



数値計算ガイド

Sun™ Studio 8

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

Part No. 817-5809-10
2004 年 3 月 Revision A

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

この配布には、第三者が開発したソフトウェアが含まれている可能性があります。

フォント技術を含む第三者のソフトウェアは、著作権法により保護されており、提供者からライセンスを受けているものです。

libdwarf and lldredblack are Copyright 2000 Silicon Graphics Inc. and are available under the GNU Lesser General Public License from <http://www.sgi.com>.

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。

Sun、Sun Microsystems、Forte、Java、iPlanet、NetBeans および docs.sun.com は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。

サンロゴマークおよび Solaris は、米国 Sun Microsystems 社の登録商標です。

Netscape および Netscape Navigator は、米国ならびに他の国における Netscape Communications Corporation の商標または登録商標です。

すべての SPARC の商標はライセンス規定に従って使用されており、米国および他の各国における SPARC International, Inc. の商標または登録商標です。SPARC の商標を持つ製品は、Sun Microsystems, Inc. によって開発されたアーキテクチャに基づいています。

このマニュアルに記載されている製品および情報は、米国の輸出規制に関する法規の適用および管理下にあり、また、米国以外の国の輸出および輸入規制に関する法規の制限を受ける場合があります。核、ミサイル、生物化学兵器もしくは原子力船に関連した使用またはかかる使用者への提供は、直接的にも間接的にも、禁止されています。このソフトウェアを、米国の輸出禁止国へ輸出または再輸出すること、および米国輸出制限対象リスト(輸出が禁止されている個人リスト、特別に指定された国籍者リストを含む)に指定された、法人、または団体に輸出または再輸出することは一切禁止されています。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含み、明示的であるか黙示的であるかを問わず、あらゆる説明および保証は、法的に無効である限り、拒否されるものとします。

原典： Numerical Computation Guide
Part No: 817-5073-10
Revision A

© 2004 by Sun Microsystems, Inc.



目次

はじめに xi

1. 概要 1

浮動小数点環境 1

2. IEEE 演算機能 3

IEEE 演算モデル 3

IEEE の演算機能 3

IEEE 形式 5

記憶形式 5

単精度の記憶形式 6

倍精度の記憶形式 8

拡張倍精度の記憶形式 (SPARC) 11

拡張倍精度の記憶形式 (x86) 13

10 進表現の範囲と精度 17

Solaris 環境における基数変換 20

アンダーフロー 21

アンダーフローしきい値 22

IEEE 演算機能におけるアンダーフローの扱い 22

段階的アンダーフローの利点	23
段階的アンダーフローの誤差特性	24
段階的アンダーフローと <code>store 0</code> の 2 つの例	27
アンダーフローは問題か	28

