸められた場合の結果は、丸めモードがもっとも近い値に向けた丸めであるときは .666667 ですが、丸めモードが 0 に向けた丸めであるときは .666666 です。

`ieee_flags` を使用して丸め方向を検査、クリア、または設定する場合に、4 つの入力パラメータに指定できる値を表3-14「丸め方向に関する `ieee_flags` の入力値」に示します。

表 3-14 丸め方向に関する `ieee_flags` の入力値

パラメータ	指定できる値 (モードは `direction`)
`action`	`get`、`set`、`clear`、`clearall`
`in`	`nearest`、`tozero`、`negative`、`positive`
`out`	`nearest`、`tozero`、`negative`、`positive`

mode が `precision` である場合は、指定されたアクションが現在の丸め精度に適用されます。x86 ベースのシステムでは、指定できる丸め精度は単精度、倍精度、および拡張です。デフォルトの丸め精度は拡張です。このモードでは、結果を x87 浮動小数点レジスタに渡す算術演算は、その結果を拡張倍精度レジスタ形式の完全な 64 ビット精度に丸めます。丸め精度が単精度または倍精度である場合、結果を x87 浮動小数点レジスタに渡す算術演算は、その結果をそれぞれ 24 または 53 の上位ビットに丸めます。ほとんどのプログラムは少なくとも同程度に正確な結果を生成しますが、拡張の丸め精度が使用されている場合、IEEE 演算のセマンティクスへの厳格な遵守が必要な一部のプログラムは拡張の丸め精度モードでは正しく機能しないため、必要に応じて、単精度または倍精度に設定された丸め精度で実行する必要があります。

丸め精度は、SPARC プロセッサを使用したシステム上では設定できません。これらのシステムでは、*mode* = `precision` で `ieee_flags` を呼び出しても計算には影響を与えません。

最後に、*mode* が `exception` である場合は、指定されたアクションが現在の IEEE 例外フラグに適用されます。`ieee_flags` を使用して IEEE 例外フラグを検査したり、制御したりする方法の詳細は、第4章「例外と例外処理」を参照してください。

3.4.4 ieee_retrospective(3m)

`libsunmath` 関数である `ieee_retrospective` は、未処理の例外および非標準の IEEE モードに関する情報を出力します。次の内容が報告されます。

- 未処理の例外。
- 有効になっているトラップ。
- 丸め方向または精度がデフォルト以外に設定されているかどうか。
- 非標準の演算が有効かどうか。

必要な情報は、ハードウェア浮動小数点ステータスレジスタから取得されます。

`ieee_retrospective` は、設定された例外フラグ、および*トラップ*が有効になっている例外に関する情報を出力します。これらの 2 つの (関連はしていても) 個別の情報を混同しないでください。例外フラグが設定されている場合は、プログラム実行中のある時点でその例外が発生しました。例外に対してトラップが有効になっている場合は、その例外はまだ実際に発生していない可能性がありますが、発生している場合は、SIGFPE シグナルが提供されています。`ieee_retrospective` メッセージは、例外フラグが*設定されている*場合は、調査を必要としている可能性がある例外についてユーザーに警告すること、また例外の*トラップ*が有効になっている場合は、その例外がシグナルハンドラによって処理された可能性がある点をユーザーに知らせることを目的にしています。第4章「例外と例外処理」では、例外、シグナル、およびトラップについて説明するとともに、発生した例外の原因を調査する方法を示しています。

プログラムは、いつでも `ieee_retrospective` を明示的に呼び出すことができます。`-f77` 互換モードの `f95` でコンパイルされた Fortran プログラムは、自動的に `ieee_retrospective` を呼び出してから終了します。デフォルトモードの `f95` でコンパイルされた C/C++ プログラムおよび Fortran プログラムは、自動的に `ieee_retrospective` を呼び出しません。

ただし、`f95` コンパイラは共通例外に対するトラップをデフォルトで有効にするため、プログラムがトラップを明示的に無効にするか、または `SIGFPE` ハンドラをインストールしないかぎり、このような例外が発生するとプログラムがただちに異常終了することに注意してください。`-f77` 互換

モードでは、コンパイラがトラップを有効にしないため、浮動小数点例外が発生してもプログラムは実行を継続し、終了時に ieee_retrospective 出力でこれらの例外を報告します。

この関数を呼び出すための構文は次のとおりです

- C、C++ - `ieee_retrospective(`*fp*`);`
- Fortran - `call ieee_retrospective()`

C 関数の場合、引数 *fp* は、出力が書き込まれるファイルを指定します。Fortran 関数は、常に出力を stderr に出力します。

次の例は、ieee_retrospective の 6 つの警告メッセージのうちの 4 つを示しています。

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

警告メッセージは、トラップが有効になっているか、または例外が発生した場合にのみ表示されます。

ieee_retrospective メッセージは、Fortran プログラムから 3 つの方法のいずれかで抑制できます。1 つの方法は、プログラムが終了する前に、未処理の例外をすべてクリアし、トラップを無効にして、もっとも近い値への丸め、拡張精度、および標準モードに戻す方法です。これを行うには、ieee_flags、ieee_handler、および standard_arithmetic を次のように呼び出します。

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

注記 - 未処理の例外を、その原因を調査せずにクリアすることはお勧めできません。

ieee_retrospective メッセージが表示されないようにするための別の方法は、stderr のファイルへのリダイレクトです。もちろん、プログラムが ieee_retrospective メッセージ以外の出力を stderr に送信する場合は、この方法を使用してはいけません。

3 番目の方法は、たとえば次のように、プログラム内にダミーの ieee_retrospective 関数を含める方法です。

```
subroutine ieee_retrospective
```

```
return
end
```

3.4.5 nonstandard_arithmetic(3m)

第2章「IEEE 演算」で説明されているように、IEEE 演算ではアンダーフローした結果を、段階的アンダーフローを使用して処理します。SPARC ベースの一部のシステムでは、段階的アンダーフローは多くの場合、演算のソフトウェアエミュレーションを部分的に使用して実装されます。多くの計算がアンダーフローすると、これによってパフォーマンスが低下する場合があります。

特定のプログラムがこれに該当するかどうかについての何らかの情報を取得するには、ieee_retrospective または ieee_flags を使用してアンダーフロー例外が発生したかどうかを判定し、さらにプログラムによって使用されているシステム時間数をチェックできます。プログラムがオペレーティングシステムで異常に多い時間を費やし、アンダーフロー例外を発生させている場合は、段階的アンダーフローが原因である可能性があります。この場合は、IEEE 以外の演算を使用すると、プログラムの実行が速くなる可能性があります。

関数 nonstandard_arithmetic は、IEEE 以外の演算モードをサポートしているプロセッサ上で、これらの演算モードを有効にします。SPARC システムでは、この関数は、浮動小数点ステータスレジスタ内の NS (非標準の演算) ビットを設定します。SSE 命令をサポートしている x86 システムでは、この関数は、MXCSR レジスタ内の FTZ (0 へのフラッシュ) ビットを設定します。また、DAZ (非正規数が 0) ビットをサポートしているプロセッサ上の MXCSR レジスタ内のこのビットも設定します。非標準モードの影響はプロセッサによって異なり、通常は堅牢なソフトウェアの誤動作を引き起こす場合もあることに注意してください。非標準モードは、通常の使用にはお勧めできません。

関数 standard_arithmetic は、デフォルトの IEEE 演算を使用するようにハードウェアをリセットします。どちらの関数も、デフォルトの IEEE 754 の演算方法のみを提供するプロセッサには影響を与えません。このようなプロセッサの 1 つに SPARC T4 があります。

3.5 C99 浮動小数点環境関数

このセクションでは、C99 の <fenv.h> 浮動小数点環境関数について説明します。Oracle Solaris 10 OS では、これらの関数は libm で使用できます。これらは ieee_flags 関数と同

じ機能の多くを提供しますが、より自然な C インタフェースを使用しており、また C99 で定義されているために移植性も高くなります。

注記 - 動作の一貫性のために、libm 内の C99 浮動小数点環境関数および例外処理拡張機能と、libsunmath 内の ieee_flags および ieee_handler 関数の両方を同じプログラム内で使用しないでください。

3.5.1 例外フラグ関数

fenv.h ファイルは、FE_INEXACT、FE_UNDERFLOW、FE_OVERFLOW、FE_DIVBYZERO、FE_INVALID という 5 つの各 IEEE 浮動小数点例外フラグのためのマクロを定義します。さらに、マクロ FE_ALL_EXCEPT は、5 つのすべてのフラグマクロのビット単位の「or」として定義されています。以降の説明で、excepts パラメータは、5 つのいずれかのフラグマクロのビット単位の「or」か、または値 FE_ALL_EXCEPT のどちらかです。fegetexceptflag および fesetexceptflag 関数の場合、*flagp* パラメータは、型 fexcept_t のオブジェクトへのポインタである必要があります。この型は、fenv.h で定義されています。

C99 では、次の表にある例外フラグ関数が定義されています。

表 3-15 C99 規格の例外フラグ関数

関数	処理
feclearexcept(*excepts*)	指定されたフラグをクリアする
fetestexcept(*excepts*)	指定されたフラグの設定を返す
feraiseexcept(*excepts*)	指定された例外を発生させる
fegetexceptflag(*flagp, excepts*)	指定されたフラグを *flagp に保存する
fesetexceptflag(*flagp, excepts*)	指定されたフラグを *flagp から復元する

feclearexcept 関数は、指定されたフラグをクリアします。fetestexcept 関数は、設定されている excepts 引数によって指定されたフラグのサブセットに対応するマクロ値のビット単位の「or」を返します。たとえば、不正確、アンダーフロー、および 0 による除算のフラグだけが現在設定されている場合は、次の式によって i が FE_DIVBYZERO に設定されます。

```
i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
```

feraiseexcept 関数は、指定された例外のトラップのいずれかが有効になっている場合はトラップを発生させます。それ以外の場合は、単に対応するフラグを設定します。例外トラップの詳細は、第4章「例外と例外処理」を参照してください。

fegetexceptflag および fesetexceptflag 関数は、特定のフラグの状態を一時的に保存し、あとで復元するための便利な方法を提供します。特に、fesetexceptflag 関数はトラップを発生させず、単に指定されたフラグの値を復元します。

3.5.2 丸めの制御

fenv.h ファイルは、FE_TONEAREST、FE_UPWARD (正の無限大に向けて)、FE_DOWNWARD (負の無限大に向けて)、FE_TOWARDZERO という 4 つの各 IEEE 丸め方向モードのためのマクロを定義します。C99 では、丸め方向モードを制御するための 2 つの関数が定義されています。fesetround は、現在の丸め方向を引数 (上の 4 つのマクロのいずれかである必要があります) によって指定された方向に設定し、fegetround は現在の丸め方向に対応するマクロの値を返します。

x86 ベースのシステムでは、fenv.h ファイルは、FE_FLTPREC (単精度)、FE_DBLPREC (倍精度)、FE_LDBLPREC (拡張倍精度) という 3 つの各丸め精度モードのためのマクロを定義します。C99 には含まれていませんが、x86 上の libm は、丸め精度モードを制御するための 2 つの関数を提供します。fesetprec は、現在の丸め精度を引数 (上の 3 つのマクロのいずれかである必要があります) によって指定された精度に設定し、fegetprec は現在の丸め精度に対応するマクロの値を返します。

3.5.3 環境関数

fenv.h ファイルは、例外フラグ、丸め制御モード、例外処理モード、SPARC 上の非標準モードなどの、浮動小数点環境全体を表すデータ型 fenv_t を定義します。以降の説明で、*envp* パラメータは、型 fenv_t のオブジェクトへのポインタである必要があります。

C99 では、浮動小数点環境を操作するための 4 つの関数が定義されています。libm は、マルチスレッドプログラムで役立つ場合がある追加の関数を提供します。次の表に、これらの関数の要約を示します。

表 3-16 libm 浮動小数点環境関数

関数	処理
fegetenv(*envp*)	環境を *envp に保存する
fesetenv(*envp*)	環境を *envp から復元する
feholdexcept(*envp*)	環境を *envp に保存し、無停止モードを確立する

関数	処理
feupdateenv(*envp*)	環境を *envp から復元し、例外を発生させる
fex_merge_flags(*envp*)	*envp からの「or」例外フラグ

fegetenv および fesetenv 関数はそれぞれ、浮動小数点環境を保存および復元します。fesetenv への引数には、fegetenv または feholdexcept の呼び出しで以前に保存された環境へのポインタか、あるいは fenv.h で定義されている定数 FE_DFL_ENV のどちらかを指定できます。後者は、デフォルトの環境を表します。この環境では、すべての例外フラグがクリアされ、丸めがもっとも近い値に、また x86 ベースのシステム上では拡張倍精度に設定され、無停止例外処理モード (つまり、トラップが無効) と非標準モードが無効になっています。

feholdexcept 関数は現在の環境を保存してから、すべての例外フラグをクリアし、すべての例外に対して無停止例外処理モードを確立します。feupdateenv 関数は保存された環境 (これは、fegetenv または feholdexcept の呼び出しで保存された環境、あるいは定数 FE_DFL_ENV のどちらかです) を復元してから、以前の環境でフラグが設定されていた例外を発生させます。復元された環境で、これらの例外のいずれかに対してトラップが有効になっている場合はトラップが発生し、そうでない場合はフラグが設定されます。これらの 2 つの関数を組み合わせて使用すると、次のコーディング例に示すように、サブルーチン呼び出しを例外に関して不可分な動作に見せることができます。

```
#include <fenv.h>

void myfunc(...) {
    fenv_t env;

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

fex_merge_flags 関数は、保存された環境から現在の環境への例外フラグの論理的な OR を、どのトラップも発生させることなく実行します。この関数をマルチスレッドプログラムで使用すると、子スレッドで計算によって設定されたフラグに関する親スレッド内の情報を保持できます。fex_merge_flags の使用を示す例については、付録A 例を参照してください。

3.6 libm および libsunmath の実装機能

このセクションでは、libm および libsunmath の実装機能について説明します。

- 無限に正確なπ、およびπでスケーリングされた三角関数を使用した引数還元
- IEEE 形式と IEEE 以外の形式の間で浮動小数点データを変換するためのデータ変換ルーチン
- 乱数ジェネレータ

3.6.1 アルゴリズムについて

SPARC ベースのシステム上の libm および libsunmath 内の基本関数は、テーブル駆動および多項式/有理式近似アルゴリズムを使用して実装されます。これらのアルゴリズムは、パフォーマンスまたは正確性を向上させるためにリリース間で変更される可能性があります。x86 ベースのシステム上の libm および libsunmath 内の一部の基本関数は、x86 命令セットで提供される基本関数カーネル命令を使用して実装されます。その他の関数は、SPARC ベースのシステム上で使用されるのと同じテーブル駆動または多項式/有理式近似アルゴリズムを使用して実装されます。

テーブル駆動および多項式/有理式近似アルゴリズムはどちらも、libm 内の共通基本関数と libsunmath 内の単精度共通基本関数に対して、1 最終桁単位 (*ulp*) 以内まで正確な結果を提供します。SPARC ベースのシステムでは、libsunmath 内の 4 倍精度共通基本関数は、2 *ulp* 以内まで正確な結果を提供する expm1l および log1pl 関数を除き、1 *ulp* 以内まで正確な結果を提供します。(これらの共通関数には、指数関数、対数関数、べき乗関数、およびラジアン引数の循環三角関数が含まれます。双曲線三角関数や高級超越関数などのその他の関数は正確性が低くなります。)これらの誤差制限は、これらのアルゴリズムの直接の分析によって得られました。また、ユーザーは、ucbtest パッケージ http://www.netlib.org/fp/ucbtest.tgz 内の netlib から入手可能な BeEF (Berkeley Elementary Function テストプログラム) を使用して、これらのルーチンの正確性をテストすることもできます。

3.6.2 三角関数の引数還元

[−π/4,π/4] の範囲外にあるラジアン引数の三角関数は通常、π/2 の整数倍を引いて引数を示された範囲に還元することによって計算されます。

π は、マシンで表現可能な数値ではないため、近似する必要があります。最終的に計算される三角関数の誤差は、近似のπや丸めによる引数還元での丸め誤差と、還元された引数の三角関数の計算での近似誤差に依存します。かなり小さい引数の場合でも、最終的な結果の相対的な誤差より、引数還元誤差の方が大きくなることがあります。一方、かなり大きい引数の場合でも、引数還元による誤差が、その他の誤差よりは大きくならないことがあります。

すべての大きい引数の三角関数は本質的に不正確であり、すべての小さい引数の三角関数は比較的正確であるという、広く信じられている誤解があります。これは、マシンで表現可能な十分に大きい数値がπを超える間隔で区切られているという単純な観察に基づいています。

計算された三角関数の値が突然悪化するような固有の境界もなければ、不正確な関数値が役に立たなくなることもありません。引数還元が一貫して実行されている場合は、すべての基本的な同一性や関係が大きい引数に対して、小さい引数に対してと同様に保持されるため、引数還元がπへの近似によって実行されていることにはほとんど気付きません。

libm および libsunmath 三角関数は、引数還元のために「無限に」正確なπを使用します。値 2/πは 16 進数 916 桁に計算され、引数還元中に使用するルックアップテーブル内に格納されます。

関数 sinpi、cospi、および tanpi のグループ (表3-3「libsunmath の内容」を参照) は、範囲縮小によって導入される誤りを回避するために π で入力引数をスケーリングします。

3.6.3 データ変換ルーチン

libm および libsunmath には、IEEE 形式と IEEE 以外の形式の間で 2 進浮動小数点データを変換するために使用される柔軟なデータ変換ルーチン convert_external があります。

サポートされている形式には、SPARC (IEEE)、IBM PC、VAX、IBM S/370、および Cray で使用される形式が含まれます。

Cray 上で生成されるデータを取得する例や、関数 convert_external を使用して SPARC ベースのシステム上で想定される IEEE 形式にデータを変換する例については、*convert_external*(3m) のマニュアルページを参照してください。

3.6.4 乱数の機能

32 ビット整数、単精度浮動小数点、および倍精度浮動小数点形式の均一擬似乱数を生成するための機能には、次の 3 つがあります。

- *addrans*(3m) のマニュアルページで説明されている関数は、テーブル駆動加法的乱数ジェネレータのファミリに基づいています。
- *lcrans*(3m) のマニュアルページで説明されている関数は、線形合同乱数ジェネレータに基づいています。
- *mwcrans*(3m) のマニュアルページで説明されている関数は、キャリー付き乗算乱数ジェネレータに基づいています。また、これらの関数には、64 ビット整数形式の均一擬似乱数を提供するジェネレータも含まれています。

さらに、*shufrans*(3m) のマニュアルページで説明されている関数をこれらのいずれかのジェネレータと組み合わせて使用すると、擬似乱数の配列をシャッフルすることによってさらに高いランダム性を、それを必要とするアプリケーションに提供できます。64 ビット整数の配列をシャッフルするための機能はないことに注意してください。

各乱数機能には、1 回につき 1 つ (つまり、1 回の関数呼び出しで 1 つ) の乱数を生成するルーチンのほか、1 回の呼び出しで乱数の配列を生成するルーチンが含まれています。1 回につき 1 つの乱数を生成する関数は、表3-17「1 回につき 1 つ値の乱数ジェネレータの間隔」に示されている範囲内の数値を提供します。

表 3-17 1 回につき 1 つ値の乱数ジェネレータの間隔

関数	下限	上限
i_addran_	-2147483648	2147483647
r_addran_	0	0.9999999403953552246
d_addran_	0	0.9999999999999998890
i_lcran_	1	2147483646
r_lcran_	4.656612873077392578E-10	1
d_lcran_	4.656612875245796923E-10	0.9999999995343387127
i_mwcran_	0	2147483647
u_mwcran_	0	4294967295
i_llmwcran_	0	9223372036854775807
u_llmwcran_	0	18446744073709551615
r_mwcran_	0	0.9999999403953552246

関数	下限	上限
d_mwcran_	0	0.9999999999999998890

1 回の呼び出しで乱数の配列全体を生成する関数では、ユーザーは生成される数値の間隔を指定できます。付録A 例には、異なる間隔にわたって均一に分布された乱数の配列を生成する方法を示すいくつかの例が紹介されています。

addrans および mwcrans ジェネレータは一般に lcrans ジェネレータより効率的ですが、それらの理論はまだそれほど洗練されていません。『Random Number Generators: Good Ones Are Hard To Find』(S. Park 氏、K. Miller 氏共著、*Communications of the ACM*、1988 年 10 月) は、線形合同アルゴリズムの理論的な特性について説明しています。加法的乱数ジェネレータは、Knuth 氏の『*The Art of Computer Programming*』の第 2 巻で説明されています。

♦ ♦ ♦ 第 4 章

4 例外と例外処理

この章では、IEEE 浮動小数点例外について説明し、それらを検出、特定、および処理する方法を示します。また、この章では、IEEE 754 に定義されている例外とそのデフォルトの結果を一覧で示し、ステータスフラグ、トラップ、および例外処理をサポートする浮動小数点環境の機能について説明します。この章では、これらのトピックを次のセクションに分割しています。

- 67 ページの「例外処理の目的」
- 68 ページの「例外とは」
- 71 ページの「例外の検出」
- 75 ページの「例外の特定」
- 93 ページの「例外の処理」

4.1 例外処理の目的

SPARC ベースのシステムおよび x86 ベースのシステム上の Oracle Solaris Studio コンパイラおよび Oracle Solaris OS で提供される浮動小数点環境では、IEEE 標準で必要とされるすべての例外処理機能、および推奨されている多数のオプションの機能がサポートされています。IEEE 754 標準 (IEEE 854、ページ 18) では、これらの機能の目的の 1 つを次のように説明しています。

> ... ユーザーのために、例外条件の発生に伴う複雑な処理を最小限に抑えることです。算術システムは、できるだけ長時間に渡って計算を続行することを目的としており、異常な事態が発生しても、適切なフラグの設定を含む合理的なデフォルトの応答によって対処します。

この目的を達成するために、標準規格では例外演算に対するデフォルトの結果が指定されており、ユーザーが検出、設定、またはクリアできる、例外が発生したことを示すステータスフラグを実装で提供することを要求しています。例外が発生したときに、プログラムがトラップする (つまり、通常の制御フローを中断する) 方法を実装が提供することも推奨しています。例外演算に代

替の結果を渡して実行を再開するなど、適切な方法で例外を処理するトラップハンドラをプログラムで用意することもできます。以降のセクションでは、浮動小数点環境の機能がこれらの例外をサポートする方法について詳細を説明します。

4.2 例外とは

例外を定義することは困難です。W. Kahan 氏によると、

> 算術演算例外は、試行された不可分な算術演算が、一般に受け入れられるような結果にならなかったときに発生します。不可分と許容の意味は、時間と場所によって異なります。(『*Handling Arithmetic Exceptions*』、W. Kahan 著を参照してください)。

たとえば、プログラムで負の数の平方根を求めようとすると例外が発生します。この例は、*無効な演算*例外の一例です。そのような例外が発生した場合、システムは 2 つの方法のいずれかで対応します。

- 例外のトラップが無効である場合 (デフォルト) は、例外が発生したことがシステムに記録され、IEEE 754 で指定されている例外演算に関するデフォルトの結果を使用して、プログラムの実行が続行されます。
- 例外のトラップが有効である場合は、`SIGFPE` シグナルが生成されます。プログラムに `SIGFPE` シグナルハンドラがインストールされている場合は、そのシグナルハンドラに制御が移ります。シグナルハンドラが設定されていない場合は、プログラムが中止されます。

IEEE 754 は、5 種類の基本的な浮動小数点例外 (*無効な演算*、*0 による除算*、*オーバーフロー*、*アンダーフロー*、および*不正確*) を定義しています。最初の 3 つ (無効な演算、0 による除算、およびオーバーフロー) は、*一般的な例外*と呼ばれることがあります。通常、これらの例外が発生した場合は無視できません。*ieee_handler*(3m) のマニュアルページには、一般的な例外のみをトラップする簡単な方法が説明されています。ほかの 2 つの例外 (アンダーフローと不正確) はより頻繁に確認されます。実際、ほとんどの浮動小数点演算で不正確例外が発生しています。これらの例外は、通常、常にとは言えませんが、安全に無視できます。Oracle Solaris Studio 12.4 C、C++、および `f77` コンパイラは、デフォルトですべての IEEE トラップを無効にします。`f95` コンパイラは、デフォルトで一般的な例外に対してトラップを有効にします。`f95 -ftrap=none` を指定してコンパイルすると、754 標準に準拠するようになります。

表4-1「IEEE 浮動小数点例外」は、IEEE 規格 754 の情報を要約しています。5 つの浮動小数点例外、およびそれらの例外が発生したときの IEEE 演算環境のデフォルトの応答を示しています。

表 4-1 IEEE 浮動小数点例外

IEEE 例外	例外の発生理由	例	デフォルトの結果: トラップが無効な場合
無効な演算	実行しようとする演算に対してオペランドが無効 (x86 では、浮動小数点スタックがアンダーフローまたはオーバーフローした場合にもこの例外が発生しますが、このことは IEEE 規格には含まれていません。)	■ 0 × ∞ ■ 0 / 0 ■ ∞ / ∞ ■ x REM 0 ■ 負のオペランドの平方根 ■ シグナルを発生する NaN オペランドを持つ任意の演算 ■ 非順序付け比較 (注 1 を参照) ■ 無効な変換 (注 2 を参照)	シグナルを発生しない NaN
0 による除算	有限オペランドに対する演算によって結果が正確な無限数となっている。	■ 有限でゼロでない x に対する x / 0 ■ log(0)	正しい符号の無限大
オーバーフロー	正しく丸めを行なった結果、宛先形式で表現可能な最大有限数よりも絶対値が大きい (つまり、指数範囲を超えた)。	■ 倍精度: ■ DBL_MAX + 1.0e294 ■ exp(709.8) ■ 単精度: ■ (float)DBL_MAX ■ FLT_MAX + 1.0e32 ■ expf(88.8)	丸めモード (RM) と中間結果の符号に依存する。70 ページの「表 4-1 の注」の項目 4 を参照してください。
アンダーフロー	正確な結果または正しく丸められた結果のどちらも、宛先形式で表現可能な最小の正規数よりも絶対値が小さい (注 3 を参照)。	■ 倍精度: ■ nextafter(min_normal, -·) ■ nextafter(min_subnormal,-·) ■ DBL_MIN §3.0 ■ exp(-708.5) ■ 単精度: ■ (float)DBL_MIN ■ nextafterf(FLT_MIN, -·) ■ expf(-87.4)	非正規数または 0
不正確	有効な演算を丸めた結果が、無限に正確な結果と異なる(ほとんどの浮動小数点演算ではこの例外が発生します)。	■ 2.0 / 3.0 ■ (float)1.12345678 ■ log(1.1)	演算結果 (丸め、オーバーフロー、またはアンダーフロー)

IEEE 例外	例外の発生理由	例	デフォルトの結果: トラップが無効な場合
		■ DBL_MAX + DBL_MAX (オーバーフローがトラップされない場合)	

4.2.1 表 4-1 の注

1. 非順序付け比較: 任意の浮動小数値のペアは、形式が異なっていても比較できます。次の 4 つの相互に排他的な関係 (より小さい、より大きい、等しい、または非順序付け) があります。非順序付けとは、オペランドのうち少なくとも 1 つが NaN (非数) であることを意味します。

   それぞれの NaN は、その NaN 自体も含めてすべての値に対して「非順序付け」で比較されます。次の表は、関係が非順序付けのときに、無効な演算例外を発生する述語を示しています。

算術述語	C、C++ の述語	Fortran の述語	無効な式 (非順序付けの場合)
=	==	.EQ.	いいえ
≠	!=	.NE.	いいえ
>	>	.GT.	はい
≥	>=	.GE.	はい
<	<	.LT.	はい
≤	<=	.LE.	はい

2. 無効な変換: NaN、または無限大から整数に変換しようとすること、あるいは浮動小数点形式からの変換時に発生した整数値オーバーフロー。
3. IEEE の単精度、倍精度、および拡張倍精度の形式で表現可能な最小の正規数は、それぞれ 2-126、2-1022、および 2-16382 です。IEEE の浮動小数点形式については、第2章「IEEE 演算」を参照してください。
4. 次の表は、オーバーフローに対するトラップが無効にされているときのデフォルトの結果を一覧表示しています。これらの結果は、丸めモードおよび中間結果の符号によって異なります。

丸めモード	正	負
一番近い値	+∞	-∞

丸めモード	正	負
ゼロ	+∞	-max
切り捨て	+max	-∞
切り上げ	+∞	-max

x86 浮動小数点環境には、IEEE 規格にはない例外である*非正規オペランド*例外があります。この例外は、浮動小数点演算が非正規数に対して実行された場合に発生します。

例外は、次の順序で優先付けされます。無効 (もっとも高い優先度)、オーバーフロー、除算、アンダーフロー、不正確 (もっとも低い優先度)。x86 ベースのシステムでは、非正規オペランド例外はもっとも低い優先度になります。

単一の演算で同時に発生する可能性のある標準例外の組み合わせは、オーバーフローと不正確、およびアンダーフローと不正確のみです。x86 ベースのシステムでは、非正規オペランド例外は 5 つの標準例外のいずれかとともに発生することがあります。オーバーフロー、アンダーフロー、および不正確のトラップが有効になっている場合は、オーバーフローとアンダーフローのトラップが不正確のトラップよりも優先されます。これらはすべて、x86 ベースのシステムで非正規オペランドのトラップよりも優先されます。

4.3 例外の検出

IEEE 規格の要求に従い、SPARC ベースのシステムおよび x86 ベースのシステムの浮動小数点環境では、浮動小数点例外の発生を記録するステータスフラグが提供されています。プログラムでこれらのフラグをテストすると、発生した例外を判別できます。これらのフラグは、明示的に設定およびクリアすることもできます。`ieee_flags` 関数は、これらのフラグにアクセスする方法の 1 つです。C または C++ で記述されるプログラムでは、C99 浮動小数点環境関数によって別の方法が提供されています。

SPARC ベースのシステムでは、各例外には「現在」と「累積」の 2 つのフラグが関連付けられています。現在の例外フラグは、最後の浮動小数点命令の実行が完了したことによって発生した例外を常に示しています。これらのフラグは累積例外フラグにも累積され (つまり、論理和が生成されます)、それにより、プログラムの実行が開始されてから、またはプログラムによって累積フラグが最後にクリアされてから発生し、まだトラップされていないすべての例外が記録されます。浮動小数点命令がトラップされた例外の原因である場合、そのトラップを発生させた例外に対応する現在の例外フラグが設定されますが、累積フラグは変更されません。現在の例外フラグと累積例外フラグは、浮動小数点ステータスレジスタ `%fsr` 内に保持されます。

x86 ベースのシステムでは、浮動小数点ステータスワード (SW) が累積例外のフラグ、および浮動小数点スタックのステータスのフラグを提供します。SSE2 命令をサポートする x86 ベースのシステムでは、それらの命令によって発生した累積例外を記録するフラグが MXCSR レジスタに含まれています。

4.3.1 ieee_flags(3m)

`ieee_flags` は、Oracle Solaris Studio C、C++、および Fortran に似ている IEEE 754 例外フラグのインタフェースを提供しています。ただし、このインタフェースは Oracle Solaris OS でのみ使用できます。より幅広く移植できるようにする C および C++ プログラムには、74 ページの「C99 例外フラグ関数」を使用してください。

ieee_flags(3m) を呼び出す構文は次のとおりです。

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

プログラムで累積例外ステータスフラグをテスト、設定、またはクリアするには、文字列 `"exception"` を 2 番目の引数として指定して `ieee_flags` 関数を使用します。たとえば、Fortran でオーバーフロー例外フラグをクリアするには、次のように記述します。

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

C または C++ で、例外が発生したかどうかを判別するには、次のように記述します。

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

action が `"get"` の場合に *out* に返される文字列は、次のいずれかです。

- `"not available"` — 例外に関する情報を取得できない場合
- `""` (空白の文字列) — 累積例外が一度も発生していない場合、または x86 の場合は、非正規オペランドが唯一の累積例外であるとき
- 例外が発生した場合は、3 番目の引数 *in* に指定されている例外の名前が返されます
- そうでない場合は、発生したもっとも高い優先順位を持つ例外の名前

たとえば、次の Fortran の呼び出しで 0 による除算の例外が発生した場合、`out` に返される文字列は `"division"` になります。それ以外の場合は、発生したもっとも優先順位の高い例外の名前です。