3. 数学ライブラリ 29

数学ライブラリ 29

付加価値数学ライブラリ 31

Sun 数学ライブラリ 31

最適化ライブラリ 33

ベクトル数学ライブラリ (SPARC のみ) 34

libm9X 数学ライブラリ 35

単精度、倍精度、4 倍精度 38

IEEE サポート関数 39

`ieee_functions(3m)` と `ieee_sun(3m)` 39

`ieee_values(3m)` 41

`ieee_flags(3m)` 43

`ieee_retrospective(3m)` 45

`nonstandard_arithmetic(3m)` 48

C99 浮動小数点環境関数 49

例外フラグ関数 49

丸め制御 50

環境関数 51

libm と libsunmath の実装上の特徴 52

アルゴリズム 52

三角関数の引数還元 53

データ変換ルーチン 54

乱数発生機構 54

4. 例外と例外処理 57

例外とは 58

表 4-1 の注 60

例外の検出 62

`ieee_flags(3m)` 62

C99 例外フラグ関数 65

例外の特定 66

デバッガを使用して例外を特定する 67

シグナルハンドラを使用して例外を特定する 75

`libm9x.so` の例外処理機能を使用して例外を検出する 82

例外処理 89

A. 例 105

IEEE の演算機能 105

数学ライブラリ 107

乱数生成 108

IEEE が推奨する関数 111

IEEE の特殊な値 115

`ieee_flags` - 丸め方向 118

C99 浮動小数点環境関数 120

例外と例外処理 125

`ieee_flags` - 例外 125

`ieee_handler` - 例外のトラップ 129

`ieee_handler` - 例外での異常終了 138

`libm9x.so` の例外処理機能 138

FORTTRAN プログラムでの `libm9x.so` の使用 145

その他 149

`sigfpe` - 整数例外のトラップ 149

FORTRAN を呼び出す C	151
効果的なデバッグコマンド	155
B. SPARC の動作と実装	159
浮動小数点ハードウェア	159
浮動小数点状態レジスタと待ち行列	163
ソフトウェアサポートが必要な特別な場合	165
fpversion(1) 関数 - FPU に関する情報の検索	171
C. x86 の動作と実装	173
D. 浮動小数点演算について	175
概要	175
はじめに	176
丸め誤差	177
浮動小数点フォーマット	177
相対誤差と ulp	179
保護桁	181
相殺	182
正確な丸め操作	187
IEEE 標準	191
フォーマットと操作	192
特殊な数	198
NaNs	199
例外、フラグ、トラップハンドラ	206
システムの側面	211
命令セット	212
言語とコンパイラ	214
例外処理	222

詳細	224
丸め誤差	224
2 進数から 10 進数への変換	234
加法の誤差	236
まとめ	237
謝辞	238
参考資料	238
定理 14 と定理 8	241
定理 14	241
定理 8 (Kahan の加法公式)	243
IEEE 754 実装間の違い	246
現在の IEEE 754 実装	248
拡張ベースシステムにおける演算の落とし穴	249
拡張精度に対するプログラミング言語サポート	255
結論	260
 E. 規格への準拠	 263
SVID の歴史	263
IEEE 754 の歴史	264
SVID の将来の方向	264
SVID の実装	265
例外のケースと libm 関数についての一般的注意事項	267
libm についての注意	269
LIA-1 準拠	269
 F. 参考文献	 273
第 2 章「IEEE 演算機能」	273
第 3 章「数学ライブラリ」	274

第 4 章「例外と例外処理」	275
付録 B「SPARC の動作と実装」	275
規格書	276
試験プログラム	277
用語集	279
索引	303

図目次

図 2-1	単精度の記憶形式	6
図 2-2	倍精度の記憶形式	9
図 2-3	拡張倍精度の記憶形式 (SPARC)	11
図 2-4	拡張倍精度の記憶形式 (x86)	14
図 2-5	10 進と 2 進表現で定義された数の比較	18
図 2-6	数直線	25
図 B-1	SPARC 浮動小数点状態レジスタ	164
図 D-1	正規化数 $\beta = 2$ 、 $p = 3$ 、 $e_{\min} = -1$ 、 $e_{\max} = 2$	179
図 D-2	段階的アンダーフローとゼロフラッシュの比較	205

表目次

表 2-1	IEEE 形式と言語の種類	5
表 2-2	IEEE 単精度記憶形式 のビットパターンで表現した値	6
表 2-3	単精度記憶形式でのビットパターンとその IEEE 値	7
表 2-4	IEEE 倍精度記憶形式のビットパターンで表現した値	9
表 2-5	倍精度記憶形式でのビットパターンとその IEEE 値	10
表 2-6	IEEE 拡張倍精度記憶形式のビットパターンで表現した値 (SPARC)	12
表 2-7	拡張倍精度記憶形式でのビットパターンとその値 (SPARC)	13
表 2-8	IEEE 拡張倍精度記憶形式のビットパターンで表現した値 (x86)	15
表 2-9	拡張倍精度記憶形式でのビットパターンとその値 (x86)	16
表 2-10	各記憶形式の範囲と精度	20
表 2-11	アンダーフローしきい値	22
表 2-12	4 つの異なる精度における ulp(1)	25
表 2-13	表現可能な単精度浮動小数点数の差	26
表 3-1	libm の内容	30
表 3-2	libsunmath の内容	31
表 3-3	libmvec の内容	35
表 3-4	libm9x の内容	36
表 3-5	単精度、倍精度、および 4 倍精度 libm 関数の呼び出し	38
表 3-6	ieee_functions(3m)	39
表 3-7	ieee_sun(3m)	40
表 3-8	FORTTRAN からの ieee_functions の呼び出し	40