```
character*8 out
```

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

in に特定の例外が指定されてない場合、in は無視されます。たとえば、次の C の呼び出しでは引数 "all" は無視されます。

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

out に例外の名前を返すことに加えて、ieee_flags は、現在発生している例外フラグをすべて組み合わせた整数値も返します。この値は、すべての累積例外フラグのビット単位の「or」であり、各フラグは 表4-2「例外ビット」 に示されているように単一のビットで表されます。各例外に対応するビットの位置は、sys/ieeefp.h ファイルに定義されている fp_exception_type 値によって示されています(これらのビットの位置はマシンによって異なり、連続しているとは限りません)。

表 4-2 例外ビット

例外	ビットの位置	累積例外ビット
invalid	fp_invalid	i & (1 << fp_invalid)
overflow	fp_overflow	i & (1 << fp_overflow)
division	fp_division	i & (1 << fp_division)
underflow	fp_underflow	i & (1 << fp_underflow)
inexact	fp_inexact	i & (1 << fp_inexact)
denormalized	fp_denormalized	i & (1 << fp_denormalized) *(x86 のみ)*

次の C または C++ のプログラムの一部は、戻り値をデコードする 1 つの方法を示しています。

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

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

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

4.3.2 C99 例外フラグ関数

C/C++ プログラムでは、C99 浮動小数点環境関数を使用して、浮動小数点の例外フラグをテスト、設定、およびクリアできます。ヘッダーファイル `fenv.h` は、5 つの標準例外 (`FE_INEXACT`、`FE_UNDERFLOW`、`FE_OVERFLOW`、`FE_DIVBYZERO`、および `FE_INVALID`) に対応する 5 つのマクロを定義しています。fenv.h は、マクロ `FE_ALL_EXCEPT` が 5 つの例外マクロすべてのビット単位の「or」となるようにも定義しています。これらのマクロを組み合わせることにより、例外フラグの任意のサブセットのテストやクリアを行なったり、例外の任意の組み合わせを発生させたりできます。次に、これらのマクロを C99 浮動小数点環境関数のいくつかと合わせて使用した例を示します。詳細は、*feclearexcept*(3M) のマニュアルページを参照してください。

注記 - 一貫した動作を保つため、`libm` の C99 浮動小数点環境関数と拡張機能、および `libsunmath` の `ieee_flags` 関数と `ieee_handler` 関数の両方を同じプログラム内で使用しないでください。

---

5 つの例外フラグすべてをクリアするには、次を使用します。

```
feclearexcept(FE_ALL_EXCEPT);
```

無効な演算フラグまたは 0 による除算フラグが発生したかどうかをテストするには、次を使用します。

```
int i;

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

`fegetexceptflag` 関数および `fesetexceptflag` 関数は、フラグのサブセットを保存および復元する方法を提供しています。次の例は、これらの 2 つの関数を使用する 1 つの方法を示しています。

```
fexcept_t flags;

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

```
}
/* restore the underflow, overflow, and inexact flags */
fesetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW, | FE_INEXACT);
```

4.4 例外の特定

例外が発生した場所を特定する方法の 1 つは、プログラム内のさまざまな箇所で例外フラグをテストすることです。ただし、この方法で正確に例外を特定するには、多くのテストが必要になり、大きなオーバーヘッドを伴う可能性があります。

例外が発生した場所を判別する簡単な方法は、例外のトラップを有効にすることです。トラップが有効にされている例外が発生すると、オペレーティングシステムは SIGFPE シグナルを送信することによってプログラムに通知します。*signal*(5) のマニュアルページを参照してください。したがって、例外のトラップを有効にすると、デバッガで実行して SIGFPE シグナルを受信した時点で停止するか、例外が発生した命令のアドレスを出力するように SIGFPE ハンドラを設定して、例外の発生箇所を判別できます。SIGFPE シグナルが生成されるようにするには、例外のトラップを有効にしておく必要があります。トラップが無効になっている場合に例外が発生すると、対応するフラグが設定され、プログラムの実行は 表4-1「IEEE 浮動小数点例外」に示されているデフォルトの結果で継続されますが、シグナルは送信されません。

4.4.1 デバッガを使用して例外を特定する

このセクションでは、dbx を使用して、浮動小数点例外の原因を調査し、例外が発生した命令を特定する方法の例を示します。dbx のソースレベルのデバッグ機能を使用するには、プログラムを -g フラグを指定してコンパイルする必要があります。詳細は、『Oracle Solaris Studio 12.4: dbx コマンドによるデバッグ 』を参照してください。

次の C プログラムについて考察します。

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

double sqrtm1(double x)
{
   return sqrt(x) - 1.0;
}

int main(void)
{
   double x, y;
```

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

このプログラムをコンパイルして実行すると、次のように出力されます。

```
-4.2  NaN
```

出力に NaN が表示されているのは、無効な演算例外が発生した可能性があることを示しています。それを判別するには、-ftrap オプションを指定して再コンパイルすることによって無効な演算のトラップを有効化し、dbx を使用してプログラムを実行して、SIGFPE シグナルが送信されたときに停止します。または、無効な演算のトラップを有効にする起動ルーチンとリンクするか、手動でトラップを有効にすると、プログラムを再コンパイルすることなく dbx を使用できます。

4.4.1.1 dbx を使用して、例外の原因となっている命令を特定する

浮動小数点例外の原因であるコードを特定するためのもっとも簡単な方法は、-g フラグおよび -ftrap フラグを指定して再コンパイルしてから、dbx を使用して例外が発生している場所を追跡することです。まず、プログラムを次のように再コンパイルします。

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

-g を指定してコンパイルすると、dbx のソースレベルのデバッグ機能を使用できるようになります。-ftrap=invalid を指定すると、無効な演算例外のトラップを有効にしてプログラムが実行されます。次に、dbx を起動して、SIGFPE が発生したときに停止するように catch fpe コマンドを発行し、プログラムを実行します。SPARC ベースのシステムでは、結果は次のようになります。

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libm.so.2
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2773)
signal FPE (invalid floating point operation) in __sqrt at 0x7fa9839c
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
Current function is sqrtm1
    5   return sqrt(x) - 1.0;
(dbx) print x
```

```
x = -4.2
(dbx)
```

この出力は、負数の平方根を求めようとした結果、sqrtm1 関数で例外が発生したことを示しています。

dbx を使用すると、ライブラリルーチンなどの -g を指定してコンパイルされていないコードで例外の原因を識別することもできます。この場合、dbx はソースファイルおよび行番号を返すことはできませんが、例外が発生した命令を示すことができます。ここでも、最初の手順では -ftrap を指定して主プログラムを再コンパイルします。

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

dbx を起動して、catch fpe コマンドを使用し、プログラムを実行します。無効な演算例外が発生すると、dbx は例外の原因となった命令の次の命令で停止します。例外の原因となった命令を特定するには、いくつかの命令を逆アセンブルし、dbx が停止した命令より前にある最後の浮動小数点命令を探します。SPARC ベースのシステムでは、結果は次の出力のようになります。

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libm.so.2
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2931)
signal FPE (invalid floating point operation) in __sqrt at 0x7fa9839c
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
(dbx) dis __sqrt+0x50/4
dbx: warning: unknown language, 'c' assumed
0x7fa98390: __sqrt+0x0050:      neg      %o4, %o1
0x7fa98394: __sqrt+0x0054:      srlx     %o2, 63, %l6
0x7fa98398: __sqrt+0x0058:      fsqrtd   %f0, %f2
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
(dbx) print $f0f1
$f0f1 = -4.2
(dbx) print $f2f3
$f2f3 = -NaN.0
(dbx)
```

この出力は、例外の原因が fsqrtd であったことを示しています。ソースレジスタを検査すると、負数の平方根を求めようとしたためにこの例外が発生したことがわかります。

x86 ベースのシステムでは、命令が固定長ではないため、コードの逆アセンブルを開始する正しいアドレスを見つけるには試行錯誤が必要になることがあります。この例では、関数の先頭付近で例外が発生しているので、ここから逆アセンブルできます。この出力は、-xlibmil フラグを

指定してプログラムがコンパイルされていることを想定しています。一般的な結果は次の出力のようになります。

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 18566)
signal FPE (invalid floating point operation) in sqrtm1 at 0x80509ab
0x080509ab: sqrtm1+0x001b:      fstpl    0xffffffe0(%ebp)
(dbx) dis sqrtm1+0x16/5
dbx: warning: unknown language, 'c' assumed
0x080509a6: sqrtm1+0x0016:      fsqrt
0x080509a8: sqrtm1+0x0018:      addl     $0x00000008,%esp
0x080509ab: sqrtm1+0x001b:      fstpl    0xffffffe0(%ebp)
0x080509ae: sqrtm1+0x001e:      fwait
0x080509af: sqrtm1+0x001f:      movsd    0xffffffe0(%ebp),%xmm0
(dbx) print $st0
$st0 = -4.2000000000000000177635683940025046 47e+00
(dbx)
```

この出力は、例外の原因が `fsqrt` 命令であったことを示しています。浮動小数点レジスタを調べると、負数の平方根を求めようとしたためにこの例外が発生したことがわかります。

4.4.1.2 再コンパイルせずにトラップを有効にする

上記の例では、`-ftrap` フラグを指定して主サブプログラムを再コンパイルすることによって、無効な演算例外のトラップを有効にしました。場合によっては、主プログラムの再コンパイルを行うことができず、ほかの方法でトラップを有効にする必要があることがあります。これを行うにはいくつかの方法があります。

`dbx` を使用しているときに、浮動小数点ステータスレジスタを直接変更することによって、トラップを手動で有効にできます。オペレーティングシステムはプログラム内で浮動小数点ユニットが最初に使用されるまで浮動小数点ユニットを使用可能にせず、その時点で浮動小数点の状態が初期化され、すべてのトラップが無効になるため、これには注多少の注意が必要です。したがって、プログラムが少なくとも 1 つの浮動小数点命令を実行するまでは、トラップを手動で有効にできません。この例では、`sqrtm1` 関数が呼び出されるまでに浮動小数点ユニットはアクセスされているので、この関数への入口にブレークポイントを設定して、無効な演算例外のトラップを有効化し、`SIGFPE` シグナルの受信時に停止するように `dbx` に指示して、実行を継続できます。SPARC ベースのシステムでの手順は次のようになります。無効な演算例外のトラップを有効にするために、`assign` コマンドを使用して `%fsr` を変更しています。

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

x86 ベースのシステムでは、同じ手順は次のようになります。

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

上記の例では、`assign` コマンドによって、浮動小数点制御ワード内の無効な演算例外がアンマスクされています (つまり、トラップを有効にしています)。プログラムで SSE2 命令を使用している場合は、MXCSR レジスタの例外をアンマスクして、それらの命令によって発生した例外のトラップを有効にする必要があります。

トラップを有効にする*初期化ルーチン*を設定すると、主プログラムを再コンパイルしたり、`dbx` を使用したりすることなくトラップを有効にできます。この方法は、デバッガで実行せずに、例外が発生したときにプログラムを中止する場合などに便利です。このようなルーチンを作成する方法は 2 つあります。

プログラムを構成するオブジェクトファイルとライブラリが使用可能な場合は、プログラムを適切な初期化ルーチンに再リンクすることによってトラップを有効にできます。最初に、次のような C のソースファイルを作成します。

```
#include <ieeefp.h>

#pragma init (trapinvalid)

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

このファイルをコンパイルしてオブジェクトファイルを作成し、元のプログラムをこのオブジェクトファイルにリンクします。

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

再リンクを行うことはできないが、プログラムが動的にリンクされている場合は、実行時リンカーの、共有オブジェクトを事前ロードする機能を使用するとトラップを有効にすることができます。SPARC ベースのシステムでこれを行うには、上記と同じ C ソースファイルを作成して、次のようにコンパイルします。

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

トラップを有効にするには、`init.so` オブジェクトのパス名を、環境変数 `LD_PRELOAD` によって指定される事前ロードする共有オブジェクトのリストに追加します。

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

共有オブジェクトの作成および事前ロードについては、『Oracle Solaris 11.2 リンカーとライブラリガイド 』を参照してください。

前述したように、浮動小数点の制御モードの初期化方法は、共有オブジェクトを事前ロードすることによって原則として変更できます。ただし、共有オブジェクト内の初期化ルーチンは、事前ロードされる場合も明示的にリンクされる場合も、メインの実行可能ファイルの一部である起動コードに制御を渡す前に、実行時リンカーによって実行されます。起動コードは、`-ftrap`、`-fround`、`-fns` (SPARC)、または `-fprecision` (x86) コンパイラフラグを介して選択される非デフォルトモードを設定し、メインの実行可能ファイルの一部である初期化ルーチン (静的にリンクされているものを含む) を実行して、最後に制御を主プログラムに渡します。このため、SPARC では次のことに留意してください。

- 上記の例で有効にされたトラップのような、共有オブジェクト内の初期化ルーチンによって設定される浮動小数点制御モードはすべて、オーバーライドされないかぎりプログラムの実行中に有効な状態のままになります。
- コンパイラフラグを介して選択された*非デフォルトモード*は、共有オブジェクト内の初期化ルーチンによって設定されるモードを無効にします (ただし、コンパイラフラグを介して選択されたデフォルトモードは、以前に設定されているモードを無効にしません)。
- メインの実行可能ファイルの一部である初期化ルーチン、または主プログラム自体によって設定されるモードは、両方とも無効にします。

x86 ベースのシステムでは状況はもう少し複雑です。通常、コンパイラによって自動的に提供される起動コードは、`__fpstart` ルーチン (標準 C ライブラリ `libc` にあります) を呼び出すことによってすべての浮動小数点モードをデフォルトにリセットしてから、`-fround`、`-ftrap`、または `-fprecision` フラグによって選択された非デフォルトモードを設定して、主プログラムに制御を渡します。そのため、初期化ルーチンを持つ共有オブジェクトを事前ロードすることによって x86 ベースのシステムでトラップを有効にしたり、ほかのデフォルトの浮動小数点モードを変更したりするには、デフォルトの浮動小数点モードを `__fpstart` ルーチンがリセットしないように、このルーチンを無効にする必要があります。ただし、代替の `__fpstart` ルーチンは、標準のルーチンが行う初期化機能の残りの部分を実行する必要があります。次のコードは、これを行うための 1 つの方法を示しています。このコードは、ホスト プラットフォームで Oracle Solaris 10 OS 以降のリリースが実行されていることを想定しています。

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

#pragma init (trapinvalid)

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

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

void __fpstart()
{
    /* perform the same floating point initializations as
       the standard __fpstart() function but leave all
       floating point modes as is */
    __flt_rounds = __fltrounds();
    (void) sysi86(SI86FPHW, &_fp_hw);

    /* set the following variable to 0 instead if the host
       platform does not support SSE2 instructions */
```

```
    _sse_hw = 1;
}
```

4.4.2 シグナルハンドラを使用して例外を特定する

前のセクションでは、例外の最初の発生を特定するためにプログラムの始めにトラップを有効にする方法をいくつか紹介しました。これに対して、プログラム内でトラップを有効にすることによって例外の特定の発生を見つけることもできます。トラップを有効にしていても、SIGFPE ハンドラをインストールしていない場合は、トラップされた例外の次の発生時にプログラムが中止されます。SIGFPE ハンドラをインストールしてある場合は、トラップされた例外が次に発生すると、システムは制御をハンドラに渡し、ハンドラは例外が発生した命令のアドレスなどの診断情報を出力して、実行を中止するか、再開します。実行を再開して意味のある結果を得るには、次のセクションで説明するように、例外演算の結果をハンドラで設定する必要がある場合があります。

ieee_handler を使用すると、5 つの IEEE 浮動小数点例外のすべてのトラップを有効にすると同時に、指定した例外が発生したときにプログラムを中止するように設定するか、SIGFPE ハンドラを設定できます。SIGFPE ハンドラは、低レベルの関数 (*sigfpe*(3)、*signal*(3c)、または *sigaction*(2)) を使用してインストールすることもできますが、ieee_handler とは異なり、これらの関数ではトラップは有効になりません。浮動小数点例外は、そのトラップが有効である場合にのみ SIGFPE シグナルをトリガーできます。

4.4.2.1 ieee_handler (3m)

ieee_handler を呼び出すための構文は、次のとおりです。

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

2 つの入力パラメータ *action* および *exception* は文字列です。3 番目のパラメータ *handler* は sigfpe_handler_type 型であり、floatingpoint.h に定義されています。

3 つの入力パラメータには次の値を指定できます。

入力パラメータ	C または C++ の型	指定できる値
action	char *	get、set、clear
exception	char *	invalid、division、overflow、 underflow、inexact、

入力パラメータ	C または C++ の型	指定できる値
ハンドラ	sigfpe_handler_type	all、common ユーザー定義のルーチン SIGFPE_DEFAULT SIGFPE_IGNORE SIGFPE_ABORT

要求された action が "set" の場合、ieee_handler は exception に指定された例外について、*handler* に指定された処理関数を設定します。処理関数は、デフォルトの IEEE 動作を選択する SIGFPE_DEFAULT または SIGFPE_IGNORE、指定した例外が発生するとプログラムを中止させる SIGFPE_ABORT、指定した例外のいずれかが発生するとサブルーチン (SA_SIGINFO フラグを設定してインストールしたシグナルハンドラのパラメータ (sigaction(2) のマニュアルページを参照) が渡される) を起動するユーザーが用意したサブルーチンのアドレスのいずれかです。ハンドラが SIGFPE_DEFAULT または SIGFPE_IGNORE の場合、ieee_handler は指定した例外のトラップを無効にし、その他のハンドラの場合、ieee_handler はトラップを有効にします。

x86 プラットフォームでは、例外のトラップが有効にされて、対応するフラグが発生するたびに、浮動小数点ハードウェアがトラップします。そのため、誤ったトラップを防ぐには、ieee_handler を呼び出してトラップを有効にする前に、指定された *exception* ごとにプログラムでフラグをクリアする必要があります。

要求された *action* が "clear" の場合、ieee_handler は指定した *exception* について現在インストールされている処理関数を取り消し、そのトラップを無効にします。これは、"set" と SIGFPE_DEFAULT を指定した場合と同じです。*action* が "clear"の場合、3 番目のパラメータは無視されます。

action が "set" および "clear" のいずれの場合も、ieee_handler は要求された操作を実行できる場合はゼロを返し、それ以外の場合はゼロ以外の値を返します。

要求された *action* が "get" の場合、ieee_handler は指定した *exception* について現在インストールされているハンドラのアドレスを返します (ハンドラがインストールされていない場合は SIGFPE_DEFAULT を返します)。

次の例は、ieee_handler の使用方法を表すいくつかのコードの一部を示しています。この C のコードは、0 による除算が発生した場合にプログラムを中止します。

```
#include <sunmath.h>
/* uncomment the following line on x86 systems */
```

```
    /* ieee_flags("clear", "exception", "division", NULL); */
    if (ieee_handler("set", "division", SIGFPE_ABORT) != 0)
        printf("ieee trapping not supported here \n");
```

Fortran の場合は次のコードのようになります。

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

次の C のコードは、すべての例外について IEEE のデフォルトの例外処理に戻します。

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

Fortran の場合は次のようになります。

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

4.4.2.2 シグナルハンドラからの例外の報告

`ieee_handler` によってインストールされた `SIGFPE` ハンドラが呼び出されると、オペレーティングシステムによって、発生した例外のタイプ、例外の原因である命令のアドレス、およびマシンの整数レジスタと浮動小数点レジスタの内容を示す追加の情報が渡されます。ハンドラはこの情報を検査して、例外と例外の発生場所を示すメッセージを出力します。

システムによって提供される情報にアクセスするには、ハンドラを次のように宣言します。この章の以降の部分では、C のコード例を示します。Fortran の `SIGFPE` ハンドラの例については、付録A 例を参照してください。

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

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

このハンドラを呼び出すと、送信されたシグナルの番号が sig パラメータに設定されます。シグナル番号は `sys/signal.h` で定義されています。`SIGFPE` のシグナル番号は 8 です。

sip パラメータは、シグナルに関する追加情報を記録する構造体を指しています。`SIGFPE` シグナルの場合、この構造体の関連メンバーは `sip->si_code` および `sip->si_addr` です (`/usr/`

include/sys/siginfo.h を参照してください)。これらのメンバーの重要性は、システム、および SIGFPE シグナルがトリガーされたイベントによって異なります。

メンバー sip->si_code は、表4-3「算術例外のタイプ」に示されている SIGFPE シグナルのタイプのいずれかです。表示されているトークンは sys/machsig.h に定義されています。

表 4-3 算術例外のタイプ

SIGFPE のタイプ	IEEE のタイプ
FPE_INTDIV	n/a
FPE_INTOVF	n/a
FPE_FLTRES	不正確
FPE_FLTDIV	除算
FPE_FLTUND	アンダーフロー
FPE_FLTINV	無効
FPE_FLTOVF	オーバーフロー

上記の表に示されているように、各 IEEE 浮動小数点例外のタイプには対応する SIGFPE シグナルタイプがあります。整数の 0 による除算 (FPE_INTDIV) および整数オーバーフロー (FPE_INTOVF) は SIGFPE のタイプにも含まれていますが、これらは IEEE 浮動小数点例外ではないため、ieee_handler でハンドラをインストールすることはできません。これらの SIGFPE タイプのハンドラは sigfpe(3) を使用すると設定できます。ただし、整数オーバーフローは、デフォルトではすべての SPARC プラットフォームおよび x86 プラットフォームで無視されます。特殊な命令によって FPE_INTOVF タイプの SIGFPE シグナルを発生させることができますが、Sun のコンパイラはこのような命令を生成しません。

IEEE 浮動小数点例外に対応する SIGFPE シグナルの場合、メンバー sip->si_code は発生した例外を示します。x86 ベースのシステムでは、実際には、フラグが発生したもっとも優先度の高いアンマスクされた例外が示されます。通常、これは最後に発生した例外と同じです。メンバー sip->si_addr は、SPARC ベースのシステムでは例外の原因である命令のアドレスを保持し、x86 ベースのシステムではトラップされた時点の命令 (通常は例外の原因である命令の次の浮動小数点命令) のアドレスを保持します。

最後に、*uap* パラメータはトラップされた時点のシステムの状態を記録する構造体を指します。この構造体の内容はシステムによって異なります。いくつかのメンバーの定義については、/usr/include/sys/siginfo.h を参照してください。

オペレーティングシステムから提供される情報を使用すると、発生した例外のタイプと例外の原因である命令のアドレスを報告する SIGFPE ハンドラを記述できます。例4-1「SIGFPE ハンドラ」は、そのようなハンドラを示しています。

例 4-1 SIGFPE ハンドラ

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

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

    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
    fprintf(stderr, "fp exception %x at address %x\n", code,
        addr);
}

int main()
{
    double  x;

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

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

SPARC システムでは、このプログラムからの出力は次のようになります。

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
Note: IEEE floating-point exception flags raised:
```

```
    Inexact;  Underflow;
IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

x86 プラットフォームでは、オペレーティングシステムが累積例外フラグのコピーを保存し、SIGFPE ハンドラを呼び出す前にそれらをクリアします。ハンドラによって保存されないかぎり、累積フラグはハンドラから戻ったときに失われます。このため、-xarch=386 を指定してコンパイルされていると、上記のプログラムからの出力にはアンダーフロー例外が発生したことが示されません。

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

ただし、デフォルトの場合、または -xarch=sse2 を指定してコンパイルされている場合は、PC がループ命令を抜けられないため、tjos テストプログラムがループします。Oracle Solaris Studio 12.4 の場合は、PC をインクリメントするコード行を追加すれば十分です。

```
uap -> UC_mcontext.gregs[REG_PC= +=5;
```

上記のコードは、-xarch=sse2 が指定されていて、SSE2 命令の長さが 5 バイトの場合にのみ使用できます。完全に汎用的な SSE2 の解決方法にするには、最適化されたコードをデコードして、次の命令の始まりを見つけます。代わりに fex_set_handling を使用してください。

多くの場合、トラップが有効であれば、例外の原因である命令は IEEE のデフォルトの結果を生成しません。上記の出力では、max_normal * max_normal に報告されている値は、オーバーフローした演算のデフォルトの結果 (つまり、正確な符号付き無限大) ではありません。通常は、意味のある値で計算を続行するために、トラップされた例外の原因である演算の結果を、SIGFPE ハンドラが提供する必要があります。これを行う 1 つの方法については、93 ページの「例外の処理」を参照してください。

4.4.3 libm の例外処理拡張機能を使用して例外を特定する

C/C++ プログラムでは、libm にある C99 浮動小数点環境関数の例外処理拡張機能を使用し、いくつかの方法で例外を特定できます。これらの拡張機能には、ieee_handler による処理と同様に、ハンドラを設定して同時にトラップを有効にできる関数が含まれていますが、これ

らの関数は ieee_handler よりも柔軟です。これらの拡張機能は、選択されたファイルに対する、浮動小数点例外に関する遡及診断メッセージのロギングもサポートしています。

4.4.3.1 fex_set_handling(3m)

fex_set_handling 関数を使用すると、浮動小数点例外のそれぞれのタイプを処理するためのいつくかのオプション (またはモード) の 1 つを選択できます。fex_set_handling を呼び出すための構文は次のとおりです。

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

引数 *ex* には、呼び出しを適用する一連の例外を指定します。この引数は、表 4-4「fex_set_handling の例外コード」の最初の列に示されている値のビット単位の「or」である必要があります(これらの値は、fenv.h に定義されています)。

表 4-4 fex_set_handling の例外コード

値	例外
FEX_INEXACT	不正確な結果
FEX_UNDERFLOW	アンダーフロー
FEX_OVERFLOW	オーバーフロー
FEX_DIVBYZERO	0 による除算
FEX_INV_ZDZ	0/0 の無効な演算
FEX_INV_IDI	無限大/無限大の無効な演算
FEX_INV_ISI	無限大-無限大の無効な演算
FEX_INV_ZMI	0*無限大の無効な演算
FEX_INV_SQRT	負数の平方根
FEX_INV_SNAN	シグナルを発生する NaN に対する演算
FEX_INV_INT	無効な整数変換
FEX_INV_CMP	無効な非順序付け比較

便宜上、fenv.h には、FEX_NONE (例外なし)、FEX_INVALID (すべての無効な演算例外)、FEX_COMMON (オーバーフロー、0 による除算、およびすべての無効な演算)、および FEX_ALL (すべての例外) の各値も定義されています。

引数 *mode* には、示された例外に設定する例外処理モードを指定します。指定可能なモードは 5 つあります。

- FEX_NONSTOP モードでは、IEEE 754 のデフォルトの停止しない動作になります。これは、例外のトラップを無効にしておくことと同等です。ieee_handler と異なり、fex_set_handling では、無効な演算例外の特定のタイプに対してデフォルト以外の処理を設定し、残りのタイプは IEEE のデフォルト処理のままにできます。
- FEX_NOHANDLER モードは、ハンドラを設定せずに例外のトラップを有効にすることと同じです。例外が発生すると、システムは、あらかじめインストールされている SIGFPE ハンドラが存在する場合はそのハンドラに制御を渡し、存在しない場合は処理を中止します。
- FEX_ABORT モードでは、例外が発生するとプログラムは abort(3c) を呼び出します。
- FEX_SIGNAL は、示された例外に対して、引数 *handler* によって指定された処理関数をインストールします。これらの例外のいずれかが発生すると、ieee_handler によってインストールされているかのように、同じ引数を使用してハンドラが呼び出されます。
- FEX_CUSTOM は、示された例外に対し、*handler* によって指定された処理関数をインストールします。FEX_SIGNAL モードと異なり、例外が発生すると、簡略化された引数リストを使用してハンドラが呼び出されます。この引数は、整数 (値は表4-4「fex_set_handling の例外コード」に示されている値の 1 つ) と、例外の原因となった演算に関する追加情報を記録する構造体を指すポインタから構成されます。この構造体の内容は、次のセクションおよび *fex_set_handling*(3m) のマニュアルページで説明されています。

指定された *mode* が FEX_NONSTOP、FEX_NOHANDLER、または FEX_ABORT である場合、*handler* パラメータは無視されます。fex_set_handling は、示された例外に対して指定されたモードが設定される場合はゼロ以外の値を返し、それ以外の場合はゼロを返します。次の例では、戻り値は無視されます。

次の例は、fex_set_handling を使用して特定のタイプの例外を見つける方法を示しています。0/0 例外で停止するようにするには、次を使用します。

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

オーバーフローおよび 0 による除算に対する SIGFPE ハンドラをインストールするには、次を使用します。

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

前の例では、前のサブセクションで示したように、ハンドラ関数は SIGFPE ハンドラに対する sip パラメータを介して渡される診断情報を出力できました。一方、次の例では、FEX_CUSTOM モードでインストールされたハンドラに渡される例外に関する情報を出力します。詳細は、*fex_set_handling*(3m) のマニュアルページを参照してください。

例 4-2 FEX_CUSTOM モードでインストールされたハンドラに渡される情報の出力

```
#include <fenv.h>

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

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

```
        break;

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

上記の例のハンドラは、発生した例外のタイプ、原因となった演算の種類、およびオペランドを報告します。このハンドラは、例外が発生した場所を示しません。例外が発生した場所を特定するには、遡及診断を使用できます。

4.4.3.2 遡及診断

libm の例外処理拡張機能を使用して例外を特定するもう 1 つの方法は、浮動小数点例外に関する遡及診断メッセージのロギングを有効にすることです。遡及診断のロギングを有効にすると、システムは特定の例外に関する情報を記録します。この情報には、例外のタイプ、その原因となった命令のアドレス、例外の処理方法、およびデバッガによって生成されるものに類似したスタックトレースが含まれます。遡及診断メッセージとともに記録されるスタックトレースには、命令のアドレスと関数名のみが含まれています。行番号、ソースファイル名、引数の値などのほかのデバッグ情報を調べるには、デバッガを使用する必要があります。

遡及診断のログには、発生した例外ごとの情報は含まれていません。例外ごとの情報を記録すると、一般にログが非常に大きくなり、特異な例外を特定することが不可能になります。代わりに、ロギングメカニズムは冗長なメッセージを取り除きます。次の 2 つの状況のいずれかである場合、メッセージは冗長と見なされます。

- 同じ例外が同じ位置 (つまり、同じ命令アドレスとスタックトレース) で前に記録されている。
- 例外に対して FEX_NONSTOP モードが有効になっていて、そのフラグが前に発生している。

具体的には、ほとんどのプログラムでは、それぞれのタイプの例外が最初に発生したときにのみログに記録されます。ある例外に対して FEX_NONSTOP 処理モードが有効な場合、任意の C99 浮動小数点環境関数を使用してそのフラグをクリアすると、その例外が次に発生したときにログに記録されます (前にログ記録された位置で発生していない場合)。

ロギングを有効にするには、fex_set_log 関数を使用してメッセージを転送するファイルを指定します。たとえば、メッセージを標準のエラーファイルに記録するには、次のように記述します。

```
fex_set_log(stderr);
```

次のコード例では、前のセクションで示した共有オブジェクトの事前ロード機能と遡及診断のロギングを組み合わせています。次の C ソースファイルを作成して、それを共有オブジェクトにコンパイルし、LD_PRELOAD 環境変数にそのパス名を指定することによってその共有オブジェクトを事前ロードして、FTRAP 環境変数に 1 つ以上の例外の名前をコンマで区切って指定すると、指定した例外の発生時にプログラムを中止すると同時に、各例外が発生した位置を示す遡及診断出力を取得できます。

例 4-3 遡及診断のロギングと共有オブジェクトの事前ロードの組み合わせ

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

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

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

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

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

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

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

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

このセクションの始めに示したプログラム例とともに上記のコードを使用すると、SPARC ベースのシステムでは、次のような結果が出力されます。

```
env FTRAP=invalid LD_PRELOAD=./init.so a.out
Floating point invalid operation (sqrt) at 0x7fa98398 __sqrt, abort
  0x7fa9839c  __sqrt
  0x00010880  sqrtm1
  0x000108ec  main
Abort
```

上記の出力は、ルーチン `sqrtm1` 内の平方根演算の結果として無効な演算例外が発生したことを示しています。

前述したように、x86 プラットフォームで共有オブジェクトの初期化ルーチンからトラップを有効にするには、標準の `__fpstart` ルーチンを無効にする必要があります。

典型的なログの出力を示した例については、付録A 例を参照してください。一般的な情報については、*fex_set_log*(3m) のマニュアルページを参照してください。

4.5 例外の処理

歴史的に、ほとんどの数値計算ソフトウェアは例外を考慮せずに作成されてきており、多くのプログラマは、例外が発生するとプログラムがただちに中止するという環境に慣れていました。LAPACK などの高品質なソフトウェアパッケージでは、0 による除算や無効な演算などの例外を回避し、入力を積極的に位取りしてオーバーフローや結果が不正確になる可能性のあるアンダーフローを除外するように注意深く設計されています。例外を処理するためのこれらの方法はいずれも、あらゆる状況で適切であるわけではありません。ただし、例外を無視すると、あるプログラマが記述したプログラムやサブルーチンを (たとえば、ソースコードにアク

セスできない) ほかのプログラマが使用する場合に、問題となることがあります。すべての例外を回避しようとすると、それに対応するための多くのテストと分岐が必要になり、非常に手間がかかる可能性があります。詳細は、『Faster Numerical Algorithms via Exception Handling』、Demmel、Li 共著、IEEE Trans. Comput. 43 発行 (1994 年) の 983–992 ページを参照してください。

第 3 の選択肢として、IEEE 算術演算のデフォルトの例外応答、ステータスフラグ、およびオプションのトラップ機能によって、例外が発生しても計算を続行してあとで例外を検出するか、発生時に割り込んで処理できます。前述のように、`ieee_flags` や C99 浮動小数点環境関数を使用してあとで例外を検出したり、`ieee_handler` や `fex_set_handling` を使用してトラップを有効にし、例外が発生したときにそれに割り込む SIGFPE ハンドラをインストールできます。ただし、計算を続行するために、IEEE 規格では、例外の原因となった演算の結果をトラップハンドラが提供できるようにすることを推奨しています。`FEX_SIGNAL` モードで `ieee_handler` または `fex_set_handling` を介してインストールされる `SIGFPE` ハンドラは、Solaris オペレーティング環境がシグナルハンドラに渡す *uap* パラメータを使用してこれを実現しています。`fex_set_handling` を介してインストールされる `FEX_CUSTOM` モードハンドラは、このようなハンドラに渡される info パラメータを使用して結果を提供できます。

C では、`SIGFPE` シグナルハンドラを次のように宣言できます。

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

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

トラップされた浮動小数点例外の結果として `SIGFPE` シグナルハンドラが呼び出されると、*uap* パラメータは、そのマシンの整数レジスタおよび浮動小数点レジスタのコピー、およびその他の例外が記述されているシステム依存の情報が含まれているデータ構造体を指します。このシグナルハンドラが正常に返されると、保存されたデータが復元され、トラップが行われた箇所からプログラムの実行が再開されます。このように、例外を記述したデータ構造体の情報にアクセスして復号化し、可能であれば保存されたデータを変更することによって、`SIGFPE` ハンドラは例外演算の結果をユーザーが指定した値に置換して計算を続行できます。

`FEX_CUSTOM` モードハンドラは、次のように宣言できます。

```
#include <fenv.h>

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

```
}
```

FEX_CUSTOM ハンドラが呼び出されるとき、ex パラメータは発生した例外のタイプ (表 4-4「fex_set_handling の例外コード」 に示されている値の 1 つ) を示し、*info* パラメータはその例外の詳細情報を含むデータ構造を指します。具体的には、このデータ構造には、例外の発生原因である算術演算を表すコードと、オペランドを記録する構造体 (利用できる場合) が含まれています。また、例外がトラップされなかった場合に置換されていたはずのデフォルトの結果を記録する構造体、および累積されたはずの例外フラグのビット単位の「or」を保持する整数値も含まれています。ハンドラは、構造体の後者のメンバーを変更して異なる結果に置き換えたり、累積された一連のフラグを変更したりできます。(これらのデータを変更せずにハンドラが戻った場合、プログラムは、例外がトラップされなかったかのように、デフォルトのトラップされていない結果とフラグを使用して続行します)。

次のセクションでは、アンダーフローまたはオーバーフローになる演算を位取りされた結果に置換する方法を示します。その他の例については、付録A 例を参照してください。

4.5.1 IEEE トラップされたアンダーフローおよびオーバーフローの置換

IEEE 規格では、アンダーフローおよびオーバーフローがトラップされた場合、*指数がラップされた*結果 (つまり、指数がその通常の範囲の終わりでラップされていることを除けば、オーバーフローまたはアンダーフローした演算を丸めた結果と一致する値) をトラップハンドラが置換する方法をシステムが提供して、結果を 2 のべき乗によって位取りすることを推奨しています。以降の計算でアンダーフローやオーバーフローが発生しないように、指数範囲の中央のできるだけ近くにアンダーフローまたはオーバーフローした結果を割り当てるような位取りが選択されます。発生したアンダーフローやオーバーフローの回数を追跡することによって、プログラムは最終的な結果を位取りし、ラップされた指数を補正できます。このアンダーフロー/オーバーフローの「カウントモード」は、有効な浮動小数点形式の範囲を超えてしまうような計算において、正確な結果を生成するために使用できます。詳細は、『*Floating-Point Computation*』、P. Sterbenz 著を参照してください。

SPARC ベースのシステムでは、浮動小数点命令がトラップされた例外の原因である場合、システムは宛先レジスタを変更しません。このため、指数がラップされた結果を置換するには、アンダーフロー/オーバーフローのハンドラが命令をデコードし、オペランドレジスタを検査して、位取りされた結果自体を生成する必要があります。次の例では、この手順を実行するハンドラを示します。

例 4-4 SPARC ベースのシステムでの、IEEE トラップされたアンダーフロー/オーバーフローハンドラの結果の置換

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

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

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

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

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

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

    case 2: /* double precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
```

```
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds1 = *(double*)&FPxreg(rs1 & 0x1e);
        }
        else
            ds1 = *(double*)&FPreg(rs1);
        if (rs2 & 1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
        fd = (float) (fscl * (fscl * qs1));
        break;

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

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

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

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

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

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

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

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

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

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

この例で変数 a、b、x、および y が volatile と宣言されているのは、コンパイラがコンパイル時に a * b などを評価することを防ぐためにすぎません。通常の使用では、volatile 宣言は必要ありません。

上記のプログラムの出力は、次のとおりです。

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note: IEEE floating-point exception traps enabled:
```

```
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

x86 ベースのシステムでは、浮動小数点命令によってアンダーフローまたはオーバーフローがトラップされ、その宛先がレジスタである場合、浮動小数点ハードウェアによって指数がラップされた結果が提供されます。ただし、トラップされたアンダーフローまたはオーバーフローが浮動小数点のストア命令で発生した場合、ハードウェアはストアを完了させずにトラップを行い、そのストア命令がストアおよびポップである場合は、スタックのポップも行いません。このため、カウントモードを実装するには、アンダーフロー/オーバーフローのハンドラが位取りされた結果を生成し、ストア命令でトラップが発生した場合は、スタックを修正する必要があります。例4-5「x86 ベースのシステムでの IEEE トラップされたアンダーフロー/オーバーフローハンドラの結果の置換」はそのようなハンドラを示しています。

例 4-5 x86 ベースのシステムでの IEEE トラップされたアンダーフロー/オーバーフローハンドラの結果の置換

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

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

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

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

    /* preserve flags for untrapped exceptions */
    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status;
    FPenv(SW) |= (sw & (FPenv(CW) & 0x3f));
    /* if the excepting instruction is a store, scale the stack
```

```
 top, store it, and pop the stack if need be */
fpsetmask(0);
    op = FPenv(OP) >> 16;
    switch (op & 0x7f8) {
    case 0x110:
    case 0x118:
    case 0x150:
    case 0x158:
    case 0x190:
    case 0x198:
        fscl = scalbnf(1.0f, (sip->si_code == FPE_FLTOVF)?
 -96 : 96);
*(float *)FPenv(EA) = (FPreg(0) * fscl) * fscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
        break;

    case 0x510:
    case 0x518:

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

```
    }
}

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

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

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

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

SPARC ベースのシステムの場合および -xarch=386 を指定してコンパイルされている場合と同様に、x86 での上記のプログラムの出力は次のようになります。

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

注記 - -xarch=sse2 を指定した場合、このプログラムはループします。-xarch=sse2 の場合は、完全に記述しなおす必要があることがあります。

C/C++ プログラムでは、libm にある fex_set_handling 関数を使用すると、アンダーフローおよびオーバーフローに対する FEX_CUSTOM ハンドラをインストールできます。SPARC ベースのシステムでは、このようなハンドラに渡される情報には、例外の原因である演算およびオペランドが常に含まれており、上記に示したように、ハンドラは、この情報を使用して IEEE の指

数がラップされた結果を計算できます。x86 ベースのシステムでは、例外の原因である特定の演算、および超越命令の 1 つが例外を発生させる時点 (たとえば、info->op パラメータが fex_other に設定された) を、提供される情報が常に示しているとは限りません(定義については、fenv.h ファイルを参照してください)。また、x86 のハードウェアは指数がラップされた結果を自動的に提供するため、例外を発生させている命令の宛先が浮動小数点レジスタである場合は、オペランドの 1 つが上書きされる場合があります。

幸いなことに、fex_set_handling 機能は、FEX_CUSTOM モードでインストールされているハンドラがアンダーフローまたはオーバーフローした演算を IEEE の指数がラップされた結果に置換する簡単な方法を提供しています。これらの例外のいずれかがトラップされたときに、ハンドラは次のように設定できます。

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

これにより、指数がラップされた結果を提供することが示されます。次に、そのようなハンドラの例を示します。

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

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

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

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

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

```
    return 0;
}
```

上記のプログラムの出力は、次のようになります。

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

上記のプログラムの例は、SPARC、`-xarch=sse2` を指定した x86、および `-xarch=386` を指定した x86 上で動作します。

♦ ♦ ♦ 第 5 章

5 コンパイラコードの生成

この章では、特に数値計算に関連する Oracle Solaris Studio 12.4 コンパイラのコンパイラコード生成機能について説明します。この章のトピックは次のとおりです。

- 107 ページの「サポートされているオペレーティングシステム、ハードウェア、およびメモリーモデル」
- 108 ページの「コード生成オプション」
- 109 ページの「デフォルトのアドレスモデルとコード生成」
- 110 ページの「コンパイルオプション」
- 111 ページの「再現可能な結果」
- 114 ページの「独立した確認」

5.1 サポートされているオペレーティングシステム、ハードウェア、およびメモリーモデル

Oracle Solaris Studio 12.4 は、Update 10 以降の Oracle Solaris 10、Oracle Solaris 11、Oracle および Red Hat Enterprise Linux リリース 5 および 6 をサポートしています。

Oracle Solaris Studio は、対応する Oracle Solaris リリースと同じハードウェアをサポートしています。SPARC の場合、Oracle Solaris 10 および 11 は 64 ビットアドレス空間メモリーモデルをサポートする SPARC プロセッサのみをサポートします。x86 の場合、Oracle Solaris 11 は 64 ビットアドレス空間メモリーモデルをサポートする x86 プロセッサのみをサポートします。Oracle Solaris 10 は、32 ビットアドレス空間メモリーモデルのみをサポートする多くの x86 プロセッサもサポートします。

すべての 64 ビットプロセッサは、32 ビットと 64 ビットのどちらのアドレス空間用にコンパイルされたプログラムでも実行できます。Oracle Solaris 10 および 11 は、64 ビットオペレーティングシステムでの 32 ビットプログラムの実行をサポートしています。

32 ビットおよび 64 ビットのアドレス指定は、コンパイル時に -m32 および -m64 コマンド行オプションで選択されます。これらは C の整数型およびポインタ変数のサイズに影響します。オペレーティングシステムには、32 ビットおよび 64 ビットのランタイムライブラリが複数用意され、コンパイラは特定の言語用に追加ライブラリを提供します。

64 ビットアドレス空間を必要とするプログラムは、-m64 を使用してコンパイルする必要があります。ほとんどのプログラムは、どのアドレスモデルでもコンパイル可能であり、正しく実行できるため、当然、どのモデルがより速いかが問題になります。大量の整数とポインタデータをメモリーに対して出し入れする C プログラムでは、-m64 を使用すると速度が半減する可能性があります。ただし、64 ビットアプリケーションバイナリインタフェース (ABI) は 32 ビット ABI より多くのレジスタを含むため、必要なメモリー移動が少なくてすむ可能性があります。ほとんどのプログラムにとって、パフォーマンスに大きな差はありませんが、特定のプログラムの場合は、両方の方法でコンパイルし、正確さとパフォーマンスのテストを実施して確認することをお勧めします。

注記 - Sun Studio 11 以前のリリースでは、メモリーモデルは、-m32 や -m64 などの明示的なオプションではなく、メモリーモデルに応じて異なる名前が付けられた -xarch オプションに組み込まれていました。Sun Studio 12 では、メモリーモデルオプションとアーキテクチャーオプションは切り離されています。

5.2 コード生成オプション

Oracle Solaris Studio 12.4 は多くのさまざまな SPARC および x86 プロセッサチップをサポートしています。これらの各プロセッサチップについて、-xtarget= コマンド行コンパイラオプションがあります。--xtarget= オプションは、実装されている命令セット、その命令セットを実装している特定のプロセッサチップ、および各種キャッシュのサイズを指定します。-xtarget は次のオプションのマクロとして機能します。

-xarch=*architecture* コンパイラは -xarch= オプションを使用してコードを最適化し、どの命令がハードウェアに実装され、コード生成に適しているかを判断します。

-xchip=*chip* コンパイラは -xchip= オプションを使用してコードを最適化し、どの特定のチップが対象とされているか、命令をどのようにスケジュール設定する必要があるかを判断します。

-xcache=*cache-size* コンパイラは -xcache= オプションを使用してコードを最適化し、メモリートラフィックを最小限に抑えるためにループをどのようにブロックするかを判断します。

特定のターゲットに合わせて最適化すると、そのターゲットに最適なコードが得られますが、別のターゲットで命令セットとスケジューリングの制約が異なる場合にはまったく適さないことがあります。ある実行可能ファイルをさまざまなターゲットシステム上で実行することを目的としている場合、最適なものはデフォルトの一般的なコード生成であり、これは、オプション `-xtarget=generic` で明示的に選択することもできます。

`-xtarget=` 名の中にはわかりにくいものもあります。特定のターゲットを指定するには、Oracle Solaris Studio コンパイラで `-native` オプションを使用できます。これにより、コンパイルされているシステムに `-xtarget=` が自動的に選択されます。SPARC システムでは、同様の情報が `fpversion` コマンドによって表示されます。詳細は、付録B SPARC の動作と実装を参照してください。

5.3 デフォルトのアドレスモデルとコード生成

Oracle Solaris Studio 12.4 以前のリリースでは、デフォルトのアドレス空間モデルは、Oracle Solaris OS (32 ビット) の `-m32` です。ただし、Linux では、デフォルトは 32 ビットハードウェアの場合が `-m32`、64 ビットハードウェアの場合が `-m64` です。

SPARC の場合、デフォルトは `-xarch=sparc` です。

x86 の場合、デフォルトは `-xarch=sse2` です。

注記 - 以前の Studio リリースでは、x86 のデフォルトは -m32 に対して -xarch-386、-m64 に対して -xarch=sse2 でした。

新しい x86 のデフォルトである `-m32 -xarch=sse2` は、以前のデフォルトである `-m32 -xarch=386` と同じ ABI を実装します。浮動小数点オペランドと結果は、x87 浮動小数点レジスタで渡されます。ただし、次の単精度および倍精度浮動小数点演算は通常、`sse2` レジスタで実行されます。

- `+`
- `-`
- `*`
- `/`
- `sqrt`
- `convert`

x87 レジスタは、引き続き 4 倍精度演算とハードウェア初等超越関数の評価に使用されます。

これは、同じハードウェアとオペレーティングシステムであっても、x86 デフォルトでコンパイルした浮動小数点計算の結果が、以前の Studio リリースに比べ Oracle Solaris Studio 12.4 でわずかに異なる可能性があることを意味します。

5.4 コンパイルオプション

Oracle Solaris Studio 12.4 コンパイラは、コード生成に影響する多数のオプションを受け入れます。次のリストでは、一部のプログラムにのみ有効な特定のコード生成オプションに焦点を当てています。長年にわたって多くの作成者が記述した大規模なプログラムの場合、プログラムのロジックに悪影響を及ぼす、または及ぼさないコード変換がどれかということについてだれ一人として断言できないことはよくあります。このため、次のオプションは慎重に使用する必要があります。

`-fast`
: 一般に有用な、さまざまな変換に使用するマクロ。`-fast` はすべての構成要素を記憶するより簡単に記憶できます。`-fast` の定義はリリースごとに異なります。その変換のすべてがあらゆるプログラムに妥当なわけではありません。`-fast` を使用する場合、その効果の一部を元に戻す追加オプションを続けることができます。たとえば、`-fast -fsimple=0 -fns=no -xvector=no` を使用すると、次に説明する 3 つのオプションが無効になります。特定のコンパイラのコマンド行オプションによって実際にどのオプションが有効になるかを確認するには、`-dryrun` を付けてコンパイルすることをお勧めします。

`-fsimple`
: 一部のプログラムには該当しない浮動小数点モード、例外、および丸めに関する特定の単純化した仮定を許可します。コンパイラはこれらの仮定を使用して、リリース間で異なる、値の変動する各種変換を位置調整します。`-fsimple=0` はデフォルトでもっとも安全です。`-fsimple=1` はほとんどのプログラムで安全であり、`-fsimple=2` はほとんどの場合に危険を伴います。わずかな入力でプログラムの実行をテストするだけでは、`-fsimple=2` が適しているかどうか検証には十分ではありません。その結果は一部の入力データでは有益でも、ほかの入力データに対しては有益でない可能性があります。

`-fns`
: コード生成には影響しませんが、非標準アンダーフローモードを有効にしてプログラムの実行を開始します。これは IEEE 標準に反しているため、無効な結果や無限ループなど、結果が非標準になることがあります。多くのプログラムは、アンダーフローがどのように処理されるかにかかわらず、同じように実行します。標準アンダーフローに対して実行時ハードウェアが低速な場合、これらのプログラムは、`-fns` を使用してより高速に実行する

ことができます。最近の SPARC サーバーでは、非標準モードのパフォーマンス上のメリットはありません。

`-ftrap=common` コード生成には影響しませんが、オーバーフロー、ゼロ除算、および無効な IEEE 例外についてトラップを有効にしてプログラムの実行を開始します。これは IEEE 標準に反しており、無停止 IEEE 例外処理に依存するプログラムの早期終了を引き起こす場合があります。`-ftrap=none` は、C、C++、および F77 ではデフォルトですが、F95 ではデフォルトではありません。

`-fnonstd` `-fns` および `-ftrap=common` のマクロ。

`-xvector` `-xvector=lib` でベクトル数学ライブラリ変換を、`-xvector=simd` で SIMD 変換を有効にするために使用されます。`-xvector=lib` を使用すると、共通の初等超越関数のうちベクトルの向きがわずかに異なる `libmvec` 実装が `libm` バージョンの代わりに使用されるため、数値結果が変わります。これらのベクトルバージョンではデフォルトの丸めが実施されていると仮定します。

`-xreduction` `-xautopar` または `-xopenmp` で並列化が有効になっている場合に、より広い範囲のプログラムループを並列化できるようにします。リダクション演算は、ベクトルの和や 2 つのベクトルの内積などの演算です。和は厳密な演算ではどの順序でも行えますが、有限精度浮動小数点演算で生成される結果がわずかに異なります。実際、累計の順序の判断も不可能な場合があり、同じプログラムと同じデータ、同じハードウェアを使用した場合でも、実行するごとに結果がわずかに変化する可能性があります。

詳細は、各コンパイラのマニュアルページとユーザーズガイドを参照してください。

5.5 再現可能な結果

『数値計算ガイド』のセクション D.11 で説明しているように、標準規格に準拠した IEEE 演算の実装でも結果が異なる場合があります。ほとんどの場合、これらの結果はほぼ等しく適切ですが、同様にほとんどの場合、その証明は冗漫であるか困難になります。ほとんどの目的に対しては、ある程度パフォーマンスを犠牲にして、結果の検証に必要なエラー解析の量を減らすことをお勧めします。出力上のわずかな差異、あるいは大きな差異がどちらも同様に適切になるタイミングと、その相違の原因がユーザープログラムのエラー、コンパイラ最適化エラー、またはハードウェアエラーのどれであるかは明らかではありません。

IEEE 浮動小数点演算の結果が異なる主な根本原因は複数あります。次にこれらの原因を示し、Oracle Solaris Studio のリリースおよびサポートされているプラットフォーム間での無意味なバリエーションを減らすいくつかのアプローチについて説明します。それぞれのアプローチは

再現性を高めますが、パフォーマンスを低下させる可能性があることに注意してください。場合によっては、パフォーマンスの低下が著しくなります。

5.5.1 超越関数

指数関数、対数関数、三角関数など、プログラミング言語で標準化された一般的な数学ライブラリ関数のほとんどは、有理数演算や、平方根 (`sqrt()`) などの代数関数と比べ、正しく丸めるにはコストがかかります。ほぼ正しく丸められる関数は、ほとんどの目的に適しており、非常に高速です。ただし、もっとも高速でほぼ正しく丸められる関数は、プラットフォームによって異なります。

- アプリケーションが使用する関数に応じた移植可能なコードを使用してください。このようなコードのソースの 1 つが、自由に配布可能な数学ライブラリである `fdlibm` です。これは Netlib ソフトウェアリポジトリから入手できます。
- `-xvector` オプションは使用しないでください。超越関数のベクトル化バージョンは、特定のプラットフォームに最適化されており、別のプラットフォームでは生成される結果がわずかに異なります。
- x86 ハードウェア超越命令を使用しないでください。これらの命令でエラーバウンドをできるかぎり小さくしたとしても、完全に正しくは丸められません。また、Intel バージョンと AMD バージョンでは、どちらも非常に適切であったとしても、異なる場合があります。Oracle Solaris Studio C/C++ コンパイラでは、特に `-fast` に続けて `-xbuiltin=%default` を使用して、どの超越命令も、コンパイラによって組み込み超越関数の代わりにインラインが使用されていないことを確認することができます。同様に、`-fast` に続けて `-xnolibmil`オプションを使用すると、インラインテンプレートが無効になります。Oracle Solaris Studio の `libm.il` には、超越命令を呼び出す複数のテンプレートが含まれています。

5.5.2 連想演算

加算および乗算は実数演算で連想型になり、和と積はどの順序でも計算できます。ただし、丸めがある場合は、評価の順序が計算結果に影響します。

- `-xreduction` 並列化オプションを使用しないでください。Oracle Solaris Studio は、予測できない方法でリダクションを最適化します。
- Fortran の `DOT` および `MATMUL` 演算は使用しないでください。Fortran 90 以降のこれらの組み込み関数は、異なるプラットフォームに別々の方法で実装され、その丸め結果も異なります。並列化も有効になっている場合、結果はリダクションの最適化によって予測

できない可能性があります。内積と行列乗算の演算は、Netlib ソフトウェアリポジトリの LAPACK ライブラリで使用可能なものなど、移植可能な Fortran でコード化できます。

5.5.3 不定の評価

ほとんどの言語では、外部式の評価の順序を言語によって指定していません。したがって、`ranf(x)()` が乱数ジェネレータである場合、式 `ranf(x) * a + ranf(x) * b()` は、2 つの `ranf(x)()` の呼び出しの評価順序が変更された場合、異なるコンパイラでは、または同じコンパイルの異なる最適化レベルでは別々の結果を算出します。

2 つの外部参照を含む式を使用しないでください。このような式は、それぞれ含まれる外部参照を 1 つ以下にするように複数のステートメントに分割します。このため、

```
z = ranf(x) * a + ranf(x) * b()
```

は、次のコマンドに置き換えることができます。

```
t = ranf(x) * a()

z = t + ranf(x) * b()
```

5.5.4 移植できない型

C/C++ の `long double` は、SPARC および x86 の Studio では、それぞれ 113 の有効ビットと 64 の有効ビットで実装されます。このため、明示的に `long double` 変数を含むプログラムは、SPARC と x86 では動作が異なることになります。

5.5.5 高い暗黙的精度

一部の状況では、式は、ソースコードに明示されているより高い精度で評価される場合があります。これは、単精度または倍精度変数を含む式の評価に x87 拡張精度レジスタが使用される場合に発生します。また、融合型積和演算が乗算と加算のペアに置き換えられるときにも起こります。

- 乗算と加算のペアは、融合型積和演算として最適化しないでください。`-fast` に続けて `-fma=none` を使用してください。

- -xarch=386 を使用する必要があり、long double 型を明示的に使用していない場合は、すべての変数が float である場合には -fprecision=single を、すべての変数が double である場合には -fprecision=double を使用してコンパイルすることによって、拡張精度式評価の影響を軽減することができます。ただし、Fortran complex*8 変数が -xarch=386 で使用されている場合は、確実にすべての式評価を単精度で行う方法はありません。-m32 より -m64 を使用することをお勧めします。関数の値が、その関数と同じ精度のレジスタで渡されます。

5.6 独立した確認

再現性に関する前述の説明は、再現される結果が正しく、おそらくは多数の正しい結果の 1 つであるという仮定に基づいています。しかし、最終的にはどのようにして確認できるのでしょうか。一部のプログラムには証明がありますが、ほとんどの証明はプログラムよりも複雑です。なぜプログラムよりも信頼できるのでしょうか。プログラムの中には、チェック可能な保存則を持つ物理システムをモデルにしているものがありますが、保存則が完全ではない、または正しくないということが物理的に発見されるとしたらどうしますか。

重要な決定はすべて、独立した手段で確認する必要があります。コンピュータを利用した決定のうち、もっとも重要なケースでは、*独立した手段*で、マシン、命令セット、オペレーティングシステム、プログラムを記述するコンピュータ言語、実装されるアルゴリズムの違い、およびこれを実行する調査者の自然言語、および国の違いに対応する必要があります。どの程度まで対応させるかについては、正しくない結果によってかかるコストに応じて異なります。

このため、たとえば、進数変換をテストするプログラムを記述する場合、最低限、使用するテストアルゴリズムは、テスト対象の進数変換関数で使用する可能性のあるすべてのアルゴリズムとは完全に別のものになるようにする必要があります。このため、コンピュータサイエンスにおいては、低速で単純なアルゴリズムでも、高速で複雑なアルゴリズムをテストできるという点で価値があります。

♦ ♦ ♦ 付 録 A

例

この付録では、一般的なタスクを行う方法の例を示します。例は Fortran または ANSI C で記述されており、その多くは現在のバージョンの `libm` および `libsunmath` に依存しています。これらの例は、Oracle Solaris 10 Update 10 OS 以降のリリースの Oracle Solaris Studio 12.4 でテストされています。C の例は、`-lsunmath -lm` オプションを使用してコンパイルされています。

A.1 IEEE 演算

次の例は、浮動小数点数の 16 進表現を検査できる 1 つの方法を示しています。格納されているデータの 16 進表現を参照するには、デバッガを使用することもできます。

次の C のプログラムは、倍精度の近似値を π と単精度の無限大に出力しています。

例 A-1 倍精度の例

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

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

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

 /* single precision */
```

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

 return 0;
}
```

-lsunmath を指定してコンパイルされた SPARC ベースのシステムでは、前述のプログラムの出力は次のようになります。

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

次の Fortran プログラムは、最小の正規数をそれぞれの形式で出力します。

例 A-2 各形式での最小の正規数の出力 (続き)

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

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

SPARC ベースのシステムでは、対応する出力は次のようになります。

```
min normal, quad:   3.3621031431120935062626778173217526026E-4932
 in hex 00010000000000000000000000000000
min normal, double:  2.2250738585072014-308 in hex 0010000000000000
min normal, single:  1.1754944E-38 in hex 00800000
```

A.2 数学ライブラリ

このセクションでは、数学ライブラリの関数を使用した例を示します。

A.2.1 乱数ジェネレータ

次の例では、数値の配列を生成する乱数ジェネレータを呼び出し、指定した数の EXP を計算するためにかかる時間を測定する関数を使用します。

例 A-3 乱数ジェネレータ

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

 program example
c
 implicit GENERIC (a-h,o-z)
 GENERIC x(SIZE), y, lb, ub
 real tarray(2), u1, u2
c
c compute EXP on random numbers in [-ln2/2,ln2/2]
 lb = -0.3465735903
 ub = 0.3465735903
c
c generate array of random numbers
#ifdef DP
 call d_init_addrans()
 call d_addrans(x,SIZE,lb,ub)
#else
 call r_init_addrans()
 call r_addrans(x,SIZE,lb,ub)
#endif
c
c start the clock
 call dtime(tarray)
 u1 = tarray(1)
c
c compute exponentials
 do 16 i=1,SIZE
  y = exp(x(i))
 16 continue
c
c get the elapsed time
 call dtime(tarray)
```