表 3-9	FORTTRAN からの <code>ieee_sun</code> の呼び出し	41
表 3-10	IEEE 値: 単精度	41
表 3-11	IEEE 値: 倍精度	42
表 3-12	IEEE 値: 4 倍精度 (SPARC)	42
表 3-13	IEEE 値: 拡張倍精度 (x86)	43
表 3-14	<code>ieee_flags</code> のパラメータ値	44
表 3-15	<code>ieee_flags</code> による丸め方向の入力値	45
表 3-16	C99 規格例外フラグ関数	49
表 3-17	<code>libm9x.so</code> 浮動小数点環境関数	51
表 3-18	単一値乱数の値の発生範囲	55
表 4-1	IEEE 浮動小数点例外	58
表 4-2	非順序付け比較	61
表 4-3	例外の優先順位	61
表 4-4	例外ビット	64
表 4-5	算術例外の型	78
表 4-6	<code>fex_set_handling</code> の例外コード	83
表 A-1	デバッグコマンド (SPARC)	155
表 A-2	デバッグコマンド (x86)	156
表 B-1	SPARC 浮動小数点オプション	160
表 B-2	浮動小数点状態レジスタフィールド	164
表 B-3	例外処理フィールド	165
表 D-1	IEEE 754 フォーマットのパラメータ	194
表 D-2	IEEE 754 の特殊な値	199
表 D-3	NaN が生成される操作	200
表 D-4	IEEE 754 の例外	207
表 E-1	例外のケースと <code>libm</code> 関数	265
表 E-2	LIA-1 準拠の記数法	271

はじめに

このマニュアルでは、Solaris™ オペレーティング環境が稼働する SPARC® および x86 プラットフォーム上のソフトウェアとハードウェアがサポートする浮動小数点環境について説明します。このマニュアルは SPARC および Intel アーキテクチャの概要も含んでいますが、基本的には Sun™ の言語関係の製品に付属するリファレンスマニュアルです。

このマニュアルでは、IEEE 2 進浮動小数点演算規格の一部について説明します。IEEE 演算機能についての詳細は、該当規格書の 18 ページを参照してください。IEEE 演算機能に関する著書目録については、付録 F「参考文献」を参照してください。

内容の紹介

このマニュアルは次の章と付録から構成されています。

第 1 章「概要」

浮動小数点環境について説明します。

第 2 章「IEEE 演算機能」

IEEE 演算モデル、IEEE 形式、アンダーフローについて説明します。

第 3 章「数学ライブラリ」

Sun™ Open Net Environment (Sun ONE) Studio コンパイラで提供される数学ライブラリについて説明します。

第 4 章「例外と例外処理」

例外と、例外の検出、特定、処理について説明します。

付録 A「例」

プログラム例を示します。

付録 B「SPARC の動作と実装」

SPARC ワークステーションのための浮動小数点ハードウェアオプションについて説明します。

付録 C「x86 の動作と実装」

Intel と SPARC の互換性に関する注意のうち、x86 マシンで使用される浮動小数点ユニットに関連する部分について説明します。

付録 D「浮動小数点演算について」

David Goldberg 氏による浮動小数点演算の具体例を編集したものです。

付録 E「規格への準拠」

標準準拠について説明します。

付録 F「参考文献」

参考文献を一覧します。

「用語集」

用語の定義を一覧します。

このマニュアルでは、C と FORTRAN を例としていますが、概念は SPARC、x86 システム上のどのコンパイラにも適用できます。

書体と記号について

次の表と記述は、このマニュアルで使用している書体と記号について説明しています。

書体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 machine_name% You have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。	machine_name% su Password:
AaBbCc123 またはゴシック	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	rm <i>filename</i> と入力します。 rm ファイル名 と入力します。
『』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』
「」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。	machinename% grep `^#define \ XV_VERSION_STRING`
➤	階層メニューのサブメニューを選択することを示します。	作成: 「返信」 ➤ 「送信者へ」