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

前述の例をコンパイルするには、コンパイラが自動的にプリプロセッサを起動するようにソースコードを接尾辞 F (f ではない) の付いたファイルに保存し、コマンド行で -DSP または -DDP を指定して単精度または倍精度を選択します。

この例では、d_addrans 関数を使用して、ユーザーが指定した範囲に均一に分散するランダムデータのブロックを生成する方法を示します。

例 A-4 d_addrans 関数の使用

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

#include <math.h>
#include <sunmath.h>
#define SIZE 10000
#define LOOPS 100
int main()
{
 double  x[SIZE], y[SIZE];
 int  i, j, n;
 double  lb, ub;
 union {
  unsigned    u[2];
  double    d;
 }  upperbound;

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

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

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

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

```
   ub = upperbound.d;
   lb = 0.0;

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

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

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

A.2.2 IEEE が推奨する関数

次の Fortran の例では、IEEE 標準規格が推奨するいくつかの関数を使用しています。

例 A-5 IEEE が推奨する関数

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

c Refer to
c ieee_functions(3m)
c libm_double(3f)
c libm_single(3f)
c
c The 5 functions demonstrated here are:
c
c ilogb(x): returns the base 2 unbiased exponent of x in
c   integer format
c signbit(x): returns the sign bit, 0 or 1
c copysign(x,y): returns x with y's sign bit
```

```
c nextafter(x,y): next representable number after x, in
c   the direction y
c scalbn(x,n): x * 2**n
c
c function    double precision      single precision
c ---------------------------------------------------
c ilogb(x)     i = id_ilogb(x)      i = ir_ilogb(r)
c signbit(x)    i = id_signbit(x)      i = ir_signbit(r)
c copysign(x,y)    x = d_copysign(x,y)      r = r_copysign(r,s)
c nextafter(x,y)    z = d_nextafter(x,y)      r = r_nextafter(r,s)
c scalbn(x,n)    x = d_scalbn(x,n)      r = r_scalbn(r,n)
 program ieee_functions_demo
 implicit double precision (d)
 implicit real (r)
 double precision   x, y, z, direction
 real               r, s, t, r_direction
 integer            i, scale

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

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

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


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

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

 x = d_min_subnormal()
 direction = 1.0d0
    y = d_nextafter(x, direction)
 write(*,3) x
 write(*,4) direction, y
 x = 2.0d0
```

```
scale = 3
y = d_scalbn(x, scale)
write (*,5) x, scale, y
5 format(' Scaling ', F4.1, ' by 2**', I1, ' is ', F4.1)
print *
print *, 'SINGLE PRECISION EXAMPLES:'
print *

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

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

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

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

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

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

print *
end
```

このプログラムからの出力は次の例のようになります。

例 A-6 例 A-5 の出力

```
DOUBLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
-5.5 was given the sign of 12.4 and is now  5.5
The sign bit of    -5.5 is   1
Starting from  4.9406564584124654E-324, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  4.9406564584124654E-324, the next representable
```

```
   number towards  1.0 is  9.8813129168249309E-324
Scaling  2.0 by 2**3 is 16.0

SINGLE PRECISION EXAMPLES:

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

`-f77` 互換性オプションを指定して `f95` コンパイラを使用した場合は、次の追加のメッセージが表示されます。

```
 Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

A.2.3 IEEE の特殊な値

次の C プログラムでは、いくつかの *ieee_values*(3m) の関数を呼び出します。

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

int main()
{
 double       x;
 float       r;

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

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

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

 return 0;
```

```
}
```

リンクするときには、-lsunmath および -lm の両方を指定してください。

SPARC ベースシステムでは、出力は次のようになります。

```
quiet NaN: NaN = 7ff80000 00000000
nextafter(max_subnormal,0) = 2.2250738585072004e-308
 = 000fffff fffffffe
single precision min subnormal = 1.40129846e-45 = 00000001
```

x86 アーキテクチャーは「リトルエンディアン」であるため、x86 での出力は若干異なり、倍精度数の 16 進表現で上位と下位のワードの順序が逆になります。

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

ieee_values 関数を使用する Fortran プログラムでは、このような関数の型を宣言する必要があります。

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

SPARC では、出力は次のようになります。

```
6.475175119438025110924438958227646E-4966
Inf
3.4028235E+38
```

A.2.4 ieee_flags — 丸め方向

次の例は、丸めモードを*ゼロへの丸め*に設定する方法を示しています。

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

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

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

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

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

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

 return 0;
}
```

上記の丸め方向の短いプログラムの次の出力は、SPARC でのゼロへの丸めの効果を示しています。

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

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

上記の丸め方向の短いプログラムの次の出力は、x86 でのゼロへの丸めの効果を示しています。

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

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

Fortran プログラムでゼロへの丸め方向を設定するには、次の例を使用します。

```
program ieee_flags_demo
character*16 out

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

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

end
```

出力は次のようになります。

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

`-f77` 互換性オプションを指定して `f95` コンパイラを使用してプログラムをコンパイルした場合は、次の追加のメッセージが出力に表示されます。

```
demo% f95 ieee_flags_demo.f -f77
demo% a.out
 Note: Rounding direction toward zero
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

A.2.5 C99 浮動小数点環境関数

次の例では、いくつかの C99 浮動小数点環境関数を使用しています。`norm` 関数は、アンダーフローおよびオーバーフローを処理するため、ベクトルのユークリッドノルムを計算し、この環境関数を使用します。遡及診断出力が示すように、メインプログラムは、アンダーフローとオーバーフローを発生させるように位取りされたベクトルを使用してこの関数を呼び出します。

例 A-7 C99 浮動小数点環境関数

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

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

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

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

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

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

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

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

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

    fex_set_log(stdout);

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

    return 0;
}
```

SPARC ベースシステムでは、このプログラムをコンパイルして実行すると、次のように出力されます。

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

次のコード例は、x86 ベースシステムでの `fesetprec` 関数の効果を示しています。この関数は SPARC ベースシステムでは使用できません。`while` ループでは、1 に加えられるときに完全に丸められる 2 の最大の累乗を見つけることによって、使用できる精度を決定しようとしています。最初のループが示すように、すべての中間結果を拡張倍精度で評価する x86 ベースシステムのようなアーキテクチャーでは、この手法が予期したとおりに動作しない場合があります。このため、2 番目のループが示しているように、`fesetprec` 関数を使用すると、すべての結果を必要な精度に丸めることができます。

例 A-8 `fesetprec` 関数 (x86)

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

int main()
{
    double  x;

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

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

    return 0;
}
```

`cc A8.c -lm -xarch=386` を指定して x86 システムでコンパイルすると、次のように出力されます。

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

最後に、次の例は、マルチスレッドプログラムで環境関数を使用して、親スレッドから子スレッドに浮動小数点モードを伝播し、子スレッドが親スレッドに結合されるときに子スレッド内で発生した例外フラグを回復する 1 つの方法を示しています。マルチスレッドプログラムの記述については、『Multithreaded Programming Guide 』を参照してください。

例 A-9 マルチスレッドプログラムでの環境関数の使用

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

fenv_t  env;

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

void parent()
```

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

A.3 例外と例外処理

A.3.1 `ieee_flags` — 累積例外

通常、累積例外ビットの*検査*または*クリア*はユーザープログラムで行います。次の例は、累積例外フラグを検査する C プログラムです。

例 A-10 累積例外フラグの検査

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

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

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

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

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

 /* decode the return value */
 inexact =      (code >> fp_inexact)     & 0x1;
```

```
 underflow =    (code >> fp_underflow)   & 0x1;
 division =     (code >> fp_division)    & 0x1;
 overflow =     (code >> fp_overflow)    & 0x1;
 invalid =      (code >> fp_invalid)     & 0x1;

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

上記のプログラムを実行した場合の出力は次のようになります。

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

これは Fortran でも同様に行うことができます。

例 A-11 累積例外フラグの検査 – Fortran

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

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

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

c Decode value returned by ieee_flags using bit-shift intrinsics
```

```
    inx   = and(rshift(accrued, fp_inexact)  , 1)
    under = and(rshift(accrued, fp_underflow), 1)
    div   = and(rshift(accrued, fp_division) , 1)
    over  = and(rshift(accrued, fp_overflow) , 1)
    inv   = and(rshift(accrued, fp_invalid)  , 1)

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

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

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

出力は次のようになります。

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

通常、ユーザープログラムでは例外フラグを*設定*しませんが、設定は可能です。次の C の例でこれを示します。

```
#include <sunmath.h>

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

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

 return 0;
}
```

SPARC では、上記のプログラムの出力は次のようになります。

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

x86 では、出力は次のようになります。

```
out is: division , fp exception code is: 4
```

A.3.2 ieee_handler: 例外のトラップ

注記 - 次の例は、Oracle Solaris OS にのみ適用されます。

次の例は、例外を特定するためのシグナルハンドラをインストールする Fortran プログラムです (SPARC ベースシステムの場合のみ)。

例 A-12 アンダーフローでのトラップ (SPARC)

```
  program demo
c declare signal handler function
 external fp_exc_hdl
 double precision d_min_normal
 double precision x
c set up signal handler
 i = ieee_handler('set', 'common', fp_exc_hdl)
 if (i.ne.0) print *, 'ieee trapping not supported here'
c cause an underflow exception (it will not be trapped)
 x = d_min_normal() / 13.0
 print *, 'd_min_normal() / 13.0 = ', x
c cause an overflow exception
c the value printed out is unrelated to the result
 x = 1.0d300
 x = x * x
 print *, '1.0d300*1.0d300 = ', x
 end
c
c the floating-point exception handling function
c
 integer function fp_exc_hdl(sig, sip, uap)
 integer sig, code, addr
 character label*16
c
c The structure /siginfo/ is a translation of siginfo_t
c from <sys/siginfo.h>
c
 structure /fault/
 integer address
 end structure
 structure /siginfo/
 integer si_signo
 integer si_code
 integer si_errno
 record /fault/ fault
 end structure

 record /siginfo/ sip
c See <sys/machsig.h> for list of FPE codes
c Figure out the name of the SIGFPE
```

```
 code = sip.si_code
 if (code.eq.3) label = 'division'
 if (code.eq.4) label = 'overflow'
 if (code.eq.5) label = 'underflow'
 if (code.eq.6) label = 'inexact'
 if (code.eq.7) label = 'invalid'
 addr = sip.fault.address
c Print information about the signal that happened
 write (*,77) code, label, addr
 77  format ('floating-point exception code ', i2, ',',
     *  a17, ',', ' at address ', z8 )
 end
```

前のコードを -f77 を指定してコンパイルすると、出力は次のようになります。

```
 d_min_normal() / 13.0 =     1.7115952757748-309
floating-point exception code  4, overflow        , at address    1131C
 1.0d300*1.0d300 =     1.0000000000000+300
 Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
 ieee_handler(3M)
```

次の例は、SPARC ベースシステムでのより複雑な C の例です。

例 A-13 無効、ゼロ除算、オーバーフロー、アンダーフロー、および不正確でのトラップ (SPARC)

```
/*
 * Generate the 5 IEEE exceptions: invalid, division,
 * overflow, underflow and inexact.
 *
 * Trap on any floating point exception, print a message,
* and continue.*
 * Note that you could also inquire about raised exceptions by
*    i = ieee("get","exception","",&out);* where out contains the name of the highest
 exception
 * raised, and i can be decoded to find out about all the
 * exceptions raised.
 */

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

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

int main()
{
 double  x, y, z;
```

```
char  *out;

/*
 * Use ieee_handler to establish "trap_all_fp_exc"
 * as the signal handler to use whenever any floating
 * point exception occurs.
 */

if (ieee_handler("set", "all", trap_all_fp_exc) != 0)
 printf(" IEEE trapping not supported here.\n");
/* disable trapping (uninteresting) inexact exceptions */
if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
  printf("Trap handler for inexact not cleared.\n");
/* raise invalid */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("1. Invalid: signaling_nan(0) * 2.5\n");
x = signaling_nan(0);
y = 2.5;
z = x * y;

/* raise division */
if (ieee_flags("clear", "exception", "all", &out) != 0)
 printf(" could not clear exceptions\n");
printf("2. Div0: 1.0 / 0.0\n");
x = 1.0;
y = 0.0;
z = x / y;

/* raise overflow */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("3. Overflow: -max_normal() - 1.0e294\n");
x = -max_normal();
y = -1.0e294;
z = x + y;

/* raise underflow */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("4. Underflow: min_normal() * min_normal()\n");
x = min_normal();
y = x;
z = x * y;

/* enable trapping on inexact exception */
if (ieee_handler("set", "inexact", trap_all_fp_exc) != 0)
  printf("Could not set trap handler for inexact.\n");
/* raise inexact */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("5. Inexact: 2.0 / 3.0\n");
x = 2.0;
y = 3.0;
```

```
 z = x / y;
 /* don't trap on inexact */
 if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
   printf(" could not reset inexact trap\n");

 /* check that we're not trapping on inexact anymore */
 if (ieee_flags("clear", "exception", "all", &out) != 0)
   printf(" could not clear exceptions\n");
 printf("6. Inexact trapping disabled; 2.0 / 3.0\n");
 x = 2.0;
 y = 3.0;
 z = x / y;

 /* find out if there are any outstanding exceptions */
 ieee_retrospective_();

 /* exit gracefully */
 return 0;
}

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

/* see /usr/include/sys/machsig.h for SIGFPE codes */
 switch (sip->si_code) {
 case FPE_FLTRES:
   label = "inexact";
   break;
 case FPE_FLTDIV:
   label = "division";
   break;
 case FPE_FLTUND:
   label = "underflow";
   break;
 case FPE_FLTINV:
   label = "invalid";
   break;
 case FPE_FLTOVF:
   label = "overflow";
   break;
 }

 printf(
 " signal %d, sigfpe code %d: %s exception at address %x\n",
   sig, sip->si_code, label, sip->__data.__fault.__addr);
}
```

出力は、次のようになります。

```
1. Invalid: signaling_nan(0) * 2.5
   signal 8, sigfpe code 7: invalid exception at address 10da8
2. Div0: 1.0 / 0.0
   signal 8, sigfpe code 3: division exception at address 10e44
3. Overflow: -max_normal() - 1.0e294
```

```
   signal 8, sigfpe code 4: overflow exception at address 10ee8
4. Underflow: min_normal() * min_normal()
   signal 8, sigfpe code 5: underflow exception at address 10f80
5. Inexact: 2.0 / 3.0
   signal 8, sigfpe code 6: inexact exception at address 1106c
6. Inexact trapping disabled; 2.0 / 3.0
Note: IEEE floating-point exception traps enabled:
   underflow; overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_handler(3M)
```

次のコードは、SPARC で `ieee_handler` およびインクルードファイルを使用して、特定の例外状況のデフォルトの結果を変更する方法を示しています。

例 A-14 例外状況のデフォルトの結果の変更

```
/*
 * Cause a division by zero exception and use the
 * signal handler to substitute MAXDOUBLE (or MAXFLOAT)
 * as the result.
 *
 * compile with the flag -Xa
 */

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

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

int main() {
 double     x, y, z;
 float     r, s, t;
 char     *out;

 /*
  * Use ieee_handler to establish division_handler as the
  * signal handler to use for the IEEE exception division.
  */
 if (ieee_handler("set","division",division_handler)!=0) {
  printf(" IEEE trapping not supported here.\n");
 }

 /* Cause a division-by-zero exception */
 x = 1.0;
 y = 0.0;
 z = x / y;

 /*
  * Check to see that the user-supplied value, MAXDOUBLE,
  * is indeed substituted in place of the IEEE default
  * value, infinity.
  */
```

```
 printf("double precision division: %g/%g = %g \n",x,y,z);

 /* Cause a division-by-zero exception */
 r = 1.0;
 s = 0.0;
 t = r / s;

 /*
  * Check to see that the user-supplied value, MAXFLOAT,
  * is indeed substituted in place of the IEEE default
  * value, infinity.
  */
 printf("single precision division: %g/%g = %g \n",r,s,t);

 ieee_retrospective_();

 return 0;
}

void division_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
 int       inst;
 unsigned       rd, mask, single_prec=0;
 float       f_val = MAXFLOAT;
 double       d_val = MAXDOUBLE;
 long       *f_val_p = (long *) &f_val;

 /* Get instruction that caused exception. */
 inst = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

 /*
  * Decode the destination register. Bits 29:25 encode the
  * destination register for any SPARC floating point
  * instruction.
  */
 mask = 0x1f;
 rd = (mask & (inst >> 25));

 /*
  * Is this a single precision or double precision
  * instruction?  Bits 5:6 encode the precision of the
  * opcode; if bit 5 is 1, it's sp, else, dp.
  */

 mask = 0x1;
 single_prec = (mask & (inst >> 5));

 /* put user-defined value into destination register */
 if (single_prec) {
  uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[rd] =
   f_val_p[0];
 } else {
 uap->uc_mcontext.fpregs.fpu_fr.fpu_dregs[rd/2] = d_val;
 }
}
```

次の出力が予測されます。

```
double precision division: 1/0 = 1.79769e+308
single precision division: 1/0 = 3.40282e+38
Note: IEEE floating-point exception traps enabled:
   division by zero;
See the Numerical Computation Guide, ieee_handler(3M)
```

A.3.3 ieee_handler: 例外での中止

特定の浮動小数点例外の場合は、ieee_handler を使用して強制的にプログラムを中止させることができます。

```
#include <floatingpoint.h>
 program abort
c
 ieeer = ieee_handler('set', 'division', SIGFPE_ABORT)
 if (ieeer .ne. 0) print *, ' ieee trapping not supported'
 r = 14.2
 s = 0.0
 r = r/s
c
 print *, 'you should not see this; system should abort'
c
 end
```

A.3.4 libm の例外処理機能

次の例は、libm によって提供される例外処理機能のいくつかを使用する方法を示しています。最初の例は、数値 x と係数 a_0、a_1、...、a_N、および b_0、b_1、...、b_{N-1} の場合に、関数 f(x) とその一次導関数 f'(x) を評価するタスクに基づいています。ここで f() は次の連分数です。

$$f(x) = a_0 + b_0 / (x + a_1 + b_1 / (x + ... / (x + a_{N-1} + b_{N-1} / (x + a_N))...))$$

IEEE 演算では、f() の計算は単純です。中間除算の 1 つがオーバーフローしたり、ゼロ除算を行う場合でも、標準規格によって指定されているデフォルトの値 (正しい符号が付けられた無限大) から正しい結果が算出されます。一方、f'() の計算は、これを評価するもっとも簡潔な形式が、削除可能な特異点を持つことができるため、f の計算よりも難しくなります。計算中にこれらの特異点の 1 つが出現すると、不定形 0/0、0*無限大、または無限大/無限大のいずれか 1 つの評価が試みられます (これらはすべて無効な演算例外を発生します)。W. Kahan は、*前置換*という機能によってこれらの例外を処理する方法を提唱しています。

前置換は、例外に対する IEEE のデフォルトの応答を拡張したものであり、例外演算の結果を置換する値をユーザーが前もって指定できます。libm の例外処理機能を使用すると、FEX_CUSTOM 例外処理モードでハンドラをインストールすることによって、プログラムで簡単に前置換を実装できます。このモードでは、ハンドラに渡された info パラメータが差しているデータ構造体に値を格納するだけで、ハンドラは例外演算の結果に任意の値を設定できます。次の例は、FEX_CUSTOM ハンドラによって実装した前置換を使用して連分数とその導関数を計算するプログラムの例を示しています。

例 A-15 FEX_CUSTOM ハンドラを使用した連分数とその導関数の計算

```
#include <stdio.h>
#include <sunmath.h>
#include <fenv.h>
volatile double p;
void handler(int ex, fex_info_t *info)
{
    info->res.type = fex_double;
    if (ex == FEX_INV_ZMI)
        info->res.val.d = p;
    else
        info->res.val.d = infinity();
}

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double x, double *pf, double *pf1)
{
    fex_handler_t    oldhdl; /* for saving/restoring handlers */
    volatile double  t;
    double           f, f1, d, d1, q;
    int              j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    fex_set_handling(FEX_DIVBYZERO, FEX_NONSTOP, NULL);
    fex_set_handling(FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI,
        FEX_CUSTOM, handler);

    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        /* the following assignment to the volatile variable t
           is needed to maintain the correct sequencing between
```

```
           assignments to p and evaluation of f1 */
        t = f1 = (-d1 / d) * q;
        p = b[j-1] * d1 / b[j];
        f = a[j] + q;
    }

    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}

/* For the following coefficients, x = -3, 1, 4, and 5 will all
   encounter intermediate exceptions */
double a[] = { -1.0, 2.0, -3.0, 4.0, -5.0 };
double b[] = { 2.0, 4.0, 6.0, 8.0 };

int main()
{
    double  x, f, f1;
    int     i;

    feraiseexcept(FE_INEXACT); /* prevent logging of inexact */
    fex_set_log(stdout);
    fex_set_handling(FEX_COMMON, FEX_ABORT, NULL);
    for (i = -5; i <= 5; i++) {
        x = i;
        continued_fraction(4, a, b, x, &f, &f1);
        printf("f(% g) = %12g, f'(% g) = %12g\n", x, f, x, f1);
    }
    return 0;
}
```

このプログラムについては、順を追って解説します。入口で、関数 `continued_fraction` は、ゼロ除算およびすべての無効な演算例外に対する現在の例外処理モードを保存します。続いて、ゼロ除算に対して無停止の例外処理、および 3 つの不定形に対して `FEX_CUSTOM` ハンドラを設定します。このハンドラは 0/0 と無限大/無限大の両方を無限大で置換しますが、0*無限大は大域変数 `p` の値で置換します。後続の 0*無限大の無効な演算の置換には、正しい値を設定できるように `p` は関数を評価するループで毎回再計算する必要があります。また、`p` はループ内で明示的に記述されていないため、コンパイラによって削除されないように `volatile` と宣言する必要があります。最後に、コンパイラが `p` に対する代入を、例外を発生させる可能性のある計算 (これに対して `p` が前置換の値を提供します) の上または下に移動させないように、その計算の結果も `volatile` 変数 (このプログラムでは `t`) に代入しています。`fex_setexcepthandler` の最後の呼び出しによって、ゼロ除算および無効な演算に対する元の処理モードが復元されます。

メインプログラムは、fex_set_log 関数を呼び出して、遡及診断のロギングを有効にしています。この呼び出しの前に、メインプログラムは不正確フラグを発生させ、これによって不正確演算がロギングされなくなります。91 ページの「遡及診断」セクションで説明しているように、FEX_NONSTOP モードでは例外フラグが発生しても例外がログに記録されません。また、メインプログラムは、一般的な例外に対して FEX_ABORT モードを設定し、continued_fraction によって明示的に処理されない特異な例外によってプログラムが終了しないようにしています。最後に、プログラムは複数のポイントで特定の連分数を評価しています。次の出力例が示しているように、計算で実際に中間例外が発生しています。

```
f(-5) =     -1.59649,   f'(-5) =      -0.1818
f(-4) =     -1.87302,   f'(-4) =    -0.428193
Floating point division by zero at 0x08048dbe continued_fraction, nonstop mode
  0x08048dc1  continued_fraction
  0x08048eda  main
Floating point invalid operation (inf/inf) at 0x08048dcf continued_fraction, handler: handler
  0x08048dd2  continued_fraction
  0x08048eda  main
Floating point invalid operation (0*inf) at 0x08048dd2 continued_fraction, handler: handler
  0x08048dd8  continued_fraction
  0x08048eda  main
f(-3) =           -3,   f'(-3) =     -3.16667
f(-2) = -4.44089e-16,   f'(-2) =     -3.41667
f(-1) =     -1.22222,   f'(-1) =    -0.444444
f( 0) =     -1.33333,   f'( 0) =     0.203704
f( 1) =           -1,   f'( 1) =     0.333333
f( 2) =    -0.777778,   f'( 2) =      0.12037
f( 3) =    -0.714286,   f'( 3) =    0.0272109
f( 4) =    -0.666667,   f'( 4) =     0.203704
f( 5) =    -0.777778,   f'( 5) =    0.0185185
```

x = 1、4、および 5 の場合の f'(x) の計算で発生する例外は、プログラム内で x = -3 のときに起きる例外と同じサイトで発生するため、遡及診断メッセージに出力されません。

上記のプログラムは、連分数とその導関数の評価で発生する例外を処理する場合の、もっとも効率がよい方法ではない可能性があります。その理由の 1 つは、必要であるかどうかにかかわらず、ループが繰り返されるごとに前置換の値を再計算する必要があることです。この場合、前置換の値の計算で浮動小数点除算が行われますが、最新の SPARC および x86 プロセッサでは浮動小数点除算は比較的遅い演算です。そのうえ、ループ自体にすでに 2 つの除算が含まれていますが、ほとんどの SPARC および x86 プロセッサは 2 つの除算演算の実行をオーバーラップできないため、除算がループ内のボトルネックとなりやすくなります。別の除算を追加するとボトルネックはさらに悪化します。

1 つの除算のみを必要とするようにループを記述しなおすことができます。実際、前置換値の計算では除算を必要としません。このようにループを記述しなおすには、b 配列内の係数の隣接要素比を前もって計算する必要があります。これにより、複数の除算の演算でボトルネックは排

除されますが、前置換値の計算に関連するすべての算術演算が除外されるわけではありません。さらに、前置換値と演算結果の両方が volatile 変数に前置換されるように割り当てる必要があるため、プログラムの速度を低下させるメモリー演算が増えます。これらの代入は、コンパイラが特定のキー操作を並べ替えないようにするために必要ですが、コンパイラがほかの無関係な演算を並べ替えることも防止することになります。このため、この例のように前置換を使用して例外を処理すると、メモリー演算が増え、通常では可能な最適化が行われなくなります。これらの例外を、さらに効率よく処理することは可能でしょうか。

高速な前置換のための特別なハードウェアがサポートされていない場合、この例の例外をもっとも効率よく処理する方法は、次のバージョンに示すようにフラグを使用することです。

例 A-16 例外処理へのフラグの使用 (続き)

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

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double         x, double *pf, double *pf1)
{
    fex_handler_t  oldhdl;
    fexcept_t      oldinvflag;
    double         f, f1, d, d1, pd1, q;
    int            j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);
    fegetexceptflag(&oldinvflag, FE_INVALID);

    fex_set_handling(FEX_DIVBYZERO | FEX_INV_ZDZ | FEX_INV_IDI |
        FEX_INV_ZMI, FEX_NONSTOP, NULL);
    feclearexcept(FE_INVALID);

    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        f1 = (-d1 / d) * q;
        f = a[j] + q;
    }

    if (fetestexcept(FE_INVALID)) {
        /* recompute and test for NaN */
```

```
        f1 = pd1 = 0.0;
        f = a[N];
        for (j = N - 1; j >= 0; j--) {
            d = x + f;
            d1 = 1.0 + f1;
            q = b[j] / d;
            f1 = (-d1 / d) * q;
            if (isnan(f1))
                f1 = b[j] * pd1 / b[j+1];
            pd1 = d1;
            f = a[j] + q;
        }
    }

    fesetexceptflag(&oldinvflag, FE_INVALID);
    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}
```

このバージョンでは、最初のループはデフォルトの無停止モードで `f(x)` および `f'(x)` の計算を試みています。無効フラグが発生すると、2 番目のループは NaN の形態をテストして `f(x)` および `f'(x)` を明示的に再計算します。通常、無効な演算例外は発生しないため、プログラムは最初のループのみを実行します。このループには、`volatile` 変数に対する参照も余分な算術演算も含まれていないため、コンパイラが許すかぎりの速度で実行されます。この効率を得るためには、例外が発生した場合の処理について、2 番目のループを最初のループとほとんど同じように記述する必要があります。このトレードオフは、浮動小数点例外処理において生じることがある典型的なジレンマです。

A.3.5 Fortran プログラムでの `libm` 例外処理の使用

`libm` の例外処理機能は主に C/C++ プログラムからの使用を想定していますが、Sun の Fortran 言語の相互運用性機能を使用すると、Fortran プログラムからも一部の `libm` 関数を呼び出すことができます。

注記 - 一貫した動作を保つため、`libm` 例外処理関数と `ieee_flags` および `ieee_handler` 関数の両方を同じプログラム内で使用しないでください。

次の例では、前置換を使用して連分数とその導関数を評価するための Fortran バージョンのプログラムを示します (SPARC のみ)。

例 A-17 前置換を使用した連分数とその導関数の評価 (SPARC)

```
c
c Presubstitution handler
c
 subroutine handler(ex, info)
 structure /fex_numeric_t/
 integer type
 union
 map
 integer i
 end map
 map
 integer*8 l
 end map
 map
 real f
 end map
 map
 real*8 d
 end map
 map
 real*16 q
 end map
 end union
 end structure

 structure /fex_info_t/
 integer op, flags
 record /fex_numeric_t/ op1, op2, res
 end structure
 integer ex
 record /fex_info_t/ info
 common /presub/ p
 double precision p, d_infinity
 volatile p
c 4 = fex_double; see <fenv.h> for this and other constants
 info.res.type = 4
c x'80' = FEX_INV_ZMI
 if (loc(ex) .eq. x'80') then
 info.res.d = p
 else
 info.res.d = d_infinity()
 endif
 return
 end
c
c Evaluate the continued fraction given by coefficients a(j) and
c b(j) at the point x; return the function value in f and the
c derivative in f1
c
 subroutine continued_fraction(n, a, b, x, f, f1)
 integer n
 double precision a(*), b(*), x, f, f1
```

```
 common /presub/ p
 integer j, oldhdl
 dimension oldhdl(24)
 double precision d, d1, q, p, t
 volatile p, t
 data ixff2/x'ff2'/
 data ix2/x'2'/
 data ixb0/x'b0'/

 external fex_getexcepthandler, fex_setexcepthandler
 external fex_set_handling, handler
c$pragma c(fex_getexcepthandler, fex_setexcepthandler)
c$pragma c(fex_set_handling)
c x'ff2' = FEX_DIVBYZERO | FEX_INVALID
 call fex_getexcepthandler(oldhdl, %val(ixff2))
c x'2' = FEX_DIVBYZERO, 0 = FEX_NONSTOP
 call fex_set_handling(%val(ix2), %val(0), %val(0))
c x'b0' = FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI, 3 = FEX_CUSTOM
 call fex_set_handling(%val(ixb0), %val(3), handler)
 f1 = 0.0d0
 f = a(n+1)
 do j = n, 1, -1
 d = x + f
 d1 = 1.0d0 + f1
 q = b(j) / d
 f1 = (-d1 / d) * q
c
c the following assignment to the volatile variable t
c is needed to maintain the correct sequencing between
c assignments to p and evaluation of f1
 t = f1
 p = b(j-1) * d1 / b(j)
 f = a(j) + q
 end do
 call fex_setexcepthandler(oldhdl, %val(ixff2))
 return
 end

c Main program
c
 program cf
 integer i
 double precision a, b, x, f, f1
 dimension a(5), b(4)
 data a /-1.0d0, 2.0d0, -3.0d0, 4.0d0, -5.0d0/
 data b /2.0d0, 4.0d0, 6.0d0, 8.0d0/
 data ixffa/x'ffa'/

 external fex_set_handling
c$pragma c(fex_set_handling)
c x'ffa' = FEX_COMMON, 1 = FEX_ABORT
 call fex_set_handling(%val(ixffa), %val(1), %val(0))
 do i = -5, 5
 x = dble(i)
```

```
 call continued_fraction(4, a, b, x, f, f1)
 write (*, 1) i, f, i, f1
 end do
 1  format('f(', I2, ') = ', G12.6, ', f''(', I2, ') = ', G12.6)
 end
```

-f77 フラグを指定してコンパイルされたこのプログラムの出力は次のようになります。

```
f(-5) = -1.59649    , f'(-5) = -.181800
f(-4) = -1.87302    , f'(-4) = -.428193
f(-3) = -3.00000    , f'(-3) = -3.16667
f(-2) = -.444089E-15, f'(-2) = -3.41667
f(-1) = -1.22222    , f'(-1) = -.444444
f( 0) = -1.33333    , f'( 0) = 0.203704
f( 1) = -1.00000    , f'( 1) = 0.333333
f( 2) = -.777778    , f'( 2) = 0.120370
f( 3) = -.714286    , f'( 3) = 0.272109E-01
f( 4) = -.666667    , f'( 4) = 0.203704
f( 5) = -.777778    , f'( 5) = 0.185185E-01
 Note: IEEE floating-point exception flags raised:
    Inexact;  Division by Zero;  Underflow; Invalid Operation;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M)
```

A.4 その他

A.4.1 sigfpe: 整数例外のトラップ

前のセクションでは、ieee_handler を使用した例を示しました。通常、ieee_handler または sigfpe のどちらを使用するかを選択する場合は、前者をお勧めします。

注記 - sigfpe を使用できるのは Oracle Solaris OS のみです。

整数演算例外をトラップする場合などに、ハンドラとして sigfpe を使用することがあります。例 A-18「整数例外のトラップ」では、SPARC ベースのシステムで整数のゼロ除算をトラップしています。

例 A-18 整数例外のトラップ

```
/* Generate the integer division by zero exception */
#include <signal.h>
```

```
#include <siginfo.h>
#include <ucontext.h>
void int_handler(int sig, siginfo_t *sip, ucontext_t *uap);
int main() {
int a, b, c;
/*
* Use sigfpe(3) to establish "int_handler" as the signal handler
* to use on integer division by zero
*/
/*
* Integer division-by-zero aborts unless a signal
* handler for integer division by zero is set up
*/
sigfpe(FPE_INTDIV, int_handler);

a = 4;
b = 0;
c = a / b;
printf("%d / %d = %d\n\n", a, b, c);
return 0;
}
void int_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
printf("Signal %d, code %d, at addr %x\n",
sig, sip->si_code, sip->__data.__fault.__addr);
/*
* automatically for floating-point exceptions but not for
* integer division by zero.
*/
uap->uc_mcontext.gregs[REG_PC] =
uap->uc_mcontext.gregs[REG_nPC];
}
```

A.4.2 C からの Fortran の呼び出し

次に、Fortran のサブルーチンを呼び出す C ドライバの簡単な例を示します。C および Fortran での動作の詳細は、『Oracle Solaris Studio 12.4: C ユーザーガイド 』および『Oracle Solaris Studio 12.4: Fortran ユーザーズガイド 』を参照してください。C ドライバを次に示します (`driver.c` というファイルに保存します)。

例 A-19 C からの Fortran の呼び出し

```
/*
 * a demo program that shows:
 * 1. how to call f95 subroutine from C, passing an array argument
 * 2. how to call single precision f95 function from C
 * 3. how to call double precision f95 function from C
 */

extern int      demo_one_(double *);
```

```
extern float    demo_two_(float *);
extern double   demo_three_(double *);

int main()
{
 double array[3][4];
 float f, g;
 double x, y;
 int i, j;

 for (i = 0; i < 3; i++)
  for (j = 0; j < 4; j++)
   array[i][j] = i + 2*j;

 g = 1.5;
 y = g;

 /* pass an array to a fortran function (print the array) */
 demo_one_(&array[0][0]);
 printf(" from the driver\n");
 for (i = 0; i < 3; i++) {
  for (j = 0; j < 4; j++)
   printf("    array[%d][%d] = %e\n",
    i, j, array[i][j]);
  printf("\n");
 }

 /* call a single precision fortran function */
 f = demo_two_(&g);
 printf(
        " f = sin(g) from a single precision fortran function\n");
 printf("    f, g: %8.7e, %8.7e\n", f, g);
 printf("\n");

 /* call a double precision fortran function */
 x = demo_three_(&y);
 printf(
  " x = sin(y) from a double precision fortran function\n");
 printf("    x, y: %18.17e, %18.17e\n", x, y);

 ieee_retrospective_();
 return 0;
}
```

drivee.f というファイルに Fortran のサブルーチンを保存します。

```
subroutine demo_one(array)
double precision array(4,3)
print *, 'from the fortran routine:'
do 10 i =1,4
 do 20 j = 1,3
  print *, '   array[', i, '][', j, '] = ', array(i,j)
20 continue
print *
```

```
10 continue
return
end

real function demo_two(number)
real number
demo_two = sin(number)
return
end

double precision function demo_three(number)
double precision number
demo_three = sin(number)
return
end
```

コンパイルとリンクを行います。

```
cc -c driver.c
f95 -c drivee.f
 demo_one:
 demo_two:
 demo_three:
f95 -o driver driver.o drivee.o
```

出力は次のようになります。

```
 from the fortran routine:
    array[ 1 ][ 1 ] =  0.0E+0
    array[ 1 ][ 2 ] =  1.0
    array[ 1 ][ 3 ] =  2.0

    array[ 2 ][ 1 ] =  2.0
    array[ 2 ][ 2 ] =  3.0
    array[ 2 ][ 3 ] =  4.0

    array[ 3 ][ 1 ] =  4.0
    array[ 3 ][ 2 ] =  5.0
    array[ 3 ][ 3 ] =  6.0

    array[ 4 ][ 1 ] =  6.0
    array[ 4 ][ 2 ] =  7.0
    array[ 4 ][ 3 ] =  8.0

 from the driver
    array[0][0] = 0.000000e+00
    array[0][1] = 2.000000e+00
    array[0][2] = 4.000000e+00
    array[0][3] = 6.000000e+00

    array[1][0] = 1.000000e+00
    array[1][1] = 3.000000e+00
    array[1][2] = 5.000000e+00
    array[1][3] = 7.000000e+00
```

```
   array[2][0] = 2.000000e+00
   array[2][1] = 4.000000e+00
   array[2][2] = 6.000000e+00
   array[2][3] = 8.000000e+00

f = sin(g) from a single precision fortran function
   f, g: 9.9749500e-01, 1.5000000e+00

x = sin(y) from a double precision fortran function
   x, y: 9.97494986604054446e-01, 1.50000000000000000e+00
```

A.4.3 役に立つデバッグコマンド

次の表は、SPARC アーキテクチャーのデバッグコマンドの例を示しています。

表 A-1 いくつかのデバッグコマンド (SPARC)

処理	dbx	adb
関数へのブレークポイントの設定	`stop in myfunct`	`myfunct:b`
行番号へのブレークポイントの設定	`stop at 29`	n/a
絶対アドレスへのブレークポイントの設定	n/a	`23a8:b`
相対アドレスへのブレークポイントの設定	n/a	`main+0x40:b`
ブレークポイントに達するまで実行	`run`	`:r`
ソースコードの表示	`list`	`<pc,10?ia`
fp レジスタの表示: IEEE 単精度	`print $f0`	`<f0=X`
fp レジスタの表示: 等価の 10 進数 (Hex)	`print -fx $f0`	`<f0=f`
fp レジスタの表示: IEEE 倍精度	`print $f0f1`	`<f0=X; <f1=X`
fp レジスタの表示: 等価の 10 進数 (Hex)	`print -flx $f0f1`	`<f0=F`
すべての fp レジスタの表示	`regs -F`	`$x for f0-f15` `$X for f16-f31`
すべてのレジスタの表示	`regs`	`$r; $x; $X`
fp ステータスレジスタの表示	`print -fllx $fsr`	`<fsr=X`
`f0` に単精度 1.0 を設定	`assign $f0=1.0`	`3f800000>f0`
`f0/f1` に倍精度 1.0 を設定	`assign $f0f1=1.0`	`3ff00000>f0; 0>f1`
実行を継続	`cont`	`:c`
シングルステップ	`step (or next)`	`:s`
デバッガの終了	`quit`	`$q`

浮動小数点数を表示する場合、レジスタのサイズは 32 ビットであり、1 つの単精度浮動小数点数は 32 ビットを占有すること (つまり、1 つのレジスタに収まります)、倍精度浮動小数点数は 64 ビットを占有する (つまり、1 つの倍精度数を保持するには 2 つのレジスタが使用されます) に留意してください。16 進数表現では、32 ビットは 8 桁の 16 進数に相当します。adb を使用して FPU レジスタを表示したスナップショットでは、表示は次のような形式になります。

<fpu レジスタ名> <IEEE 16 進数の値> <単精度> <倍精度>

3 番目の列には、2 番目の列に表示されている 16 進数を単精度の 10 進数に変換した値が示されます。4 番目の列は、レジスタのペアを解釈しています。たとえば、f11 の行の 4 番目の列は、f10 および f11 を 64 ビットの IEEE 倍精度数として解釈しています。

f10 および f11 は 1 つの倍精度値の保持に使用されているため、その値の最初の 32 ビットの (f10 行の) 7ff00000 が +NaN として解釈されても無意味になります。64 ビット全体 7ff00000 00000000 の解釈である +Infinity は、意味のある解釈になっています。

最初の 16 個の浮動小数点データレジスタの表示に使用されている adb コマンド $x は、fsr (浮動小数点ステータスレジスタ) も表示します。

```
$x
fsr    40020
f0 400921fb      +2.1426990e+00
f1 54442d18      +3.3702806e+12      +3.1415926535897931e+00
f2         2      +2.8025969e-45
f3         0      +0.0000000e+00      +4.2439915819305446e-314
f4 40000000      +2.0000000e+00
f5         0      +0.0000000e+00      +2.0000000000000000e+00
f6 3de0b460      +1.0971904e-01
f7         0      +0.0000000e+00      +1.2154188766544394e-10
f8 3de0b460      +1.0971904e-01
f9         0      +0.0000000e+00      +1.2154188766544394e-10
f10 7ff00000      +NaN
f11        0      +0.0000000e+00      +Infinity
f12 ffffffff      -NaN
f13 ffffffff      -NaN                 -NaN
f14 ffffffff      -NaN
f15 ffffffff      -NaN                 -NaN
```

次の表は、x86 アーキテクチャーのデバッグコマンドの例を示しています。

表 A-2 いくつかのデバッグコマンド (x86)

処理	dbx	adb
関数へのブレークポイントの設定	stop in myfunct	myfunct:b
行番号へのブレークポイントの設定	stop at 29	n/a

処理	dbx	adb
絶対アドレスへのブレークポイントの設定	n/a	23a8:b
相対アドレスへのブレークポイントの設定	n/a	main+0x40:b
ブレークポイントに達するまで実行	run	:r
ソースコードの表示	list	<pc,10?ia
fp レジスタの表示	print $st0 ... print $st7	$x
すべてのレジスタの表示	refs -F	$r
fp ステータスレジスタの表示	print -fx $fstat	<fstat=X
実行を継続	cont	:c
シングルステップ	step (or next)	:s
デバッガの終了	quit	$q

次の例は、adb のルーチン myfunction に対応するコードの先頭にブレークポイントを設定する 2 つの方法を示しています。最初に次のコマンドを使用できます。

```
myfunction:b
```

次に、myfunction に対応するコードの先頭に対応する絶対アドレスを判別して、その絶対アドレスにブレークを設定できます。

```
myfunction=X
   23a8
23a8:b
```

f95 を指定してコンパイルされた Fortran プログラムのメインサブルーチンは、adb への MAIN_ と認識されています。adb の MAIN_ にブレークポイントを設定するには、次のコマンドを実行します。

```
   MAIN_:b
```

浮動小数点レジスタの内容を検査する場合、dbx コマンドの regs -F で表示される 16 進数値は、数値の 10 進数表現ではなく base-16 表現です。SPARC ベースシステムでは、adb コマンドの $x および $X は、16 進数表現および 10 進数値の両方を表示します。x86 ベースシステムでは、adb コマンドの $x は 10 進数値のみを表示します。SPARC ベースシステムでは、倍精度値の場合、奇数レジスタの横に 10 進数値が表示されます。

オペレーティングシステムではプロセスが最初に使用するまで浮動小数点ユニットが無効にされるため、デバッグしているプログラムがアクセスするまで、浮動小数点レジスタを変更できません。

x86 での対応する出力は次のようになります。

```
$x
80387 chip is present.
cw      0x137f
sw      0x3920
cssel 0x17  ipoff 0x2d93               datasel 0x1f  dataoff 0x5740

 st[0]  +3.24999988079071044921875 e-1            VALID
 st[1]  +5.65391332434795490344196888 e73         EMPTY
 st[2]  +2.000000000000000008881784197            EMPTY
 st[3]  +1.80732183080704405560160047 e-1         EMPTY
 st[4]  +7.9180300235748291015625 e-1             EMPTY
 st[5]  +4.20163903669390492723323 4 e-13         EMPTY
 st[6]  +4.20163903669390492723323 4 e-13         EMPTY
 st[7]  +2.72249992132186946491856 36             EMPTY
```

注記 - x86 の場合、cw は制御ワードで、sw はステータスワードです。

♦ ♦ ♦ 付 録 B

SPARC の動作と実装

この章では、SPARC ベースのワークステーションで使用される浮動小数点ユニットに関連する問題について説明し、特定のワークステーションに最適なコード生成フラグを決定する方法を示します。

B.1 浮動小数点ハードウェア

このセクションでは、多数の SPARC プロセッサを示し、それらがサポートする命令セットと例外処理機能について説明します。

次の表に、最近の SPARC システムで使用されているハードウェア浮動小数点実装を示します。

表 B-1 Oracle Solaris 11 以降でサポートされている SPARC システム

チップ	代表的なシステム	最適なコード生成オプション
T1	T1000、T2000、T6300、CP3060	`-xarch=sparcvis2 -xchip=ultraT1`
T2	T5120、T5220、T6320、CP3260	`-xarch=sparcvis2 -xchip=ultraT2`
T2+	T5140、T5240、T5440	`-xarch=sparcvis2 -xchip=ultraT2plus`
T3	T3-1、T3-2、T3-4	`-xarch=sparcvis3 -xchip=T3`
T4	T4-1、T4-1B、T4-2、T4-4	`-xarch=sparc4 -xchip=T4`
T5	T5-1B、T5-2、T5-4、T5-8	`-xarch=sparc4 -xchip=T5`
M5	M5-32	`-xarch=sparc4 -xchip=M5`
M6	M6-32	`-xarch=sparc4 -xchip=M6`
M7	M7-32	`-xarch=sparc5 -xchip=M7`
SPARC64-VI	M4000、M5000、M8000、M9000	`-xarch=sparcfmaf -xchip=sparc64vi`

チップ	代表的なシステム	最適なコード生成オプション
SPARC64-VII	M3000、M4000、M5000、M8000、M9000	`-xarch=spracima -xchip=sparc64vii`
SPARC64-VII+	M3000、M4000、M5000、M8000、M9000	`-xarch=sparcima -xchip=sparc64viiplus`
SPARC64-X	M10-1、M10-4、M10-4S	`-xarch=sparcace -xchip=sparc64x`

表 B-2 Oracle Solaris 10 Update 10 ではサポートされているが Oracle Solaris 11 ではサポートされていない UltraSPARC システム

UltraSPARC チップ	代表的なシステム	最適なコード生成オプション
`I`	Ex000	`-xarch=sparcvis -xchip=ultra`
`II`	Ex000、E10000	`-xarch=sparcvis -xchip=ultra2`
`IIi`	Ultra-5、Ultra-10	`-xarch=sparcvis -xchip=ultra2i`
`IIe`	Sun Blade 100	`-xarch=sparcvis -xchip=ultra2e`
`III`	Sun Blade 1000、2000	`-xarch=sparcvis2 -xchip=ultra3`
`IIIi`	Sun Blade 1500、2500	`-xarch=sparcvis2 -xchip=ultra3i`
`IIICu`	Sun Blade 1000、2000	`-xarch=sparcvis2 -xchip=ultra3cu`
`IV`	V490、V890、Ex900、E20K、E25K	`-xarch=sparcvis2 -xchip=ultra4`
`IV+`	V490、V890、Ex900、E20K、E25K	`-xarch=sparcvis2 -xchip=ultra4plus`

Oracle Solaris 10 Update 10 以前の Oracle Solaris リリースで Oracle Solaris Studio 12.4 を使用してコンパイルされたプログラムはサポート対象ではありませんが、ほとんどの場合、以前の Oracle Solaris Studio リリースをサポートする旧 SPARC システムで実行できます。このようなプラットフォームでテストする実行可能ファイルを作成するには、次のオプションを使用してコンパイルしてみてください。

```
-m32 -xarch=generic -xchip=generic
```

サポート対象のソリューションについては、使用するもっとも古い Oracle Solaris リリース上でコンパイルし、その Oracle Solaris リリースでサポートされる最新の Oracle Solaris Studio バージョンを使用してコンパイルします。

前述の表の最後の列は、それぞれの FPU で最速のコードを取得するために使用するコンパイラフラグを示しています。これらのフラグは、コード生成の 2 つの独立した属性を制御します。`-xarch` フラグはコンパイラが使用できる命令セットを決定し、`-xchip` フラグはコードのスケ

ジューリングにおけるプロセッサのパフォーマンス特性に関するコンパイラによる仮定を決定します。デフォルトの -xarch を使用するか、明示的な -xarch=sparc を使用してコンパイルされたプログラムは、前述の SPARC ベースシステムすべてで実行されますが、最新のプロセッサの機能を十分に活用できない場合があります。同様に、特定の -xchip 値を使用してコンパイルされたプログラムは、-xarch で指定された命令セットをサポートするすべての SPARC ベースシステムで実行されますが、指定以外のプロセッサを備えたシステムでは実行速度が低下する場合があります。

UltraSPARC I、UltraSPARC II、UltraSPARC IIe、UltraSPARC IIi、UltraSPARC III、UltraSPARC IIIi、UltraSPARC IV、および UltraSPARC IV+ 浮動小数点ユニットは、4 倍精度の命令を除き、『*SPARC Architecture Manual*』バージョン 9 で定義されている浮動小数点命令セットを実装します。具体的には、これらは 32 倍精度の浮動小数点レジスタを提供しています。-xarch=sparc でコンパイルすると、コンパイラはこれらのすべての機能を使用できます。これらのプロセッサは、命令セットの拡張機能も提供しています。今までのこれらの追加命令は、次の -xarch 値で有効になります。

- sparcvis
- sparcvis2
- sparcvis2
- sparc4
- sparc5

これらの追加命令の多くは、コンパイラによって自動的に生成されることはまれですが、その場合はアセンブリコードで使用できます。

-xarch と -xchip オプションは、-xtarget マクロオプションを使用して同時に指定できます。-xtarget フラグは、-xarch、-xchip、および -xcache フラグの適切な組み合わせとしてのみ展開します。デフォルトのコード生成オプションは -xtarget=generic です。-xarch、-xchip、および -xtarget の値の完全なリストを含む詳細については、cc(1)、CC(1)、および f95(1) のマニュアルページと『Oracle Solaris Studio 12.4: Fortran ユーザーズガイド 』、『Oracle Solaris Studio 12.4: C ユーザーガイド 』、および『Oracle Solaris Studio 12.4: C++ ユーザーズガイド 』のコンパイラマニュアルを参照してください。

B.1.1 浮動小数点ステータスレジスタおよびキュー

すべての SPARC 浮動小数点ユニットは、実装する SPARC アーキテクチャーのバージョンとは関係なく、FPU に関連したステータスビットおよびコントロールビットを含む浮動小数点ス

テータスレジスタ (FSR) を提供しています。遅延浮動小数点トラップを実装する SPARC FPU はすべて、現在実行中の浮動小数点命令に関する情報を含む浮動小数点キュー (FQ) を提供しています。FSR はユーザーソフトウェアからアクセスして、発生した浮動小数点例外を検出し、丸め方向、トラップ、および非標準演算モードを制御できます。FQ はオペレーティングシステムカーネルによって浮動小数点トラップの処理に使用され、通常、ユーザーソフトウェアからは見えません。

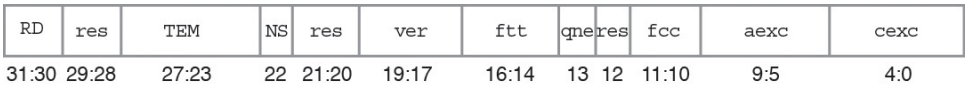
ソフトウェアは、FSR をメモリーに格納する `STFSR` 命令と、メモリーからロードする `LDFSR` 命令を介して、浮動小数点ステータスレジスタにアクセスします。SPARC アセンブリ言語では、これらの命令は次のように記述されます。

```
st      %fsr, [addr]  ! store FSR at specified address
ld      [addr], %fsr  ! load FSR from specified address
```

Sun Studio コンパイラに付属のライブラリを含むディレクトリにあるインラインテンプレートファイル `libm.il` には、`STFSR` および `LDFSR` 命令の使用例があります。

次の図に、浮動小数点ステータスレジスタのビットフィールドのレイアウトを示します。

図 B-1 SPARC 浮動小数点ステータスレジスタ



バージョン 7 および 8 の SPARC アーキテクチャーでは、図に示すように FSR は 32 ビットを占めます。バージョン 9 では、FSR は 64 ビットに拡張されますが、そのうち下位の 32 ビットがこの図に一致し、上位 32 ビットには 3 つの追加浮動小数点条件コードフィールドが入っているのみで、大部分が未使用です。

この図では、`res` は予約されているビットを示し、`ver` は、FPU のバージョンを示す読み取り専用フィールドであり、`ftt` と `qne` は浮動小数点トラップを処理するときにシステムによって使用されます。残りのフィールドについては、次の表で説明します。

表 B-3 浮動小数点ステータスレジスタフィールド

フィールド	内容
RM	丸め方向モード

フィールド	内容
TEM	トラップ有効化モード
NS	非標準モード
fcc	浮動小数点条件コード
aexc	累積例外フラグ
cexc	現在の例外フラグ

RM フィールドには、浮動小数点演算での丸め方向を指定する 2 ビットが保持されます。NS ビットは、非標準演算モードを実装する SPARC FPU ではこのモードを有効にし、実装していない場合はこのビットが無視されます。fcc フィールドには、浮動小数点比較命令によって生成され、分岐演算および条件付き移動演算で使用される浮動小数点条件コードが保持されます。TEM、aexc、および cexc フィールドには、トラップを制御し、5 つの IEEE 754 浮動小数点例外それぞれについて累積例外フラグと現在の例外フラグを記録する 5 つのビットが含まれます。これらのフィールドは、次の表に示すようにさらに分割されます。

表 B-4 例外処理フィールド

フィールド	レジスタ内の対応するビット				
TEM、トラップ有効化モード	NVM	OFM	UFM	DZM	NXM
	27	26	25	24	23
aexc、累積例外フラグ	nva	ofa	ufa	dza	nxa
	9	8	7	6	5
cexc、現在の例外フラグ	nvc	ofc	ufc	dzc	nxc
	4	3	2	1	0

(上記の記号 NV、OF、UF、DZ、および NX は、それぞれ無効演算、オーバーフロー、アンダーフロー、ゼロ除算、および不正確な例外を表します)。

B.1.2 ソフトウェアサポートを必要とする特殊な場合

ほとんどの場合、SPARC 浮動小数点ユニットは、ハードウェア内で完全に命令を実行し、ソフトウェアサポートを必要とすることはありません。ただし、次の 4 つの場合は、ハードウェアでは浮動小数点命令が正常に完了しません。

- 浮動小数点ユニットが無効になっている場合。

- SPARC FPU での 4 倍精度命令など、命令がハードウェアによって実装されていない場合。
- ハードウェアが命令のオペランドに対して正しい結果を出すことができない場合。
- 命令によって IEEE 754 浮動小数点例外が発生し、その例外のトラップが有効になっている場合。

どの場合でも初期応答は同じです。プロセスがシステムカーネルにトラップを発行すると、システムカーネルがトラップの原因を特定して適切なアクションを実行します。「トラップ」という用語は、通常の制御フローの割り込みを意味します。最初の 3 つの場合では、カーネルはソフトウェア内でトラップ命令をエミュレートします。エミュレートされた命令でもまた、トラップが有効になっている例外が発生する可能性があることに注意してください。

上記の最初の 3 つの場合では、エミュレートされた命令によって、トラップが有効になっている IEEE 浮動小数点例外が発生しなければ、カーネルは命令を完了します。命令が浮動小数点の比較である場合は、カーネルは結果を反映させるために条件コードを更新します。命令が算術演算である場合は、適切な結果をデスティネーションレジスタに配布します。カーネルはまた、命令によって発生したすべての (トラップされていない) 例外を反映するように現在の例外フラグも更新し、これらの例外を累積例外フラグに加えます。続いてカーネルは、トラップが行われた位置からプロセスの実行を継続するように準備します。

ハードウェアが実行する、またはカーネルソフトウェアがエミュレートする命令によって、トラップが有効になっている IEEE 浮動小数点例外が発生した場合、その命令は完了しません。デスティネーションレジスタ、浮動小数点条件コード、および累積例外フラグは変更されず、トラップの原因となった特定の例外を反映するように現在の例外フラグが設定され、カーネルは `SIGFPE` シグナルをプロセスに送信します。

次の擬似コードは、浮動小数点トラップの処理の概要を示しています。`aexc` フィールドは通常、ソフトウェアでしかクリアできないことに注意してください。

```
FPop provokes a trap;
if trap type is fp_disabled, unimplemented_FPop, or
  unfinished_FPop then
    emulate FPop;
texc = all IEEE exceptions generated by FPop;
if (texc and TEM) = 0 then
    f[rd]  = fp_result;  // if fpop is an arithmetic op
    fcc = fcc_result;  // if fpop is a compare
    cexc = texc;
    aexc = (aexc or texc);
else
    cexc = trapped IEEE exception generated by FPop;
    throw SIGFPE;
```

多くの浮動小数点命令をカーネルがエミュレートする必要がある場合、プログラムのパフォーマンスは大幅に低下します。この低下の相対的な発生頻度は、トラップの種類など、いくつかの要因によって異なります。

通常の環境下では、`fp_disabled` トラップは、プロセスごとに一度だけ発生します。システムカーネルは、プロセスが最初に開始されるときに浮動小数点ユニットを無効にします。このため、プロセスによって実行される最初の浮動小数点演算によってトラップが発生します。トラップを処理したあと、カーネルによって浮動小数点ユニットが有効なり、プロセスの実行中は有効なままになります。(システム全体に対して浮動小数点ユニットを無効にすることは可能ですが、これは推奨されていません。カーネルまたはハードウェアのデバッグプロセスでのみ実行します)。

`unimplemented_FPop` トラップは、浮動小数点ユニットが実装していない命令に遭遇すると必ず発生します。現在の SPARC 浮動小数点ユニットのほとんどは、4 倍精度命令を除き、少なくとも『*SPARC Architecture Manual*』バージョン 8 に定義されている命令セットを実装しており、また、Oracle Solaris Studio コンパイラは 4 倍精度命令を生成しないため、この種のトラップは、`-xarch=sparc` でコンパイルされたほとんどのシステムで発生することはありません。

残る 2 種類のトラップである `unfinished_FPop` およびトラップされた IEEE 例外は通常、NaN、無限大、および非正規数を含む特殊な演算状況に関連しています。

B.1.2.1 IEEE 浮動小数点例外、NaN、および無限大

浮動小数点命令がトラップが有効になっている IEEE 浮動小数点例外に遭遇すると、その命令は完了されません。代わりにシステムから `SIGFPE` シグナルがプロセスに送信されます。プロセスが `SIGFPE` シグナルハンドラを確立している場合は、そのハンドラが呼び出され、確立していない場合はプロセスが中止します。トラップはほとんどの場合、例外発生時にプログラムを中止させることを目的として有効になっているため、メッセージを出力しプログラムを終了させるシグナルハンドラを呼び出した場合、またはシグナルハンドラがインストールされていないときにシステムのデフォルト動作に分類し直した場合、ほとんどのプログラムでは、トラップされた IEEE 浮動小数点例外の発生は少なくなります。ただし、第4章「例外と例外処理」で説明しているように、シグナルハンドラがトラップ命令の結果を供給し、実行を継続することもできます。多くの浮動小数点例外がトラップされ、この方法で処理される場合、パフォーマンスが大幅に低下する可能性があることに注意してください。

SPARC 浮動小数点ユニットの中には、トラップが無効になっている場合や、命令によってトラップが有効になっている例外が発生しない場合でも、少なくとも無限オペランド、NaN オペランド、あるいは IEEE 浮動小数点例外を含むケースをトラップするものがあります。これ

は、ハードウェアがこのような特殊なケースをサポートしていない場合に発生します。代わりに unfinished_FPop トラップが生成され、そのままカーネルエミュレーションソフトウェアで命令を続行させます。SPARC FPU が異なれば、unfinished_FPop トラップを発生させる条件も異なります。たとえば、もっとも初期の SPARC FPU は、トラップが有効かどうかにかかわらず、すべての IEEE 浮動小数点例外をトラップしますが、UltraSPARC FPU は、浮動小数点例外のトラップが有効であり、命令によって例外が発生するかどうかをハードウェアが判断できない場合に、悲観的にトラップすることがあります。ただし、最新の SPARC プロセッサであればどれでも、ハードウェアの例外的なケースがすべて処理されるため、unfinished_FPop トラップが生成されることはありません。

ほとんどの unfinished_FPop トラップは浮動小数点例外とともに発生するため、プログラムでは、例外フラグのテスト、結果のトラップおよび代用、または例外発生時の停止などの例外処理を採用することによって、これらのトラップの過剰な発生を回避できます。例外処理にかかるコストと、例外による unfinished_FPop トラップの発生を許容するコストのバランスを取るようにしてください。

B.1.2.2 非正規数と非標準演算

一部の SPARC 浮動小数点ユニットが unfinished_FPop でトラップを発生させるもっとも一般的な状況には、非正規数が関係しています。これまでの SPARC 浮動小数点ユニットの多くは、浮動小数点演算に非正規オペランドが含まれているか、ゼロ以外の非正規結果、つまり段階的アンダーフローを招く結果を必ず生成する場合に常にトラップします。アンダーフローの発生はそれほど多くありませんが、プログラムが困難であり、また、アンダーフローした中間結果の正確さは、計算の最終結果全体の正確さにほとんど影響しないため、SPARC アーキテクチャーでは、非正規数を含む unfinished_FPop トラップによるパフォーマンス低下の回避方法をユーザーに提供する非標準演算モードを定義しています。

SPARC アーキテクチャーでは非標準演算モードを明確には定義していません。このモードが有効になっていると、これをサポートするプロセッサが IEEE 754 標準に準拠しない結果を生成する可能性があることを述べているだけです。ただし、このモードをサポートする既存の SPARC 実装はすべて、このモードを使用して段階的アンダーフローを無効にし、非正規オペランドと結果をゼロに置き換えます。

SPARC 実装すべてに非標準モードが用意されているわけではありません。このモードをサポートしていない SPARC 実装では、単に無視されるため、非標準モードでも数値結果と例外結果は同じです。これらのプロセッサでは、段階的アンダーフローでパフォーマンスが低下することはありません。

段階的アンダーフローがプログラムのパフォーマンスに影響しているかどうかを判断するには、最初にアンダーフローが発生しているかどうかを判断し、次にプログラムがどれだけのシステム時間を使用しているかを確認する必要があります。アンダーフローが発生しているかどうかを判断するには、数学ライブラリ関数 `ieee_retrospective()` を使用して、プログラムの終了時にアンダーフロー例外フラグが発生するかどうかを確認できます。Fortran プログラムはデフォルトで `ieee_retrospective()` を呼び出します。C および C++ プログラムは、終了する前に明示的に `ieee_retrospective()` を呼び出す必要があります。アンダーフローが発生した場合、`ieee_retrospective()` が次のようなメッセージを出力します。

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

プログラムがアンダーフローに遭遇した場合、`time` コマンドでプログラムの実行時間を測定することによって、プログラムがどれだけのシステム時間を使用しているかを判断できます。