シェルプロンプトについて

シェル	プロンプト
UNIX の C シェル	machine_name%
UNIX の Bourne シェルと Korn シェル	machine_name\$
スーパーユーザー (シェルの種類を問わない)	#

コンパイラコレクションのツールとマニュアルページへのアクセス

コンパイラコレクションのコンポーネントとマニュアルページは、標準の `/usr/bin/` と `/usr/share/man` の各ディレクトリにインストールされません。コンパイラとツールにアクセスするには、`PATH` 環境変数にコンパイラコレクションのコンポーネントディレクトリを必要とします。マニュアルページにアクセスするには、`PATH` 環境変数にコンパイラコレクションのマニュアルページディレクトリが必要です。

`PATH` 変数についての詳細は、`csh(1)`、`sh(1)` および `ksh(1)` のマニュアルページを参照してください。`MANPATH` 変数についての詳細は、`man(1)` のマニュアルページを参照してください。このリリースにアクセスするために `PATH` および `MANPATH` 変数を設定する方法の詳細は、『インストールガイド』を参照するか、システム管理者にお問い合わせください。

注 – この節に記載されている情報は Sun ONE Studio コンパイラコレクションコンポーネントが `/opt` ディレクトリにインストールされていることを想定しています。ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

コンパイラとツールへのアクセス方法

PATH 環境変数を変更して、コンパイラとツールにアクセスできるようにする必要があるかどうか判断するには以下を実行します。

▼ PATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、PATH 変数の現在値を表示します。

```
% echo $PATH
```

2. 出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。

パスがある場合は、PATH 変数はコンパイラとツールにアクセスできるように設定されています。パスがない場合は、次の指示に従って、PATH 環境変数を設定してください。

▼ PATH 環境変数を設定してコンパイラとツールにアクセスする

1. C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。
2. 次のパスを PATH 環境変数に追加します。

```
/opt/SUNWspro/bin
```

マニュアルページへのアクセス方法

マニュアルページにアクセスするために MANPATH 変数を変更する必要があるかどうかを判断するには以下を実行します。

▼ MANPATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、dbx マニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

dbx(1) マニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って MANPATH 環境変数を設定してください。

▼ MANPATH 変数を設定してマニュアルページにアクセスする

1. C シェルを使用している場合は、ホームの `.cshrc` ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの `.profile` ファイルを編集します。
2. 次のパスを PATH 環境変数に追加します。

```
/opt/SUNWspro/man
```

コンパイラコレクションのマニュアルへのアクセス

マニュアルには、以下からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークの製品にインストールされているマニュアルの索引から入手できます。
`/opt/SUNWspro/docs/ja/index.html`

製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

- マニュアルは、`docs.sun.com` の Web サイトで入手できます。以下に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (`docs.sun.com` Web サイトでは入手できません)。
 - 『Standard C++ Library Class Reference』
 - 『標準 C++ ライブラリ・ユーザズガイド』
 - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
 - 『Tools.h++ ユーザズガイド』
- リリースノートは、`docs.sun.com` で入手できます。

インターネットの `docs.sun.com` Web サイト (<http://docs.sun.com>) から、サンのマニュアルを参照したり、印刷したり、購入することができます。マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 - Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。**Sun** は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは以下の表に示す場所から参照することができます。製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアル (サードパーティ製マニュアルは除く)	形式 : HTML (日本語版は PDF のみ) 場所 : http://docs.sun.com
サードパーティ製マニュアル: 『Standard C++ Library Class Reference』 『標準 C++ ライブラリ・ユーザーズガイド』 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 『Tools.h++ ユーザーズガイド』	形式 : HTML 場所 : <code>file:/opt/SUNWspro/docs/ja/index.html</code> のマニュアル索引
Readme および マニュアル ページ	形式 : HTML 場所 : <code>file:/opt/SUNWspro/docs/ja/index.html</code> のマニュアル索引
リリースノート	製品 CD 内の HTML ファイル

関連する Solaris マニュアル

次の表では、docs.sun.com の Web サイトで参照できる関連マニュアルについて説明します。

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris 8 Reference Manual Collection	マニュアルページの節を参照。	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris 8 Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris 8 Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。