```
demo% /bin/time myprog > myprog.output

real 305.3
user 32.4
sys 271.9
```

システム時間 (前述の出力の 3 番目の数値) が異常に高い場合、複数のアンダーフローが原因であると考えられます。この場合、プログラムが段階的アンダーフローの正確さに依存していなければ、非標準モードを有効にしてパフォーマンスを高めることができます。

そのためには、次の 2 つの方法を利用します。まず、マクロ `-fast` および `-fnonstd` の一部として暗黙的に指定されている `-fns` フラグを使用してコンパイルし、プログラムの起動時に非標準モードを有効にすることができます。次に、付加価値数学ライブラリ `libsunmath` では、非標準モードをそれぞれ有効および無効にする 2 つの関数を提供しており、`nonstandard_arithmetic()` を呼び出すと、非標準モードが有効になり (サポートされている場合)、`standard_arithmetic()` を呼び出すと IEEE 動作が復元されます。これらの関数を呼び出す C および Fortran の構文は次のとおりです。

C、C++	`nonstandard_arithmetic();`
	`standard_arithmetic();`
Fortran	`call nonstandard_arithmetic()`
	`call standard_arithmetic()`

注意 - 非標準演算モードでは、段階的アンダーフローの利点である正確さが失われるため、使用の際は注意が必要です。段階的アンダーフローについての詳細は、第2章「IEEE 演算」を参照してください。

B.1.2.3 非標準演算およびカーネルエミュレーション

非標準モードを実装する SPARC 浮動小数点ユニットでは、このモードを有効にすると、ハードウェアが非正規オペランドをゼロとして扱い、非正規結果をゼロにフラッシュします。ただし、トラップした浮動小数点命令のエミュレートに使用されるカーネルソフトウェアは、非標準モードを実装していません。その理由の 1 つは、このモードの影響が定義されておらず、実装に依存するためであり、また 1 つは、段階的アンダーフローの処理にかかる追加コストが、ソフトウェアでの浮動小数点演算のエミュレーションのコストに比べれば微々たるものであるためです。

非標準モードの影響を受ける浮動小数点演算に割り込みが発生する場合 (たとえば、発行後にコンテキストスイッチが行われたり、別の浮動小数点命令によってトラップが発生したため完了しなかった場合など)、標準 IEEE 演算を使用して、カーネルソフトウェアによってエミュレーションされます。そのため、特殊な状況下では、非標準モードで実行しているプログラムが、システム負荷に応じてわずかに異なる結果を生成する可能性があります。実際には、この動作は観察されてはいません。これは、数百回の演算のうちある 1 つの演算が段階的アンダーフローと突発的アンダーフローのどちらで実行されるかということに対して感度の高いプログラムだけに影響すると考えられます。

B.2 *fpversion*(1) 関数: FPU に関する情報の検索

コンパイラに付属の *fpversion* ユーティリティーは、装備されている CPU を識別し、プロセッサおよびシステムバスのクロック速度を推定します。`fpversion` は、CPU および FPU によって格納されている識別情報を解釈して、CPU と FPU の種類を特定します。このユーティリティーは、予測可能な所要時間内に動作する単純な命令を実行するループの時間を計測することによって、クロック速度を推定します。時間計測の正確さを高めるため、このループは何度も実行されます。このため、`fpversion` は即座には完了しません。実行には数秒かかる場合があります。

`fpversion` は、ホストシステムでの使用に最適な `-xtarget` コード生成オプションも報告します。

T4-2 サーバーでは、`fpversion` は次のような情報を表示します。これはタイミングやマシン構成の違いによって異なる場合があります。

```
demo% fpversion
 A SPARC-based CPU is available.
 Kernel says CPU's clock rate is 1500.0 MHz.
 Kernel says main memory's clock rate is 150.0 MHz.

 Sun-4 floating-point controller version 0 found.
 An UltraSPARC chip is available.

 Use "-xtarget=T4 -xcache=16/32/4/8:128/32/8/8:4096/64/16/64" code-generation option.

Hostid = hardware_host_id
```

詳細は、*fpversion*(1) のマニュアルページを参照してください。

♦ ♦ ♦ 付 録 C

C x86 の動作と実装

この付録では、x86/x64 ベースのシステムで使用される浮動小数点ユニットに関連した x86/x64 と SPARC の互換性の問題について説明します。

C.1 サポートされているシステムのコード生成

Oracle Solaris は、Intel、AMD、その他チップベンダーの x86 プロセッサを備えた、Oracle、Sun、その他システムベンダーの多くのシステムをサポートしています。特定の Oracle Solaris リリースでは、これらのチップを含む特定のシステムを多数サポートしています。特定の Oracle Solaris リリースについては、対応するハードウェア互換性リストを参照してください。

Oracle Solaris 11 は、64 ビットのアドレス指定をサポートする x86 プロセッサをサポートしています。Oracle Solaris 10 Update 10 は、ハードウェア浮動小数点および 120 MHz 以上のクロック周波数を備えた 64 ビットプロセッサと多数の 32 ビット専用 x86 プロセッサをサポートしています。

大多数のシステムの要求を満たすコードを生成するには、`-m32 -xarch=generic -xchip=generic` フラグを使用してコンパイルします。次の表に、いくつかの代表的な Oracle および Sun x86 システムに使用する特定のコード生成オプションを示します。

システム	コード生成オプション
Ultra 20	`-xarch=sse2a -xchip=opteron`
X2200	`-xarch=amdsse4a -xchip=amdfam10`
X6250	`-xarch=sse3 -xchip=core2`
X4170	`-xarch=aes -xchip=westmere`
X2-4	`-xarch=sse4_2 -xchip=nehalem`
X3-2	`-xarch=avx -xchip=sandybridge`

システム	コード生成オプション
X4-2 X4-4	-xarch=avx_i -xchip=ivybridge
?	-xarch=avx2 -xchip=haswell

何百もの個別の x86 チップがあり、それぞれ複雑な命名法に従っています。

特定のシステムを最適化するために必要なコンパイラの処理を把握するには、`cc -dryrun -native` を使用することをお勧めします。数種類の x86 システムを対象としたコードを生成する場合は、もっとも古いシステムに適したオプションを使用すると、すべてのシステムに適したものになります。

C.2 SPARC との差異

Oracle Solaris Studio コンパイラは通常、SPARC と x86 で同様に実行されるコードを生成します。ただし、x86 ベースシステムについては、次の重要な相違点に注意してください。

- x87 浮動小数点レジスタは 80 ビット幅です。x87 浮動小数点レジスタスタックが使用されている場合、算術計算の中間結果が拡張倍精度 (80 ビット) になるため、計算結果が異なることがあります。`-fstore` フラグはこのような不一致を最小限に抑えます。ただし、`-fstore` フラグを使用するとパフォーマンスが低下します。Oracle Solaris Studio 12.4 はデフォルトでは単精度および倍精度の式評価に x87 レジスタを使用しませんが、`-xarch=386` が指定されている場合、x87 ハードウェア超越命令が使用されている場合、または拡張倍精度変数が使用されている場合には、x87 レジスタが使用されます。
- 単精度または倍精度浮動小数点数が x87 浮動小数点レジスタスタックにロードされるか、メモリーに格納されるたびに、拡張倍精度 (80 ビット) に対する双方向の変換が行われます。このため、浮動小数点数のロードと格納によって例外が発生することがあります。`-m32` を使用すると、浮動小数点サブルーチンのオペランドと結果は、x87 レジスタで渡されます。
- x87 浮動小数点レジスタスタックが使用中の場合は、マイクロコードによって段階的アンダーフローがハードウェアに実装されます。非標準モードはありません。
- `fpversion` ユーティリティーはありません。
- 拡張倍精度 (80 ビット) 形式では、浮動小数点値を表現しない特定のビットパターンが認められています (表2-8「各ビットパターンによって表される値 (x86)」 参照)。ハードウェアは通常、これらの「未サポート形式」をシグナルを発生する NaN として扱いますが、数学ライブラリではこのような表現の処理に一貫性がありません。これらのビットパターンはハードウェアによって生成されることはないため、配列の末尾を超えた読み取りなどの無効なメモ

リー参照、または C の union 構造体を介した、メモリー内のデータのある型から別の型への明示的な強制型変換が行われる場合のみ作成されます。このため、ほとんどの数値プログラムでは、これらのビットパターンは生じません。

♦ ♦ ♦ 付 録 D

D 『浮動小数点演算について計算機科学者は何を知っておくべきか』の付録

この『*数値計算ガイド*』の対象読者すべてにとって、David Goldberg 著 1991 年 3 月 Computing Surveys 発行の論説『*浮動小数点演算について計算機科学者は何を知っておくべきか*』が役立ちます (http://dl.acm.org/citation.cfm?id=103163 を参照)。

この概略は次のとおりです。

浮動小数点演算は、概して難解な問題と捉えられています。コンピュータシステム内部には、浮動小数点が至る所に存在することを考えると、これはむしろ意外なことです。ほとんどの言語で浮動小数点のデータ型が使用されています。PC からスーパーコンピュータまで、コンピュータには浮動小数点アクセラレータが使用されており、浮動小数点アルゴリズムをコンパイルするために、さまざまなコンパイラが随時呼び出されます。事実上、オペレーティングシステムはすべて、オーバーフローなどの浮動小数点例外に対応していると言えます。この論説では、コンピュータシステムの設計者に直接影響を与える浮動小数点の側面について説明します。浮動小数点表現の背景、丸め誤差、続いて IEEE 浮動小数点標準について説明し、最後にコンピュータシステム設計者が浮動小数点を適切にサポートする方法の例を紹介します。

この付録は、発行済みの Goldberg 氏の論説の一部ではありません。特定のポイントを明確にし、読者が論説から推測する可能性のある IEEE 標準に関する誤解を正すために追加されました。この資料は David Goldberg 氏による文書ではありませんが、氏の許可のもと記載しています。754-1985 と 854-1987 標準は、2 進と 10 進の両方の浮動小数点演算を指定した 754-2008 に置き換えられています。これは Goldberg 氏の論説には影響しません。

この付録では特に、172 ページの「IEEE 754 実装間の相違」について説明しています。このトピックには、次のサブトピックがあります。

- 173 ページの「現在の IEEE 754 の実装」
- 175 ページの「拡張ベースシステムでの計算の落とし穴」
- 180 ページの「拡張精度におけるプログラミング言語のサポート」
- 185 ページの「結論」

D.1 IEEE 754 実装間の相違

Goldberg 氏の論説では、プログラマがそのプログラムの正確さと精度を浮動小数点演算のプロパティーに依存する場合があるので、浮動小数点演算を慎重に実装する必要があると述べています。特に、IEEE 標準は慎重な実装が必要であり、この標準に従ったシステム上でのみ正しく機能し、正確な結果をもたらす有用なプログラムを作成することが可能です。一部の読者は、このようなプログラムがすべての IEEE 標準に移植できると判断する可能性もあります。実際、「プログラムが 2 つのマシン間で移され、どちらも IEEE 演算をサポートしている場合、中間結果が異なっていても、演算の差ではなく、ソフトウェアバグによるものに違いない」という見解が真であるならば、移植可能なソフトウェアの作成は簡単になります。

しかし、IEEE 標準では、準拠するすべてのシステムで同じプログラムが同一の結果をもたらすことを保証していません。ほとんどのプログラムは実際には、さまざまな理由により、別々のシステムで異なる結果を生成します。ひとつには、多くのプログラムは、システムライブラリに用意されている初等関数を使用しますが、標準はこれらの関数を完全には規定していないからです。1985 標準は、10 進形式と 2 進形式との間の数値変換を完全には規定しておらず、超越関数をまったく規定していませんでした。

ほとんどのプログラマは、IEEE 標準で定められた数値形式と演算だけを使用するプログラムでさえ、別々のシステムでは算出する結果が異なるという可能性を認識していません。実際、標準規格の作成者は、別々の実装では異なる結果を取得できるようにすることを意図していました。その意図は、IEEE 754 標準での*デスティネーション*という用語の定義で明らかです。「デスティネーションは、ユーザーによって明示的に指定される場合も、システムによって暗黙的に与えられる場合 (たとえば、部分式の中間結果やプロシージャーの引数など) もあります。一部の言語には、デスティネーションにおける中間計算の結果をユーザーが制御できないようにするものがあります。それでもこの標準は、デスティネーションの形式とオペランドの値を単位として、演算の結果を定義します」(IEEE 754-1985、7 ページ)。つまり、IEEE 標準では、それぞれの結果が配置先のデスティネーションの精度に正しく丸められる必要がありますが、デスティネーションの精度をユーザーのプログラムで判断することを必要としていません。したがって、別々のシステムでは、デスティネーションの結果の精度が異なることがあるため、これらのシステムがすべて標準に準拠していても、同じプログラムで異なる結果が生成され、大幅に異なる結果になる場合もあります。

参照先の論説のいくつかの例は、浮動小数点演算を丸める方法に関する知識に依存しています。このような例を信頼するには、プログラマは、どのようにプログラムが解釈されるか、特に IEEE システムでは、各算術演算のデスティネーションの精度がどのようになるかについて予測できることが必要になります。しかし、IEEE 標準のデスティネーションの定義における盲点によって、どのようにプログラムが解釈されるかを認識するプログラマの能力が損なわれます。

結果的に、Goldberg 氏の論説に示す例のいくつかは、高級言語で一見、移植可能なプログラムとして実装された場合、一般に、デスティネーションの結果がプログラマの予想と異なる精度になるような IEEE システムでは正しく機能しない可能性があります。ほかの例には機能するものもありますが、例が機能するかどうかの証明は、平均的なプログラマの能力の範疇外である可能性があります。

この付録では、IEEE 754 演算の既存の実装は、通常は実装で使用するデスティネーション形式の精度に基づいて分類されています。論説から、一部の例では、プログラムの予想より結果の精度が大きい場合、予想された精度が使用された場合には正しくなる結果でも、誤った結果が演算されることを示しています。論説では、想定外の精度によってプログラムが無効にならない場合でも、その精度を処理するために必要な作業について解説するため、1 つの論証が訂正されています。これらの例では、IEEE 標準のさまざまな規定にもかかわらず、この標準規格が異なる実装で許容している相違によって、動作を正確に予測できる移植可能で効率的な数値ソフトウェアの作成が妨げられている可能性を示唆しています。このようなソフトウェアを開発するには、プログラムが依存する浮動小数点演算セマンティックスを表現できるように、IEEE 標準が許容する多様性を制限するプログラミング言語と環境を最初に作成する必要があります。2008 バージョンの 1985 標準では、このようなプログラミング言語に関する推奨事項を定めています。

D.1.1 現在の IEEE 754 の実装

IEEE 754 演算の現在の実装は、ハードウェアで異なる浮動小数点形式のサポートの程度で区別される 2 つのグループに分けられます。Intel x86 ファミリのプロセッサに例示される拡張ベースシステムは、`-xarch=386` オプションでコンパイルされており、拡張倍精度形式を完全にサポートしていますが、単精度と倍精度については一部しかサポートしていません。このシステムには、単精度と倍精度のデータをロードおよび格納する命令が用意されており、これらのデータをその場で拡張倍精度形式に、または拡張倍精度形式から変換します。また、算術演算の結果が、拡張倍精度形式でレジスタに保持されている場合でも、単精度または倍精度に丸められる特殊モード (デフォルトではない) も用意されています。Motorola 68000 シリーズプロセッサは、これらのモードで、単精度または倍精度形式の範囲に結果を丸めます。Intel x86 および互換プロセッサは、単精度または倍精度形式に結果を丸めますが、保持する範囲は拡張倍精度形式と同じです。ほとんどの RISC プロセッサを含む単精度/倍精度システムでは、単精度および倍精度形式を完全にサポートしていますが、IEEE 準拠の拡張倍精度形式はサポートしていません。x86 SSE2 拡張機能では、SSE2 レジスタで単精度/倍精度システムを提供しますが、拡張精度レジスタでは拡張精度を引き続きサポートします。

拡張ベースシステムと単精度/倍精度システムで計算の動作がどのように異なるかを確認するために、次のように、Goldberg 氏の論説の例「システムの諸側面」の C バージョンについて考えてみます。

```
int main() {
    double  q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

定数 3.0 と 7.0 は倍精度浮動小数点数と解釈され、式 3.0/7.0 は `double` データ型を継承します。単精度/倍精度システムでは、倍精度のほうが効率的に使用できる形式のため、この式は倍精度で評価されます。したがって、`q` には正しく倍精度に丸められた値 3.0/7.0 が代入されます。次の行で、式 3.0/7.0 はふたたび倍精度で評価され、結果は、`q` に直前に代入した値に等しくなるので、プログラムは予想どおり「Equal」と出力します。

拡張ベースシステムでは、式 3.0/7.0 の型が `double` であっても、商はレジスタにおいて拡張倍精度形式で計算されるため、デフォルトモードで拡張倍精度に丸められます。しかし、結果の値が変数 `q` に代入されるときにとメモリーに格納され、`q` が `double` と宣言されているために値は倍精度に丸められます。次の行で、式 3.0/7.0 はふたたび拡張精度で評価されることがあり、`q` に格納された倍精度値とは異なる結果が生じ、この結果、プログラムは「Not equal」と出力します。

ほかの結果も考えられます。コンパイラは、2 行目の式 3.0/7.0 の値を格納し、丸めたあとで `q` と比較することも、格納せずに拡張精度で `q` をレジスタに保持することもできます。最適化コンパイラは、コンパイル時に式 3.0/7.0 を倍精度で、あるいは拡張倍精度で評価することが考えられます。ある x86 コンパイラにおいて、プログラムは、最適化でコンパイルするときには「Equal」を出力し、デバッグ用にコンパイルするときには「Not Equal」を出力します。また、拡張ベースシステムの一部のコンパイラには、丸め精度モードを自動的に変更し、より広い範囲を使用して、レジスタ内に結果を生成する演算によってその結果を単精度または倍精度に丸めるものもあります。そのため、これらのシステムでは、そのソースコードを読み取り、IEEE 754 演算の基本的な解釈を適用するだけでは、プログラムの動作を予測できません。IEEE 754 準拠環境を提供できない原因は、ハードウェアにもコンパイラにもありません。ハードウェアは、要求されているとおり各デスティネーションに丸めた結果を正しく配布しており、コンパイラは、許可されているとおりユーザーの制御の及ばないデスティネーションに中間結果を代入しています。

このセクションの残りでは、Oracle Solaris Studio 12 以前のリリースでデフォルトであった -xarch=386 でコンパイルされた x86 の動作について説明します。Oracle Solaris Studio 12.4 では、デフォルトは -xarch=sse2 です。

D.1.2 拡張ベースシステムでの計算の落とし穴

世間一般の通念では、拡張ベースシステムは、常に単精度/倍精度システムと少なくとも同じ精度、多くの場合はそれを超える精度を提供しているため、単精度/倍精度システムで配布される結果以上に正確ではないとしても、最低限は正確な結果を生成する必要があるとされています。次のセクションに示す単純な例、およびその例に基づく微妙なプログラムでは、この通念が単なる理想であることを示しています。単精度/倍精度システムには実際に移植できる、一見、移植性のあるプログラムの中には、コンパイラとハードウェアが重なり合って、プログラムの予想以上の精度を提供しようとするため、拡張ベースシステムで正しくない結果をもたらすものがあります。

現在のプログラミング言語では、プログラムが予想する精度を指定することが困難になっています。Goldberg 氏の論説の言語とコンパイラのセクションで述べているように、多くのプログラミング言語では、10.0*x などの式が同じコンテキストで複数出現したときに、それぞれが同じ値に評価される必要があるとは指定しません。この点に関しては、Ada などの一部の言語では、IEEE 標準以前のさまざまな演算のバリエーションの影響を受けています。これより新しい ANSI-C などの言語は、標準に準拠した拡張ベースシステムに影響を受けています。実際、ANSI C 標準では明示的に、コンパイラが浮動小数点式を、通常その型に関連付けられている精度よりも広い精度に評価することを許可しています。結果として、式 10.0*x の値は、次のようなさまざまな要因によって異なることがあります。式がただちに変数に代入されるのか、それともより大きな式のサブエクスプレッションとして表現されるのか、式が比較に関係するかどうか、式が引数として関数に渡されるかどうか、渡される場合、引数は値と参照のどちらとして渡されるのか。現在の精度モード。プログラムのコンパイルに使用された最適化のレベル、プログラムのコンパイル時にコンパイラが使用した精度モードと式の評価方法、などです。

予測のつかない式の評価のすべての原因が言語規格にあるわけではありません。拡張ベースシステムは、可能なかぎり式が拡張精度レジスタで評価されるときにもっとも効率的に実行しますが、格納する必要のある値は、要求されるもっとも短い精度で格納されます。10.0*x がどこでも同じ値に評価されるように言語に要求させると、これらのシステムのパフォーマンス低下を避けられなくなります。これらのシステムで、構文的に同等のコンテキストで 10.0*x を別の値に評価すること許可すると、それ自体のペナルティーが正確な数値ソフトウェアのプログラマに課され、意図した意味を表現するためにプログラムの構文に依存できなくなります。

次の例では、指定された式が常に同じ値に評価されるという仮定に、実際のプログラムが依存しているかどうかを調べます。Goldberg 氏の論説の定理 4 にある、Fortran で記述された ln(1 + x) を計算するアルゴリズムを参照してください。

```
real function log1p(x)
real x
if (1.0 + x .eq. 1.0) then
   log1p = x
else
   log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

拡張ベースシステムでは、コンパイラは 3 行目の式 `1.0 + x` を拡張精度で評価し、その結果を `1.0` と比較します。ただし、同じ式が 6 行目のログ関数に渡される場合、コンパイラはその値をメモリーに格納して、単精度に丸めることができます。したがって、拡張精度では `1.0 + x` が `1.0` に丸められるほど `x` は小さくないが、単精度では `1.0 + x` が `1.0` に丸められるほどには小さい場合、`log1p(x)` で返される値は `x` ではなくゼロになり、相対誤差は 1 で 5ε よりもわずかに大きくなります。同様に、サブエクスプレッション `1.0 + x` が繰り返し出現する 6 行目の式の残りが拡張精度で評価されるとします。この場合、`x` が小さいが、単精度で `1.0 + x` が `1.0` に丸められるほど小さくはない場合、`log1p(x)` で返される値が正しい値を `x` と同じくらい上回り、この場合も相対誤差は 1 に近づきます。具体的な例としては、`x` を $2^{-24} + 2^{-47}$ とすると、`x` は、`1.0 + x` が、次に大きな数である $1 + 2^{-23}$ に丸められるような最小の単精度数になります。この場合、`log(1.0 + x)` はおよそ 2^{-23} になります。6 行目の式の分母は拡張精度で評価されるため、正確に計算され、`x` になります。したがって、`log1p(x)` はおよそ 2^{-23} を返し、これは正確な値のほぼ 2 倍になります。

これは実際に、少なくとも 1 つのコンパイラで発生します。前述のコードが、`-O` 最適化フラグを使用して x86 システム用の Sun WorkShop Compilers 4.2.1 Fortran 77 コンパイラでコンパイルされる場合、生成されたコードは説明のとおり `1.0 + x` を計算します。その結果、この関数は `log1p(1.0e-10)` にはゼロを、`log1p(5.97e-8)` には `1.19209E-07` を配布します。

定理 4 のアルゴリズムが正しく動作するには、式 `1.0 + x` が出現するたびに同じ方法で評価される必要があります。このアルゴリズムは、`1.0 + x` が、あるときは拡張倍精度に評価され、別のときには単精度または倍精度に評価されるというような場合にのみ、拡張ベースシステムで失敗する可能性があります。`log` は、Fortran の総称組み込み関数なので、コンパイラは完全に拡張精度で式 `1.0 + x` を評価し、同じ精度でその対数を計算できますが、明らかに、コンパイラがそうすることは想定できません。ユーザー定義の関数を使用した場合の類似例についても想定できます。この場合、関数が単精度の結果を返したとしても、コンパイラはまだ拡張精度で引数を保持できますが、既存の Fortran コンパイラでこれを実行するものは、存在すると

してもごくわずかです。そのため、`1.0 + x` を変数に割り当てることによって、これを常に同じように評価するよう試行するかもしれません。しかし、変数 `real` を宣言した場合でも、ある変数が出現しても、これに拡張精度でレジスタに保持されている値を代入し、別の変数にはメモリーに格納されている単精代変数のある出現の代わりに、拡張精度でレジスタに保持されている値を使用し、別の変数には単精度でメモリーに格納されている値を代入するようなコンパイラによって失敗することがあります。そのために、拡張精度形式に対応する型の変数を宣言する必要があります。標準の FORTRAN 77 にはこの方法が提供されておらず、Fortran 95 にはさまざまな形式を記述するための `SELECTED_REAL_KIND` メカニズムが用意されていますが、拡張精度で変数を宣言されるように拡張精度で式を評価する実装を明示的には規定していません。つまり、標準 Fortran では、式 `1.0 + x` が実証を無効にするような方法で評価することを確実に防ぐように、このプログラムを記述する移植可能な方法はありません。

拡張ベースシステムでは、それぞれのサブエクスプレッションが格納され、そのために同じ精度に丸められている場合でも、正しく動作しない可能性がある例がほかにもあります。原因は二重丸めです。デフォルトの精度モードでは、拡張ベースシステムは、最初にそれぞれの結果を拡張倍精度に丸めます。この結果がその後倍精度に格納されると、ふたたび丸められます。これら 2 回の丸めが組み合わさることにより、最初の結果を正しく倍精度に丸めるときに得られるものとは異なる値が生成されることがあります。これは、拡張倍精度に丸められたときの結果が「中間の場合」である場合、つまり、2 つの倍精度数のちょうど中間になるため、2 回目の丸めが round-ties-to-even 規則によって決定される場合に起こる可能性があります。この 2 回目の丸めが 1 回目と同じ方向になる場合、最終的な丸め誤差は最後の 1 位の桁の半分を超えてしまいます。しかし、この二重丸めは、倍精度計算にしか影響しません。2 つの p ビット数の和、差、積、または商、あるいは p ビット数の平方根が、最初 q ビットに丸められ、続いて p ビットに丸められるような場合、$q \geq 2p + 2$ であれば、結果は p ビットに 1 回だけ丸められた場合と同じになります。拡張倍精度は十分な広いため、単精度計算は二重丸めを受けることはありません。

正確な丸めを前提とするアルゴリズムの中には、二重丸めに対応しないものもあります。実際、正しい丸めを必要とせず、IEEE 754 に準拠していないさまざまなマシンで正しく機能するアルゴリズムでも、二重丸めで失敗することがあります。この中でもっとも便利なものは、Goldberg 氏の論説の定理 5 に述べられている、シミュレーションされた多倍精度演算を実行する、移植可能なアルゴリズムです。たとえば、定理 6 に述べられている、浮動小数点を上位部分と下位部分に分けるためのプロシージャーは、二重丸め演算では正しく動作しません。倍精度数 $2^{52} + 3 \times 2^{26} - 1$ を 2 つの部分に、それぞれ最大でも 26 ビットで分割してみます。それぞれの演算が正しく倍精度に丸められたら、上位部分は $2^{52} + 2^{27}$ で下位部分は $2^{26} - 1$ になりますが、それぞれの演算が最初に拡張倍精度に丸められ、続いて倍精度に丸められると、$2^{52} + 2^{28}$ の上位部分と $-2^{26} - 1$ の下位部分が生成されます。下位部分の数は

27 ビットを必要とするため、その 2 乗は倍精度では正確に計算できません。この数の 2 乗を拡張倍精度で計算することは可能ですが、その結果のアルゴリズムは単精度/倍精度システムには移植できなくなります。また、多倍精度乗算アルゴリズムの後半の手順では、すべての部分積が倍精度で計算されていることを前提としています。倍精度と拡張倍精度の変数の組み合わせを正しく処理するには、非常に実装コストがかかります。

同様に、倍精度数の配列として表される多倍精度数を追加するための移植可能なアルゴリズムは、二重丸め演算で失敗することがあります。これらのアルゴリズムは通常、カハンの総和公式に類似した手法に依存しています。総和公式の非公式な説明として、Goldberg 氏の論説の加算でのエラーのセクションでは次のように提案しています。s と y が |s| ≥ |y| である浮動小数点変数であり、次のように計算すれば、

```
t = s + y;
e = (s - t) + y;
```

ほとんどの演算で、e が t の計算で生じた丸め誤差を正確に回復させます。ただし、この手法は二重丸め演算では機能しません。$s = 2^{52} + 1$、$y = 1/2 - 2^{-54}$ の場合、s + y が最初に拡張倍精度で $2^{52} + 3/2$ に丸められ、この値がさらに round-ties-to-even 規則によって倍精度で $2^{52} + 2$ に丸められます。したがって、t の計算での最終的な丸め誤差は $1/2 + 2^{-54}$ になり、これは倍精度では正しく表現できず、このため上記の式では正しく計算できません。この場合も、拡張倍精度で和を計算することにより丸め誤差を回復することは可能ですが、その場合、プログラムは最終出力を引き下げて倍精度に戻す作業が必要になり、二重丸めがこのプロセスにも影響することがあります。このような理由により、これらの方法で多倍精度演算をシミュレーションする移植可能なプログラムは、さまざまなマシンで正しく効率的に動作しますが、拡張ベースシステムでは公表されているようには動作しません。

最後に、一見正確な丸めに依存するようなアルゴリズムでも、実際は二重の丸めで正しく動作するものがあります。このような場合では、実装ではなく、アルゴリズムが公表どおりに動作するかどうかを検証するために、二重丸めを実行する必要があります。これを説明するため、定理 7 を別の例で証明します。

D.1.2.1 定理 7

m と n が IEEE 754 の倍精度で表現可能な $|m| < 2^{52}$ の整数であり、n が特殊な式 $n = 2^i + 2^j$ である場合、両方の浮動小数点演算が倍精度に正しく丸められるか、最初に拡張倍精度に丸められてから倍精度に丸められるとすると、$(m \# n) \otimes n = m$ になる。

注記 - この定理では、# と ⊗ はそれぞれ計算された除法と計算された乗法を表します。

D.1.2.2 証明

損失なしの $m > 0$ と仮定します。$q = m \# n$ とします。2 の累乗でスケールする場合、$2^{52} \le m < 2^{53}$ で q についても同様であり、その結果、m と q の両方が、その最下位ビットが 1 位の桁を占める整数になる (つまり、$\mathrm{ulp}(m) = \mathrm{ulp}(q) = 1$) 同等の設定を考えられます。スケール前に $m < 2^{52}$ と仮定したため、スケール後の m は偶数の整数になります。また、m と q のスケール後の値は $m/2 < q < 2m$ を満たすため、対応する n の値は、m か q のどちらが大きいかに応じて次の 2 つの式のいずれかになります。$q < m$ の場合、明らかに $1 < n < 2$ であり、n は 2 つの 2 の累乗の和であるため、特定の k については $n = 1 + 2^{-k}$ であり、同様に $q > m$ の場合は $1/2 < n < 1$ であるため、$n = 1/2 + 2^{-(k+1)}$ になります。n は 2 つの 2 の累乗の和であるため、1 にもっとも近い値である n は、$n = 1 + 2^{-52}$ です。$m/(1 + 2^{-52})$ は、m より小さい 2 番目に小さな倍精度数より大きくないため、$q = m$ にはなりません)。

q を計算するための丸め誤差を e と仮定し、その結果 $q = m/n + e$ となり、計算された値 $q \otimes n$ は $m + ne$ の (1 回または 2 回) 丸められた値になるとします。まず、それぞれの浮動小数点演算が倍精度に正しく丸められる場合を考えてみます。この場合、$|e| < 1/2$ になります。n の式が $1/2 + 2^{-(k+1)}$ である場合、$ne = nq - m$ は、$2^{-(k+1)}$ の整数の倍数であり、$|ne| < 1/4 + 2^{-(k+1)}$ になります。これは $|ne| \le 1/4$ であることを意味します。m と次に大きな表現可能な数の差は 1 であり、m と次に小さな表現可能な数の差は、$m > 2^{52}$ の場合は 1、$m = 2^{52}$ の場合は $1/2$ であることを思い出してください。このため、$|ne| \le 1/4$ の場合、$m + ne$ は m に丸められます。($m = 2^{52}$ で $ne = -1/4$ の場合でも、積は、round-ties-to-even 規則により、m に丸められます)。同様に、n の式が $1 + 2^{-k}$ である場合、ne は、2^{-k} の整数倍であり、$|ne| < 1/2 + 2^{-(k+1)}$ です。これは $|ne| \le 1/2$ を意味します。m は厳密には q より大きいため、この場合、$m = 2^{52}$ にはなりません。そのため、m と隣接した表現可能な数値との差は ±1 です。したがって、$|ne| \le 1/2$ の場合も、$m + ne$ は m に丸められます。($|ne| = 1/2$ の場合でも、m が偶数であるため、積は round-ties-to-even 規則で m に丸められます)。正確に丸められる演算の証明はこれで終了です。

二重丸め演算でも、q は (実際に 2 回丸められた場合でも) 正確に丸められた商になることがあるため、上記と同じく $|e| < 1/2$ です。この場合、$q \otimes n$ が 2 回丸められるという事実を考慮すると、前のパラグラフの引数に訴えることができます。このことを説明するために、IEEE 標準では拡張倍精度形式が少なくとも 64 の有効ビットを含むことを規定し、そのため、$m \pm 1/2$ と $m \pm 1/4$ の数は拡張倍精度で正確に表現できることに注意してください。したがって、n の式が $1/2 + 2^{-(k+1)}$ であり、その結果 $|ne| \le 1/4$ になる場合、$m + ne$ を拡張倍精度に丸めると、m と最大でも $1/4$ の差がある結果が生成され、上記のように、この値は倍精度で m に丸められます。同様に、n の式が $1 + 2^{-k}$ であり、その結果 $|ne| \le 1/2$ になる場合、$m + ne$ を拡

張倍精度に丸めた結果は、m と最大 1/2 の差がある結果が生成され、その値は倍精度では m に丸められます。この場合 $m > 2^{52}$ であることに注意してください。

最後に、二重丸めのために q が正しく丸められた商になっていない場合を考えてみます。このような場合、最悪の場合には $|e| < 1/2 + 2^{-(d+1)}$ となり、ここでは d は拡張倍精度形式での余分なビット数です。既存の拡張ベースシステムはすべて、ちょうど 64 個の有効ビットを持つ拡張倍精度形式をサポートします。この形式では $d = 64 - 53 = 11$ になります。二重丸めは、2 回目の丸めが round-ties-to-even 規則で決定された場合にかぎって正しくない丸め結果を生成するため、q は偶数の整数である必要があります。したがって、n の式が $1/2 + 2^{-(k+1)}$ である場合は、$ne = nq - m$ は 2^{-k} の整数倍になり、$|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}$ になります。

$k \leq d$ の場合、これは $|ne| \leq 1/4$ を意味します。$k > d$ の場合、$|ne| \leq 1/4 + 2^{-(d+2)}$ になります。どちらの場合でも、積の最初の丸めにより、m と最大 1/4 の差のある結果が得られ、前に示した引数によって、2 回目の丸めで m に丸めます。同様に、n の式が $1 + 2^{-k}$ である場合、ne は $2^{-(k-1)}$ の整数倍になり、$|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}$ になります。

$k \leq d$ の場合、これは $|ne| \leq 1/2$ を意味します。$k > d$ の場合、$|ne| \leq 1/4 + 2^{-(d+1)}$ になります。どちらの場合も、積の最初の丸めにより、m と最大 1/2 の差のある結果が得られ、再度、前に示した引数によって、2 回目の丸めで m に丸めます。

前述の証明では、商が二重丸めを引き起こす場合のみ積も二重丸めを引き起こし、その場合でも正しい結果に丸められることを示しています。この証明では、二重丸めの可能性を含めるように推論を拡張することは、浮動小数点演算が 2 つしかないプログラムでも困難になることも示しています。さらに複雑なプログラムでは、二重の丸めの影響について体系的に説明することは不可能であり、倍精度と拡張倍精度の演算の一般的な組み合わせは言うまでもありません。

D.1.3 拡張精度におけるプログラミング言語のサポート

前の例では、拡張精度自体が有害であることを示しているわけではありません。プログラマが拡張精度の使用を選択できる場合には、多くのプログラムが拡張精度のメリットを得ることができます。しかし、現在のプログラミング言語では、プログラマが拡張精度をいつどのような方法で使用するかを指定する十分な手段が用意されていません。どのようなサポートが必要かを示すため、拡張精度の使用を管理できる方法について検討してみます。

名目上の作業精度として倍精度を使用する移植可能なプログラムでは、より広い精度の使用を制御する 5 つの方法があります。

1. 拡張ベースシステムでは、可能なかぎり拡張精度を使用して最速のコードを生成するようコンパイルする。ほとんどの数値ソフトウェアでは、「計算機イプシロン」が範囲を指定する各演算の相対誤差以上の演算は必要とされません。入力データは入力時に丸められていると想定され、その結果が格納されるときにも同様に丸められるため、メモリー内のデータが倍精度で格納されると、計算機イプシロンがその精度の最大の相対丸め誤差と見なされます。したがって、拡張精度の中間結果の計算では、より正確な結果を生成することもありますが、拡張精度は必須ではありません。この場合、感知できるほどプログラムの速度が低下しない場合にのみコンパイラで拡張精度を使用し、それ以外の場合は倍精度を使用することをお勧めします。
2. かなり高速で十分に広い場合は倍精度よりも広い形式を使用し、それ以外の場合はほかの形式を使用する。一部の計算では、拡張精度を使用できればより簡単に実行できるものもありますが、倍精度でも、いくらか取り組むだけで実行できるようになります。倍精度数のベクトルのユークリッドノルムの計算を考えてみます。要素の 2 乗を計算し、IEEE 754 拡張倍精度形式でその広い指数範囲を使用してそれらの和を累積すると、実際の長さのベクトルに対して早まったアンダーフローまたはオーバーフローを簡単に回避できます。拡張ベースシステムでは、これがノルムを計算するもっとも速い方法です。単精度/倍精度システムでは、拡張倍精度形式がサポートされている場合はソフトウェア内でエミュレートする必要がありますが、これは例外フラグをテストしてアンダーフローまたはオーバーフローが発生していないか確認し、発生した場合は明示的なスケールで計算を繰り返すため、単に倍精度を使用する場合より大幅に速度が低下します。言語では、拡張精度のこのような使用をサポートするため、使用する方法をプログラムが選択できるように適度に高速で利用可能なうちもっとも広い形式を示すとともに、その形式が十分広いこと (たとえば倍精度よりも広い範囲であること) をプログラムが検証できるようにする、それぞれの形式の精度と範囲を示す環境パラメータを提供する必要があります。
3. ソフトウェアでエミュレートする必要がある場合でも、倍精度より広い形式を使用する。ユークリッドノルムの例よりも複雑なプログラムの場合、プログラマは、2 つのバージョンのプログラムを作成する必要性を避け、代わりに、速度が遅くても拡張精度に頼ることがあります。この場合も、言語には、利用可能なうちもっとも範囲と精度の広い形式をプログラムが判断できるように、環境パラメータが用意されている必要があります。
4. 倍精度より広い精度を使用しない (範囲が拡張された場合でも、結果は倍精度形式の精度に正しく丸められます)。前述のいくつかの例のように、正しく丸められる倍精度演算に依存するように非常に簡単に作成されるプログラムの場合、言語は、中間結果が倍精度より広い指数範囲のレジスタで計算できるとしても、拡張精度を使用しないように指示する方法をプログラマに提供する必要があります。この方法で計算された中間結果でも、メモ

リーに格納されるときにアンダーフローする場合は二重丸めを引き起こすことがあります。算術演算の結果が最初に 53 の有効ビットに丸められ、次に非正規化する必要があるときにそれより少ない有効ビットにふたたび丸められる場合、最終結果は、非正規化数に一度だけ丸めた場合に取得できる結果とは異なる場合があります。当然、このような二重丸めが、実際のプログラムに悪影響を及ぼすことはほとんどありません。

5. 倍精度形式の精度と範囲の両方に正しく結果を丸める。この倍精度の厳密な強制は、倍精度形式の範囲と精度の両方の限度の近くで、数値ソフトウェアか演算自体をテストするプログラムにもっとも役立ちます。このような慎重を要するテストプログラムは、移植可能な方法で作成することは通常困難であり、特定の形式に結果を強制的に丸めるためにダミーサブルーチンやほかのトリックを使用する必要があるときにさらに難しく、エラーが起こりやすくなります。したがって、拡張ベースシステムを使用して、すべての IEEE 754 実装に移植可能である必要のある堅固なソフトウェアを開発するプログラマはすぐに、膨大な労力を必要とせず、単精度/倍精度システムの演算をエミュレートできることを高く評価するようになります。

現在、これら 5 つのオプションすべてをサポートしている言語はありません。実際、プログラマが拡張精度の使用を制御できる機能を提供した言語はほとんどありません。著名な例外の 1 つが、C 言語のメジャーリビジョンである ISO/IEC 9899:1999 プログラミング言語 - C 標準です。

C99 標準では、通常その型に関連付けられているよりも広い形式で式を評価する実装が可能ですが、式を評価する 3 つの方法のうち、どれか 1 つを使用することを推奨しています。3 つの推奨方法は、式がより広い形式に「変換」される程度によって区別され、実装では、プリプロセッサマクロ `FLT_EVAL_METHOD` を定義することによって使用する方法を特定することが推奨されています。`FLT_EVAL_METHOD` が 0 の場合、それぞれの式は、その型に対応する形式で評価されます。`FLT_EVAL_METHOD` が 1 の場合、`float` の式は、`double` に対応する形式に拡張されます。`FLT_EVAL_METHOD` が 2 の場合、`float` および `double` の式は `long double` に対応する形式に拡張されます。(実装では、式の評価方法が確定できないことを示すために `FLT_EVAL_METHOD` を −1 に設定することができます)。C99 標準では、`<math.h>` ヘッダーファイルで型 `float_t` と `double_t` を定義することも求められます。`float` と `double` は少なくとも同じ広さであり、それぞれ `float` および `double` の式を評価するために使用される型に一致するようになっています。たとえば、`FLT_EVAL_METHOD` が 2 の場合、`float_t` と `double_t` はどちらも `long double` です。最後に、C99 標準では、`<float.h>` ヘッダーファイルで、それぞれの浮動小数点型に対応する形式の範囲と精度を指定するプリプロセッサマクロを定義することが求められています。

C99 標準で求められる、または推奨されている機能の組み合わせは、上記の 5 つのオプションのすべてではなく一部をサポートしています。たとえば、実装が、long double 型を拡張倍精度形式にマッピングし、FLT_EVAL_METHOD を 2 と定義している場合、プログラマは、拡張精度が比較的高速であり、そのためユークリッドノルムの例のようなプログラムが、型 long double (または double_t) の中間変数を使用できると想定できます。一方、この同じ実装は、無名の式がメモリーに格納されるときでも (たとえば、コンパイラが浮動小数点レジスタをあふれさせる必要があるとき)、この式を拡張精度で保持する必要があり、その結果は、double に宣言された変数に割り当てた式の結果を格納し、倍精度に変換する必要があります。これは、式のレジスタに保持できた場合でも同様です。したがって、double と double_t のどちらの型も、現在の拡張ベースのハードウェアで最速のコードを生成するようにコンパイルすることはできません。

同様に、このセクションの例が示す問題は、すべてではなく、一部を C99 標準が提供する方法で解決できます。C99 標準バージョンの log1p 関数は、式 1.0 + x が (任意の型の) 変数に代入され、その変数が常に使用される場合に正しく動作することが保証されています。しかし、倍精度数を上位部分と下位部分に分割するための移植可能で効率的な C99 標準プログラムは、さらに困難です。double の式が倍精度に正しく丸められることを保証できない場合に、どうすれば正しい位置で分割し、二重丸めを回避できるでしょうか。1 つの解決方法として、double_t 型を使用して、単精度/倍精度システムでは倍精度に、拡張ベースシステムでは拡張精度に分割して、どちらの場合でも演算が正しく丸められるようできます。定理 14 では、ベースとなる演算の精度がわかっているならば、どのビット位置でも分割できるとしており、この情報は FLT_EVAL_METHOD と環境パラメータマクロで取得できます。

次のフラグメントは可能な実装例の 1 つを示しています。

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

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif

...
    double   x, xh, xl;
    double_t m;

    m = scalbn(1.0, PWR2) + 1.0;  // 2**PWR2 + 1
    xh = (m * x) - ((m * x) - x);
    xl = x - xh;
```

この解決方法を見つけるには、double の式が拡張精度で評価されることがあること、その後の二重丸め問題によってアルゴリズムが誤って動作する可能性があること、および定理 14 に従って拡張精度を代用できるということを認識している必要があります。よりわかりやすい解決方法として、それぞれの式が正しく倍精度に丸められるように指定することもできます。拡張ベースシステムでは、これには丸め精度モードの変更が必要なだけですが、C99 標準には、これを行う移植可能な方法は用意されていません。浮動小数点をサポートするために C90 標準に加える変更点を指定した作業文書である、浮動小数点 C 編集の初期ドラフトでは、丸め精度モードを使用したシステムでの実装で、丸め方向を取得し設定する fegetround および fesetround 関数に類似した、丸め精度を取得し設定する fegetprec および fesetprec 関数を提供することを推奨していました。この推奨は、C99 標準に変更が加えられる前に削除されました。

また、整数演算機能が異なるシステム間の移植可能性をサポートする C99 標準のアプローチは、異なる浮動小数点アーキテクチャーをサポートするより適切な方法を提示しています。C99 標準のそれぞれの実装には、実装がサポートする整数型を定義する <stdint.h> ヘッダーファイルが用意されており、そのサイズと効率性に応じて名前が付けられています。たとえば、int32_t は、ちょうど 32 ビット幅の整数型であり、int_fast16_t は、16 ビット幅以上の実装で最速の整数型であり、intmax_t はサポートされるもっとも広い整数型です。浮動小数点の型についても、類似のスキームを想像できます。たとえば、float53_t は、精度がちょうど 53 ビットだが、範囲がこれを超える可能性のある浮動小数点、float_fast24_t は、精度が 24 ビット以上である実装の最速の型、floatmax_t は、サポートするもっとも広い適度に高速な型の名前になります。高速型では、レジスタがあふれた結果として名前付きの変数の値が変更されてはならないという制約がありますが、拡張ベースシステム上のコンパイラでは、可能なかぎり最速のコードを生成できます。幅がちょうどの型では、拡張ベースシステム上のコンパイラは、前述の制限を条件として、丸め精度モードを指定の精度に丸めるように設定でき、より広い範囲を許可します。double_t は、IEEE 754 の倍精度形式の精度と範囲の両方を持つ型の名前として指定でき、厳密な倍精度評価を実現します。このようなスキームを、適宜名前を付けられた環境パラメータマクロとともに使用すると、前述の 5 つのオプションすべてを簡単にサポートし、プログラムで必要とされる浮動小数点演算セマンティックスをプログラマが簡単かつ明確に指定できるようになります。

拡張精度の言語サポートはそれほど複雑ででしょうか。単精度/倍精度システムでは、上記の 5 つのオプションのうち 4 つが該当し、高速な方と正確な幅の型を区別する必要はありません。ただし、拡張ベースシステムでは難しい選択を迫られます。拡張ベースシステムでは、純粋な倍精度計算、純粋な拡張精度計算のどちらも、2 つを組み合わせた場合ほど効率的にはサポートされず、プログラムに応じて異なる組み合わせが必要になります。また、拡張精度をいつ使用するかの選択は、コンパイラライターには任せないでください。彼らはベンチマークの結果、

浮動小数点演算は「本質的に正確でなく」、整数演算には適しておらず、その予測能力もないと判断される傾向があり、また、数値アナリストが公然と語ることもあります。この選択はプログラマに委ねられる必要があり、プログラマは、自身の選択を表すことができる言語が必要になります。

D.1.4 結論

前述の見解は、拡張ベースシステムを軽んじるためのものではなく、いくつかの誤った議論を明らかにすることを目的としており、その議論の最たるものは、すべての IEEE 754 システムが同じプログラムで同一の結果をもたらす必要があるというものでした。これまで拡張ベースシステムと単精度/倍精度システムとの違いに焦点を当てていましたが、これらの各系統内のシステム間にはさらに相違点があります。たとえば、一部の単精度/倍精度システムには、2 つの数を掛け合わせ、3 番目の数を加算し、最後に一度だけ丸めるという単一の命令があります。この演算は、融合型積和演算と呼ばれ、同じプログラムが単精度/倍精度システムごとに異なる結果を生成する場合があり、拡張精度のように、プログラムが使用されるかどうか、またいつ使用されるか応じて、同じシステム上でも同じプログラムが異なる結果を生成する場合もあります。融合型積和演算は、移植できない方法で分割せずに多倍精度乗算を実行する場合に使用できますが、定理 6 の分割プロセスが失敗する場合もあります。IEEE 標準でこのような演算を予期していなかったとしても、中間の積はユーザーの制御の及ばない、正確に保持できる十分な広さを持つ「デスティネーション」に配布され、最終的な和はその単精度または倍精度のデスティネーションに適合するように正しく丸められます。

それでもなお、IEEE 754 が一定のプログラムが配布する必要のある結果を正確に規定しているという考えは、非常に有用です。多くのプログラマは、プログラムをコンパイルするコンパイラやそれを実行するコンピュータに関係なく、プログラムの動作を理解し、それが正しく動作することを証明できると思い込みがちです。この見解をサポートすることは、コンピュータシステムやプログラミング言語の設計者にとっては、多くの点で意味のある目標になります。しかし、浮動小数点演算に関して言えば、この目標を達成することは事実上不可能です。IEEE 標準の作成者もこのことを理解しており、この実現を試みていません。その結果、コンピュータ業界のほぼ全体が IEEE 754 標準に準拠しているにもかかわらず、移植可能なソフトウェアのプログラマは、予測不可能な浮動小数点演算に取り組み続けなければなりません。

プログラマが IEEE 754 の機能を利用するのであれば、浮動小数点演算を予測可能にするプログラミング言語が必要になります。C99 標準では、プログラマは、`FLT_EVAL_METHOD` ごとに 1 つずつ、プログラムのバージョンを作成する必要がありますが、それなりに予測可能性が向上しています。将来の言語では、IEEE 754 セマンティックスに依存している範囲を明らかにする構文によって、単一のプログラムを作成できるかどうかはまだ明らかではありません。既存の拡

張ベースシステムは、所定のシステムでどのように演算を実行するかについては、プログラマよりもコンパイラとハードウェアのほうがより認識できると想定させることによって、その展望を危うくしています。その想定が 2 番目の誤りであり、計算結果に求められる精度は、結果を生成したマシンではなく、そこから引き出される結論にのみ依存し、これらの結論が何であるかを知ることができるのは、プログラマ、コンパイラ、ハードウェアのうちせいぜいプログラマだけなのです。

♦ ♦ ♦ 付 録 E

標準規格への準拠

Oracle Solaris Studio コンパイラ製品は、Solaris 10 オペレーティング環境内のヘッダーファイルやライブラリとともに、System V Interface Definition Edition 3 (SVID)、X/Open、ANSI C (C90)、POSIX.1-2001 (SUSv3)、ISO C (C99) を含む複数の標準規格をサポートしています。(詳細は、*standards*(5) を参照してください。) これらの標準規格の中には、特定の点で実装の相違が認められるものがあります。場合によっては、これらの標準の仕様が矛盾することもあります。数学ライブラリの場合、これらの相違点や矛盾は主に、特殊なケースおよび例外に関連しています。この付録では、`libm` に含まれる関数の動作を示し、C プログラムがどのような条件でそれぞれの標準規格に準拠するよう動作するかについて説明します。この付録の最後のセクションでは、Sun Studio C および Fortran 言語製品の LIA-1 への準拠について説明します。

E.1 `libm` の特殊なケース

表E-1「特殊なケースと `libm` 関数」は、上で挙げた標準規格の 2 つ以上が `libm` 内の関数に対して矛盾する動作を指定するすべてのケースを示しています。C プログラムがどの動作に準拠するかは、そのプログラムをコンパイルし、リンクするときに使用されるコンパイラフラグによって異なります。考えられる動作としては、浮動小数点例外を発生させる、発生した特殊なケースに関する情報と返される値を指定してユーザー提供の関数 `matherr` を呼び出す (*matherr*(3M) を参照)、メッセージを標準エラーファイルに出力する、大域変数を設定する `errno` (*intro*(2) および *perror*(3C) を参照) などがあります。

表E-1「特殊なケースと `libm` 関数」の最初の列は、特殊なケースを定義しています。2 番目の列は、`errno` に設定される値を示しています (設定される場合)。`errno` の可能性のある値は、`<errno.h>` で定義されています。数学ライブラリで使用される値は、ドメインエラーを示す `EDOM` と、範囲エラーを示す `ERANGE` の 2 つだけです。2 番目の列に `EDOM` と `ERANGE` の両方が示されている場合、`errno` に設定される値は、この後に説明するとおり標準規格で決定され、4 番目と 5 番目の列に示されます。3 番目の列は、エラーメッセージが出力される場合に示され

るエラーコードを示しています。4 番目、5 番目、および 6 番目の列は、さまざまな標準規格の定義に従って名目上返される関数値を示しています。場合によっては、ユーザー提供の matherr ルーチンがこれらの値をオーバーライドし、別の戻り値を提供することがあります。

これらの特殊なケースへの具体的な対応は、次のようにプログラムがリンクされるときに指定されたコンパイラフラグによって決定されます。-xlibmieee または -xc99=lib のどちらかが指定されている場合は、表E-1「特殊なケースと libm 関数」の特殊なケースが発生すると、いずれかの該当する浮動小数点例外が発生し、表の 6 番目の列に示されている関数値が返されます。

-xlibmieee と -xc99=lib のどちらも使用されていない場合、動作は、プログラムがリンクされるときに指定された言語準拠フラグによって異なります。

-Xa フラグを指定すると、X/Open 準拠が選択されます。表にあるいずれかの特殊なケースが発生すると、いずれかの該当する浮動小数点例外が発生し、errno が設定され、表の 5 番目の列に示されている関数値が返されます。ユーザー定義の matherr ルーチンが提供されている場合、動作は定義されていません。-Xa は、ほかの言語準拠フラグが指定されていないときのデフォルトです。

-Xc フラグを指定すると、厳格な C90 準拠が選択されます。特殊なケースが発生すると、いずれかの該当する浮動小数点例外が発生し、errno が設定され、表の 5 番目の列に示されている関数値が返されます。この場合、matherr は呼び出されません。

最後に、-Xs または -Xt フラグのどちらかを指定すると、SVID 準拠が選択されます。特殊なケースが発生すると、いずれかの該当する浮動小数点例外が発生し、matherr が呼び出されます。matherr が 0 を返した場合は、errno が設定され、エラーメッセージが出力されます。表の 4 番目の列に示されている関数値は、それが matherr によってオーバーライドされないかぎり返されます。

-xc99、-Xa、-Xc、-Xs、および -Xt フラグの詳細は、*cc*(1) のマニュアルページおよび『Oracle Solaris Studio 12.4: C ユーザーガイド 』を参照してください。

表 E-1 特殊なケースと libm 関数

関数	errno	エラーメッセージ	SVID	X/Open、C90	IEEE、C99、SUSv3
acos(|x|>1)	EDOM	DOMAIN	0.0	0.0	NaN
acosh(x<1)	EDOM	DOMAIN	NaN	NaN	NaN
asin(|x|>1)	EDOM	DOMAIN	0.0	0.0	NaN

関数	errno	エラーメッセージ	SVID	X/Open、C90	IEEE、C99、SUSv3
atan2(+/-0,+/-0)	EDOM	DOMAIN	0.0	0.0	+/-0.0,+/-pi
atanh(|x|>1)	EDOM	DOMAIN	NaN	NaN	NaN
atanh(+/-1)	EDOM/ERANGE	SING	+/-HUGE1 (EDOM)	+/-HUGE_VAL2 (ERANGE)	+/-infinity
cosh overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
exp overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
exp underflow	ERANGE	-	0.0	0.0	0.0
fmod(x,0)	EDOM	DOMAIN	x	NaN	NaN
gamma(0 or -integer)	EDOM	SING	HUGE	HUGE_VAL	infinity
gamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
hypot overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
j0(X_TLOSS3<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
j1(X_TLOSS<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
jn(n,X_TLOSS<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
ldexp overflow	ERANGE	-	+-infinity	+/-infinity	+/-infinity
ldexp underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
lgamma(0 or -integer)	EDOM	SING	HUGE	HUGE_VAL	infinity
lgamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
log(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log1p(-1)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log1p(x<-1)	EDOM	DOMAIN	NaN	NaN	NaN
logb(0)	EDOM	-	-HUGE_VAL	-HUGE_VAL	-infinity
nextafter overflow	ERANGE	-	+-HUGE_VAL	+/-HUGE_VAL	+/-infinity
pow(0,0)	EDOM	DOMAIN	0.0	1.0 (no error)	1.0 (no error)
pow(NaN,0)	EDOM	DOMAIN	NaN	NaN	1.0 (no error)
pow(0,x<0)	EDOM	DOMAIN	0.0	-HUGE_VAL	+/-infinity
pow(x<0, non-integer)	EDOM	DOMAIN	0.0	NaN	NaN
pow overflow	ERANGE	-	+/-HUGE	+/-HUGE_VAL	+/-infinity
pow underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0

関数	errno	エラーメッセージ	SVID	X/Open、C90	IEEE、C99、SUSv3
remainder(x,0) or remainder(inf,y)	EDOM	DOMAIN	NaN	NaN	NaN
scalb overflow	ERANGE	-	+-HUGE_VAL	+/-HUGE_VAL	+/-infinity
scalb underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
scalb(0,+inf) or scalb(inf,-inf)	EDOM/ERANGE	-	NaN (ERANGE)	NaN (EDOM)	NaN
scalb(|x|>0,+inf)	ERANGE	-	+-infinity	+/-infinity (no error)	+/-infinity (エラーなし)
scalb(|x|<inf, -inf)	ERANGE	-	+/-0.0	+/-0.0 (no error)	+/-0.0 (エラーなし)
sinh overflow	ERANGE	-	+/-HUGE	+/-HUGE_VAL	+/-infinity
sqrt(x<0)	EDOM	DOMAIN	0.0	NaN	NaN
y0(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
y0(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y0(X_TLOSS<x<inf)	ERANGE	TLOSS	0.0	0.0	correct answer
y1(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
y1(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y1(X_TLOSS<x<inf)	ERANGE	TLOSS	0.0	0.0	correct answer
yn(n,0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
yn(n,x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
yn(n,X_TLOSS<x< inf)	ERANGE	TLOSS	0.0	0.0	correct answer

注:

1. HUGE は、<math.h> で定義されています。SVID では、HUGE が MAXFLOAT (約 3.4e+38) に等しくなる必要があります。
2. HUGE_VAL は、<math.h> に含まれている <iso/math_iso.h> で定義されています。HUGE_VAL は、無限大に評価されます。
3. X_TLOSS は、<values.h> で定義されています。

E.1.1 標準規格への準拠に影響を及ぼすその他のコンパイラフラグ

上に挙げたコンパイラフラグは、表E-1「特殊なケースと libm 関数」に示されている特殊なケースを処理する場合に、いくつかの標準規格のうちのどれに従うかを直接選択します。その他のコンパイラフラグには、前に説明した動作をプログラムが準拠するかどうかに間接的に影響するものもあります。

まず、-xlibmil と -xlibmopt はどちらも libm 内の一部の関数をより高速な実装に置き換えます。これらの高速な実装は、SVID、X/Open、または C90 に準拠していません。また、errno の設定や matherr の呼び出しも行いません。ただし、必要に応じて浮動小数点例外を発生させたり、IEEE 754 または C99、あるいはその両方で指定されている結果を生成したりします。-xvector フラグにも同じことが当てはまります。このフラグによって、コンパイラが標準数学関数の呼び出しをベクトル数学関数の呼び出しに変換することがあるためです。

2 番目に、-xbuiltin フラグにより、コンパイラは <math.h> で定義されている標準数学関数を組み込み関数として扱い、パフォーマンスを向上させるためにインラインコードに置き換えることができます。置き換えたコードは、SVID、X/Open、C90、または C99 に準拠していない場合があります。errno を設定したり、matherr を呼び出したり、浮動小数点例外を発生させたりする必要はありません。

3 番目に、C プリプロセッサトークン __MATHERR_ERRNO_DONTCARE が定義されていると、<math.h> 内のいくつかの #pragma ディレクティブがコンパイルされます。これらのディレクティブは、標準数学関数に副次的影響がないと仮定するようコンパイラに指示します。この仮定の下では、数学関数の呼び出し、errno などの大域データへの参照、ユーザー提供の matherr ルーチンで変更されている可能性のあるデータの順序がコンパイラによって変更され、前述の予測される動作に反する可能性があります。たとえば、次のコードフラグメントについて考えてみます。

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

...
errno = 0;
x = acos(2.0);
if (errno) {
    printf(“error\n”);
}
```

__MATHERR_ERRNO_DONTCARE を定義してこのコードをコンパイルすると、コンパイラは errno が acos の呼び出しによって変更されないと仮定し、それに応じてコードを変換するため、printf の呼び出しを完全に削除する可能性があります。

-fast マクロフラグには、フラグ -xbuiltin、-xlibmil、-xlibmopt、および -D__MATHERR_ERRNO_DONTCARE が含まれることに注意してください。

最後に、libm 内のすべての数学関数が必要に応じて浮動小数点例外を発生させるため、通常、これらの例外に対するトラップを有効にしてプログラムを実行すると、前述の標準規格で指定されているものとは異なる動作が実行されます。そのため、-ftrap コンパイラフラグも標準規格への準拠に影響を及ぼす場合があります。

E.1.2 C99 への準拠に関するその他の注意事項

C99 では、表E-1「特殊なケースと libm 関数」に示すような特殊なケースを実装によって処理できると想定される方法が 2 つ規定されています。1 つの実装は、値 MATH_ERRNO (1) または MATH_ERREXCEPT (2)、あるいはこのビット単位の「論理和」を持つ整数式を評価する識別子 math_errhandling を定義することによって、2 つの方法のどちらをサポートするかを示します。(これらの値は、<math.h> で定義されています) 式 (math_errhandling & MATH_ERRNO) が 0 以外である場合、実装では errno を EDOM に設定することによって、関数の引数がその数学的ドメインの外側に存在するケースを処理し、errno を ERANGE に設定することによって、関数の結果値がアンダーフローする、オーバーフローする、または正確に無限大に等しくなるケースを処理します。式 (math_errhandling & MATH_ERREXCEPT) が 0 以外である場合、実装では無効な演算例外を発生させることによって、関数の引数がその数学的ドメインの外側に存在するケースを処理し、アンダーフロー、オーバーフロー、またはゼロ除算の例外を発生させることによって、それぞれ、関数の結果値がアンダーフローする、オーバーフローする、または正確に無限大に等しくなるケースを処理します。

Oracle Solaris では、<math.h> は、math_errhandling を MATH_ERREXCEPT として定義します。表E-1「特殊なケースと libm 関数」に示す関数は、そこに示す特殊なケースに対して別のアクションを実行する可能性がありますが、float および long double 関数、複素関数、C99 で指定されているその他の関数を含め、すべての libm 関数が浮動小数点例外を発生させることによって特殊なケースに対応します。これは、すべての C99 関数対して一貫してサポートされている特殊ケースを処理するための唯一の方法です。

最後に、C99 または SUSv3 のどちらの場合でも、Oracle Solaris のデフォルトとは異なる動作が要求される 3 つの関数が存在することに注意してください。これらの相違点を次の表

にまとめます。この表には、各関数の double バージョンのみを示していますが、これらの相違点は float および long double バージョンにも同様に適用されます。いずれのケースでも、プログラムが -xc99=lib でリンクされている場合は SUSv3 の仕様に従い、それ以外の場合は Solaris のデフォルトに従います。

表 E-2 Solaris と C99/SUSv3 の相違点

関数	Solaris の動作	C99/SUSv3 の動作
pow	pow(1.0, +/-inf) は NaN を返す	pow(1.0, +/-inf) は 1 を返す
	pow(-1.0, +/-inf) は NaN を返す	pow(-1.0, +/-inf) は 1 を返す
	pow(1.0, NaN) は NaN を返す	pow(1.0, NaN) は 1 を返す
logb	logb(subnormal) は Emin を返す	x issubnormal のときは logb(x) = ilogb(x)
ilogb	ilogb(+/-0)、ilogb(+/-inf)、	ilogb(+/-0)、ilogb(+/-inf)、
	ilogb(NaN) は例外を発生させない	ilogb(NaN) は無効な演算を発生させる

E.2 LIA-1 への準拠

このセクションでは、LIA-1 は「ISO/IEC 10967-1:1994 Information Technology - Language Independent Arithmetic - Part 1: Integer and floating-point arithmetic」を指します。

Sun Studio コンパイラのリリースに含まれている C および Fortran 95 コンパイラ (cc および f95) は、次の点で LIA-1 に準拠しています (段落の文字は LIA-1 のセクション 8 の文字に対応しています)。

E.2.1 a.データ型 (LIA 5.1):

LIA-1 準拠のデータ型は、C の int と Fortran の INTEGER です。その他のデータ型の準拠もありますが、ここでは規定されていません。特定の言語に対するそれ以上の仕様は、管轄している言語標準規格の組織から LIA-1 への言語バインディング待ちです。

E.2.2 b.パラメータ (LIA 5.1):

```
#include <values.h> /* defines MAXINT */
#define TRUE 1
```

```
#define FALSE 0
#define BOUNDED TRUE
#define MODULO TRUE
#define MAXINT 2147483647
#define MININT -2147483648
 logical bounded, modulo
 integer maxint, minint
 parameter (bounded = .TRUE.)
 parameter (modulo = .TRUE.)
 parameter (maxint = 2147483647)
 parameter (minint = -2147483648)
```

E.2.3 d.DIV/REM/MOD (LIA 5.1.3):

C の / と %、および Fortran の / と mod() によって、`DIVtI(x,y)` と `REMtI(x,y)` が提供されます。また、`modaI(x,y)` も次のコードで使用できます。

```
int modaI(int x, int y) {
 int t = x % y;
 if (y < 0 && t > 0)
 t -= y;
 else if (y > 0 && t < 0)
 t += y;
  return t;
 }
```

これは、次のコードでも使用できます。

```
 integer function modaI(x, y)
 integer x, y, t
 t = mod(x, y)
 if (y .lt. 0 .and. t .gt. 0) t = t - y
 if (y .gt. 0 .and. t .lt. 0) t = t + y
 modaI = t
 return
 end
```

E.2.4 i.表記法 (LIA 5.1.3):

次の表は、LIA の整数演算で認識される表記法を示しています。

表 E-3 LIA-1 への準拠 - 表記法

LIA	C	Fortran (異なる場合)
`addI(x,y)`	`x+y`	n/a
`subI(x,y)`	`x-y`	n/a

LIA	C	Fortran (異なる場合)
mulI(x,y)	x*y	n/a
divtI(x,y)	x/y	n/a
remtI(x,y)	x%y	mod(x,y)
modaI(x,y)	上記を参照	n/a
negI(x)	-x	n/a

absI(x)	#include <stdlib.h>
	abs(x)
		abs(x)

signI(x)	#define signI(x) (x > 0 ? 1 : (x < 0 ? -1 : 0))	下記を参照
eqI(x,y)	x==y	x.eq.y
neqI(x,y)	x!=y	x.ne.y
lssI(x,y)	x<y	x.lt.y
leqI(x,y)	x<=y	x.le.y
gtrI(x,y)	x>y	x.gt.y
geqI(x,y)	x>=y	x.ge.y

次のコードは、signI(x) の Fortran での表記法を示しています。

```
integer function signi(x)
integer x, t
if (x .gt. 0) t=1
if (x .lt. 0) t=-1
if (x .eq. 0) t=0
return
end
```

E.2.5 j.式評価:

デフォルトでは、最適化が指定されていない場合、式は int (C) または INTEGER (Fortran) の精度で評価されます。括弧も同様です。a + b + c や a * b * c などの、括弧で囲まれていない連結した式の評価順序は規定されていません。

E.2.6 k.パラメータの取得方法:

ソースコード内の 193 ページの「b.パラメータ (LIA 5.1):」に定義が含まれています。

E.2.7 n.通知:

整数の例外は、x/0 と x%0、または mod(x,0) です。デフォルトでは、これらの例外によって SIGFPE が生成されます。SIGFPE に対してシグナルハンドラが指定されていない場合は、プロセスが終了し、メモリーがダンプされます。

E.2.8 o.選択メカニズム:

signal(3) または signal(3F) を使用すると、SIGFPE に対するユーザー例外処理を有効にすることができます。

♦ ♦ ♦ 付 録 F

参考資料

次のマニュアルには、SPARC® 浮動小数点ハードウェアに関する詳細が記載されています。

- UltraSPARC アーキテクチャー 2005 (T1 用の基本 ISA) (http://www.oracle.com/technetwork/systems/opensparc/1537734)
- UltraSPARC アーキテクチャー 2007 (T2、T2+、T3 用の基本 ISA) (http://www.oracle.com/technetwork/systems/hardware/usparcarchdoc2007-329425.pdf)
- Oracle SPARC アーキテクチャー 2011 (T4、T5、M5、M6 用の基本 ISA) (http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf)

残りの参考資料は章ごとにまとめられています。標準規格に関するドキュメントとテストプログラムの取得に関する情報は最後に記載されています。

F.1 第 2 章:「IEEE 演算」

Cody 他著、『A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic』、IEEE Computer、1984 年 8 月。

Coonen, J.T. 著、『An Implementation Guide to a Proposed Standard for Floating Point Arithmetic』、Computer, Vol. 13, No. 1、1980 年 1 月、68 - 79 ページ。

Demmel, J. 著、『Underflow and the Reliability of Numerical Software』、SIAM J. 『*Scientific Statistical Computing*』、Volume 5 (1984)、887 - 919。

Hough, D. 著、『Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic』、Computer, Vol. 13, No. 1、1980 年 1 月、70 - 74 ページ。

Kahan, W. および Coonen, J.T. 著、「The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments」(『The Relationship

between Numerical Computation and Programming Languages』に収録)、Reid, J.K. (編集)、North-Holland Publishing Company、1982 年。

Kahan, W. 著、「Implementation of Algorithms」、Computer Science Technical Report No. 20、カリフォルニア大学、Berkeley CA、1973 年。National Technical Information Service から入手可能、NTIS ドキュメント番号 AD–769 124 (339 ページ)、1-703-487-4650 (通常注文) または 1-800-336-4700 (至急注文)。

Karpinski, R. 著、『Paranoia: a Floating-Point Benchmark』、Byte、1985 年 2 月。

Knuth, D.E. 著、『The Art of Computer Programming, Vol.2: Semi-Numerical Algorithms』、Addison-Wesley, Reading, Mass、1969 年、195 ページ。

Linnainmaa, S. 著、「Combatting the effects of Underflow and Overflow in Determining Real Roots of Polynomials」、SIGNUM ニュースレター、(1981 年)、11 - 16。

Rump, S.M. 著、「How Reliable are Results of Computers?」(「Wie zuverlassig sind die Ergebnisse unserer Rechenanlagen?」の翻訳)、Jahrbuch Uberblicke Mathematik 1983 年、163 - 168 ページ、C Bibliographisches Institut AG 1984 年。

Sterbenz, P 著、『Floating-Point Computation』、Prentice-Hall, Englewood Cliffs, NJ、1974 年。(絶版。ほとんどの大学図書館にあります)。

Stevenson, D. 他、Cody, W.、Hough, D.Coonen, J.著、2 進浮動小数点演算の標準の草案を提案および解析したさまざまな論文、IEEE Computer、1981 年 3 月。

提案された IEEE Floating-Point Standard、ACM SIGNUM ニュースレターの特別号、1979 年 10 月。

F.2　第 3 章:「数学ライブラリ」

Cody, William J. および Waite, William 著、『Software Manual for the Elementary Functions』、Prentice-Hall, Inc.、Englewood Cliffs, New Jersey, 07632、1980 年。

『Contributions to a Proposed Standard for Binary Floating-Point Arithmetic』、カリフォルニア大学バークレー校博士論文、1984 年。

Tang, Peter Ping Tak 著、『Some Software Implementations of the Functions Sin and Cos』、Technical Report ANL-90/3、Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois、1990 年 2 月。

Tang, Peter Ping Tak 著、『Table-driven Implementations of the Exponential Function EXPM1 in IEEE Floating-Point Arithmetic』、予稿 MCS-P125-0290、Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois、1990 年 2 月。

Tang, Peter Ping Tak 著、『Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic』、ACM Transactions on Mathematical Software, Vol. 15, No. 2、1989 年 6 月、144 - 157 ページ、コミュニケーション、1988 年 7 月 18 日。

Tang, Peter Ping Tak 著、『Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic』、予稿 MCS-P55-0289、Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois、1989 年 2 月 (ACM Trans. on Math. Soft に掲載)。

Park, Stephen K. および Miller, Keith W. 著、『Random Number Generators: Good Ones Are Hard To Find』、Communications of the ACM, Vol. 31, No. 10、1988 年 10 月、1192 - 1201 ページ。

F.3　第 4 章:「例外と例外処理」

Coonen, J.T 著、『Underflow and the Denormalized Numbers』、Computer, 14, No. 3、1981 年 3 月、75 - 87 ページ。

Demmel, J. および X.Li 著、『Faster Numerical Algorithms via Exception Handling』、IEEE Trans。Comput.Vol. 48, No. 8、1994 年 8 月、983 - 992 ページ。

Kahan, W. 著、『A Survey of Error Analysis』、Information Processing 71, North-Holland, Amsterdam、1972 年、1214 - 1239 ページ。

F.4 標準規格

American National Standard for Information Systems ISO/IEC 9899:1999 プログラミング言語 - C (C99)、American National Standards Institute, 1430 Broadway, New York, NY 10018。

IEEE Standard for Floating-Point Arithmetic、ANSI/IEEE 標準 754-2008、Institute of Electrical and Electronics Engineers, Inc 発行、3 Park Avenue, New York, NY 10016、2008 年。

IEEE Standard Glossary of Mathematics of Computing Terminology、ANSI/IEEE 標準 1084-1986、Institute of Electrical and Electronics Engineers, Inc 発行、345 East 47th Street, New York, NY 10017、1986 年。

IEEE Standard Portable Operating System Interface for Computer Environments (POSIX®)、IEEE 標準 1003.1-1988、Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017。

『System V Application Binary Interface (ABI)』、AT&T (1-800-432-6600)、1989 年。

『SPARC System V ABI Supplement (SPARC ABI)』、AT&T (1-800-432-6600)、1990 年。

『System V Interface Definition, 3rd edition, (SVID89, or SVID Issue 3)』、Volumes I–IV、部品番号 320-135、AT&T (1-800-432-6600)、1989 年。

『X/OPEN Portability Guide, Set of 7 Volumes』、Prentice-Hall, Inc.、Englewood Cliffs, New Jersey 07632、1989 年。

F.5 テストプログラム

浮動小数点演算と数学ライブラリの多数のテストプログラムは、Netlib から ucbtest パッケージで入手できます。これらのプログラムには、`Paranoia` のバージョン、Z. Alex Liu のバークレー初等関数テストプログラム、IEEE テストベクトル、および Prof. W. Kahan の開発による数論的方法に基づく、正しく丸められた乗算、除算、および平方根のハードテストケースを生成するプログラムが含まれています。

ucbtest は http://www.netlib.org/fp/ucbtest.tgz にあります。

用語集

A-E

共通例外 3 つの浮動小数点例外であるオーバーフロー、無効、およびゼロ除算はまとめて、`ieee_flags`(3m) および `ieee_handler`(3m) のための共通例外と呼ばれています。これらは共通してエラーとしてトラップされるために共通例外と呼ばれます。

コンテキストスイッチ SunOS™ オペレーティングシステムなどのマルチタスクオペレーティングシステムでは、プロセスは定められた時間だけ実行されます。その時間の終わりに、CPU はタイマーからシグナルを受信し、現在実行中のプロセスを中断し、新しいプロセスの実行を準備します。CPU は古いプロセスのレジスタを保存し、続いて新しいプロセスのレジスタをロードします。古いプロセスの状態から新しい状態に切り替えることをコンテキストスイッチと呼びます。コンテキストスイッチにかかる時間はシステムオーバーヘッドです。所要時間は、レジスタの数、およびプロセスに関連付けられたレジスタを保存するための特殊な命令があるかどうかによって異なります。

指数 表現された数値を算出するため、基数をどれだけ累乗するかを示す整数べきを表した浮動小数点数の構成要素。

正確さ ある数値が別の数値にどれだけ近似しているかを表す尺度。たとえば、計算結果の正確さは、多くの場合、計算値がその誤差のために数学的に正確な結果からどの程度異なっているかを反映します。正確さは、たとえば「結果は小数点第 6 位まで正確である」のように有効桁数として、またはより一般的に、関連する数学的プロパティーの保持 (たとえば「結果の算術符号が正しい」) のように表すことができます。

デフォルト結果 例外に対してほかの処理が指定されていなかった場合に、その例外の原因となった浮動小数点演算の結果として配布された値。

突発的アンダーフロー 浮動小数点演算がアンダーフローすると、結果が非正規数の範囲内にある場合でも、常にゼロを返します。

バイアス付きの指数 格納された指数の範囲を負でなくするために選択された定数 (バイアス) と基数が 2 の指数の和。たとえば、2^{-100} の指数は、IEEE 単精度形式では (-100) + (127 の単精度バイアス) = 27 として格納されます。

倍精度 精度を維持および向上させるため、2 ワードを使用して 1 つの数値を表すこと。SPARC® ワークステーションでは、倍精度は 64 ビット IEEE 倍精度です。

非正規化数 非正規数の古い命名。

例外 算術例外は、不可分算術演算を試みた結果が、どこでも許容可能な結果にならなかったときに起こります。不可分と許容の意味は、時間と場所によって異なります。

連鎖 演算の結果を、2 番目の演算のオペランドとしてただちに使用すると同時に、デスティネーションレジスタに結果を書き込めるようにする一部のパイプラインアーキテクチャーのハードウェア機能。2 つの連鎖した演算の合計サイクル時間は、その命令の単独でのサイクル時間の合計より短くなります。たとえば TI 8847 は、(同じ精度の) 連続する `fadd`、`fsub`、および `fmul` の連鎖をサポートします。連鎖した `faddd/fmuld` は 12 サイクル必要ですが、連鎖していない連続の `faddd/fmuld` は 17 サイクル必要です。

binade 2 つの連続する 2 の累乗値の間隔。

F-I

インラインテンプレート Sun Studio コンパイラのインラインパス中に、定義済みの関数コールと置き換えられる、アセンブリ言語コードのフラグメント。(たとえば) C プログラムから三角関数やほかの初等関数のハードウェア実装にアクセスするために、インラインテンプレートファイル (`libm.il`) の数学ライブラリで使用されます。

隠しビット 丸めが正しいことを確認するためにハードウェアで使用される追加ビットであり、ソフトウェアではアクセスできません。たとえば、IEEE 倍精度演算では、3 つの隠しビットを使用して 56 ビットの結果を計算します。この結果は、その後 53 ビットに丸められます。

段階的アンダーフロー 浮動小数点演算が非正規数の範囲にアンダーフローすると、0 の代わりに非正規数を返します。このアンダーフローを処理する方法では、小さい数に対する浮動小数点計算での正確さの低下を最小限に抑えます。

浮動小数点数体系 表現可能な数の間の間隔が固定された絶対定数でない実数の部分集合を表すための体系。このような体系は、基数、符号、仮数、および指数 (通常はバイアス) で特徴付けられます。数値は、バイアスなしの指数まで基数を累乗した数とその仮数の符号付きの積です。

IEEE 標準 754 1985 年に発行され、2008 年に改訂された、Institute of Electrical and Electronics Engineers が定めた 2 進浮動小数点演算の標準。

L-P

正規数 IEEE 演算で、有界値のある小さな相対誤差を伴う、正常範囲の実数の部分集合を表す、ゼロでも最大値 (すべて 1) でもないバイアス付きの指数を持つ数。

精度 表現可能な数値の密度の定量的測度。たとえば、53 の有効ビットの精度を持つ 2 進浮動小数点形式では、(正規数の範囲内で) 2 つの隣接する 2 の累乗の間に 253 の表現可能な数値があります。精度は、ある数が別の数にどれだけ近似しているかを表す正確さと混同しないでください。

パイプライン化 演算が複数の段階に減らされるハードウェア機能であり、その段階はそれぞれ (通常は) 1 サイクルかかる。パイプラインは、各サイクルで新しい演算が可能になるときに満たされます。パイプ内の命令間に依存関係がない場合、サイクルごとに新しい結果をもたらすことができます。連鎖は、依存している命令同士のパイプライン化を意味します。依存した命令が連鎖できない場合 (ハードウェアがこれらの特定の命令の連鎖をサポートしていない場合)、パイプラインは停止します。

NaN 非数を表します。浮動小数点形式でエンコードされた記号エンティティー。

Q-R

基数 あらゆる数体系の基となる数。たとえば、2 は 2 進法の基数であり、10 は 10 進法の基数です。SPARC ワークステーションは基数 2 の演算を使用しており、IEEE 標準 754 は基数 2 の演算の標準規格です。

シグナルを発生しない NaN 新しい例外を発生することなく、ほとんどの算術演算によって伝達される NaN (非数)。

丸め 厳密でない結果は、表現可能な値にするため切り捨てまたは切り上げを行う必要があります。切り上げられると、結果は増大されて次の表現可能な値になります。切り下げられると、結果は縮小され、直前の表現可能な値になります。

丸め誤差 実数がマシンで表現可能な数に丸められたときに生じる誤差。ほとんどの浮動小数点計算では、丸め誤差が発生します。IEEE 標準 754 では、1 回の浮動小数点演算でその結果に丸め誤差が複数生じることは認められていません。

S-T

2 の補数 1 から各桁を差し引き、次に最後の有効桁に 1 を加え、必要な桁上げを行うことで求められる、2 進数の基数の補数。たとえば、1101 の 2 の補数は 0011 です。

仮数 符号付きの基数の累乗で乗算されて数の値を特定する浮動小数点数の構成要素。正規化数では、仮数は、基数点の左側のゼロ以外の 1 つの桁と右側の小数部分から構成されます。

シグナルを発生する NaN オペランドとして現れるときは常に、無効な処理例外を生成する NaN (非数)。

ゼロ格納 突発的アンダーフローと同じです。*突発的アンダーフロー*を参照してください。

単精度 1 つのコンピュータワードを使用して 1 つの数を表すこと。

非正規数 IEEE 演算では、バイアス付きゼロの指数を含むゼロ以外の浮動小数点数。非正規数は、ゼロと最小の正規数の間にある数です。

stderr 標準エラー (Standard Error) は、標準エラー出力への UNIX ファイルポインタです。このファイルは、プログラムが起動すると開かれます。

U-Z

アンダーフロー 浮動小数点算術演算の結果が非常に小さいため、通常の丸めだけではデスティネーションの浮動小数点形式で正規数として表現できない場合に起きる状況。

ラップ数 IEEE 演算で、そのままではオーバーフローまたはアンダーフローするような値に対して、ラップされた値を正規数範囲に位置付けられるように、その指数に固定オフセットを加えて作成した数。現在、ラップされた結果は SPARC ワークステーションでは正しく生成されません。

ワード 特定のコンピュータ内で単一のエンティティーとして格納、アドレス指定、転送、および演算が行われる順序付けられた文字セット。SPARC ワークステーションの場合、1 ワードは 32 ビットです。

ulp 最下位の単位 (unit in last place) を表します。2 進形式では、仮数の最小有効ビットであるビット 0 が最下位の単位です。

ulp(x) 処理中の形式で切り捨てられた x の `ulp` を表します。

索引

数字・記号

-fast, 163
-fnonstd, 163
0 へのフラッシュ (abrupt underflow を参照), 33
10 進数表現
  最小の正の正規数, 28
  最大の正の正規数, 28
  精度, 28
  範囲, 28
10 進数文字列と 2 進浮動小数点数の間の変換, 16

あ

アンダーフロー
  nonstandard_arithmetic, 58
  しきい値, 38
  段階的, 33
  浮動小数点演算, 33
アンダーフローしきい値
  拡張倍精度, 33
  単精度, 33
  倍精度, 33
オペレーティングシステム数学ライブラリ
  libm.a, 43

か

基数変換
  基数 10 から基数 2 への, 31
  基数 2 から基数 10 への, 32
  書式付き I/O, 31
クロック速度, 164

さ

最終桁単位 (ulp), 62
三角関数
  引数還元, 62, 62
シグナルを発生しないNaN
  無効な演算のデフォルトの結果, 69
数値のセット間の変換, 30
数値の配列の生成
  FORTRAN の例, 117
数直線
  2 進数表現,, 29
  2 のべき乗, 36
  10 進数表現, 29
正確性
  しきい値, 39
  浮動小数点演算, 16
  有効桁数, 29
正規数
  最小の正の, 33, 37
  最大の正の, 19

た

段階的アンダーフロー
  誤差の属性, 35
単精度形式, 17
単精度値の表現
  C の例, 116
単精度表現
  C の例, 115
データ型
  IEEE 形式との関係, 17
突発的アンダーフロー
  アンダーフローの結果のフラッシュ, 38
トラップ
  ieee_retrospective, 56
  例外での中止, 138

は

倍精度値の表現
C の例, 116
FORTRAN の例, 116
倍精度の表現
FORTRAN の例, 116
倍精度表現
C の例, 115
引数還元
三角関数, 62
非順序付け比較
NaN, 70
浮動小数点値, 70
非正規数, 37
浮動小数点演算, 33
浮動小数点
丸め精度, 16
丸め方向, 16
例外のリスト, 16
浮動小数点キュー (FQ), 158
浮動小数点ステータスレジスタ (FSR), 151, 158
浮動小数点の正確性
10 進数文字列と 2 進浮動小数点数, 16
浮動小数点例外, 13
ieee_functions, 51
ieee_retrospective, 56
一般的な例外, 68
定義, 68
デフォルトの結果, 69
トラップの優先度, 71
フラグ, 71
現在, 71
累積, 71
優先度, 71
累積例外ビット, 129
例外での中止, 138
例外のリスト, 68
浮動小数点例外のトラップ
C の例, 132

ま

丸め誤差
正確性
損失, 34
丸め精度, 16
丸め方向, 16
C の例, 124

ら

乱数ジェネレータ, 117
乱数ユーティリティー
shufrans, 64
累積例外ビットの検査
C の例, 129
累積例外フラグの検査
C の例, 130
例外での中止
C の例, 138
例外のトラップ
C の例, 132, 133
例外フラグの設定
C の例, 131

A

abrupt underflow, 33
アンダーフローの結果のフラッシュ, 37
addrans
乱数ユーティリティー, 64

C

C ドライバ
例、C から FORTRAN のサブルーチンを呼び出す, 147
convert_external
2 進浮動小数点, 63
データ変換, 63

D

dbx, 75

F

floatingpoint.h
ハンドラタイプの定義

C および C++, 82

I

ieee_flags
  切り捨て丸め, 55
  丸め精度, 54, 55
  丸め方向, 54
  累積例外ビットの検査 - C の例, 129
  例外発生フラグ, 54
  例外フラグの設定 - C の例, 131
ieee_functions
  ビットマスク演算, 50
  浮動小数点例外, 51
ieee_handler, 82
  一般的な例外のトラップ, 68
  例外での中止
    FORTRAN の例, 138
  例外のトラップ
    C の例, 132
  例, 呼び出しシーケンス, 76
ieee_retrospective
  nonstandard_arithmetic が有効, 56
  アンダーフロー例外フラグの確認, 163
  精度, 56
  非標準の IEEE モードに関する情報の取得, 56
  浮動小数点ステータスレジスタ (FSR), 56
  浮動小数点例外, 56
  丸め, 56
  未処理の例外に関する情報の取得, 56
  例外メッセージの抑制, 57
ieee_sun
  IEEE 分類関数, 50
ieee_values
  4 倍精度の値, 52
  Inf の表現, 51
  NaN の表現, 51
  正規数の表現, 51
  単精度の値, 52
  浮動小数点値の表現, 51
ieee_values 関数
  C の例, 122
IEEE 拡張倍精度形式
  4 倍精度
    SPARC アーキテクチャー, 22
  Inf
    SPARC アーキテクチャー, 24
    x86 アーキテクチャー, 26
  NaN
    x86 アーキテクチャー, 28
  仮数
    明示的先行ビット (x86 アーキテクチャー), 25
  小数部
    x86 アーキテクチャー, 25
  正規数
    SPARC アーキテクチャー, 23
    x86 アーキテクチャー, 26
  バイアス付き指数
    x86 アーキテクチャー, 25
  非正規数
    SPARC アーキテクチャー, 23
    x86 アーキテクチャー, 26
  ビットフィールドの割り当て
    x86 アーキテクチャー, 25
  符号ビット
    x86 アーキテクチャー, 25
IEEE 規格 754
  拡張倍精度形式, 15
  単精度形式, 15
  倍精度形式, 15
IEEE 形式
  言語のデータ型との関係, 17
IEEE 単精度形式
  Inf, 正の無限大, 18
  Inf, 負の無限大, 18
  NaN, 非数, 19
  混在した数値, 仮数, 19
  小数部, 17
  正規数
    最大の正の, 19
  正規数のビットパターン, 18
  精度、正規数, 19
  バイアス付き指数, 17
  バイアス付き指数, 暗黙ビット, 18
  非正規化数, 19
  非正規数のビットパターン, 18
  ビットの割り当て, 17
  ビットパターンおよび同等の値, 19
  ビットフィールドの割り当て, 17
  符号ビット, 18
IEEE 倍精度形式

Inf, 無限大, 21
NaN, 非数, 22
暗黙ビット, 21
仮数, 21
小数部, 20, 20
SPARC 上の格納, 20
x86 上の格納, 20
正規数, 21
精度, 21
バイアス付き指数, 20
非正規化数, 21
非正規数, 21
ビットパターンおよび同等の値, 21
ビットフィールドの割り当て, 20
符号ビット, 20
Inf, 13, 189
0 による除算のデフォルトの結果, 69

L

lcrans
乱数ユーティリティー, 64
libm
関数のリスト, 44
libm 関数
4 倍精度, 49
単精度, 49
倍精度, 49
libsunmath
関数のリスト, 47

N

NaN, 13, 25, 188
nonstandard_arithmetic
IEEE 段階的アンダーフローをオフにする, 163
アンダーフロー, 58
段階的アンダーフロー, 58

P

pi
無限に正確な値, 63

S

shufrans
擬似乱数のシャッフル, 64
standard_arithmetic
IEEE 動作をオンにする, 163
System V Interface Definition (SVID), 187