開発者向けのリソース

<http://www.sun.com/developers/studio> にアクセスし、**Compiler Collection** というリンクをクリックして、以下のようなリソースを利用できます。リソースは頻繁に更新されます。

- プログラミング技術と最適な演習に関する技術文書
- プログラミングに関する簡単なヒントを集めた知識ベース
- **Compiler Collection** のコンポーネントのマニュアル、ソフトウェアと共にインストールされるマニュアルの訂正
- サポートレベルに関する情報
- ユーザーフォーラム
- ダウンロード可能なサンプルコード

- 新しい技術の紹介
- <http://www.sun.co.jp/developers/> でも開発者向けのリソースが提供されています。

技術サポートへの問い合わせ

製品についての技術的なご質問がございましたら、以下のサイトからお問い合わせください (このマニュアルで回答されていないものに限りです)。

<http://sun.co.jp/service/contacting>

第1章

概要

SPARC[®]、x86 システムで、サン・マイクロシステムズ社の浮動小数点環境を利用すると、正確、高性能、かつ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。また、この環境は、他のユーザーが作成した数値計算プログラムが異常な動作をした場合に原因を究明するのに便利です。これらのシステムは、IEEE 規格 754 (2 進浮動小数点演算) で規定された演算モデルを実装しています。このマニュアルは、これらのシステム上で IEEE 規格によって可能となったオプション、および柔軟性について、その使用方法を説明しています。

浮動小数点環境

浮動小数点環境は、データ構造、アプリケーションプログラマが IEEE 規格 754 を実装したハードウェア、ソフトウェア、およびソフトウェアライブラリから構成されています。IEEE 規格 754 を使用することによって、数値計算アプリケーションの作成がさらに簡単になります。IEEE 規格 754 は、数値計算プログラミングの作成において利用するコンピュータ算術のすでに確立された基礎となるものです。

たとえば、ハードウェアは IEEE データ形式に対応する記憶装置書式、IEEE 形式のデータに関する演算、これらの演算が生成する結果の丸めの制御、IEEE 数値例外の発生を知らせるステータスフラグ、およびこのような例外の発生に対してユーザーがハンドラを定義していない場合の IEEE 規定の結果などを提供します。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、式の生成を考慮した IEEE 規格 754 に従った方法で、 $\exp(x)$ 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は、システムは例外を発生してユーザーに伝えます)。また、数学ライブラリは Inf (無限大) や NaN (非数) のような特別な IEEE の値を返す関数呼び出しも行います。

浮動小数点環境では上記の 3 つの要素は微妙に影響し合いますが、一般的にはこれらの相互作用はアプリケーションプログラマには見えません。プログラマは、IEEE 標準に規定され、推奨された計算メカニズムだけを見ることができます。一般には、このマニュアルではプログラマが有効に IEEE メカニズムを使用し、アプリケーションを効果的に作成することを目標としています。

浮動小数点に関する質問には、基本的な演算に関するものがあります。たとえば、次のような質問です。

- 無限に正確な結果がコンピュータシステムで表現できない時、演算の結果はどうなるか？
- 乗算、加算のような繰り返し演算は？

その他の質問は、例外や例外処理に関連しています。たとえば、次のような問題が発生した場合です。

- 極めて大きい 2 つの数の乗算
- ゼロによる除算
- 負の数の平方根を求めようとした

特別な分野での算術では、基本的演算に関する質問にも期待するような解答が得られないかもしれません。また、例外処理に関する疑問についても、基本的演算の場合と同じと言わざるを得ません。プログラムがただちに終了するか、あるいは古いマシンでは計算を続行し、誤った結果を出すことになります。

IEEE 規格 754 は、演算で数学的に期待した結果をもたらし、数学的に役立つ演算特性を提供します。また、ユーザーが特別にその他の選択をしなければ、例外におけるケースでは、確実に指定した結果 (デフォルトの結果) が得られます。

このマニュアルでは、NaN や非正規数など、馴染みのうすい用語が使用されているかもしれません。浮動小数点演算に関する用語については、「用語集」で定義しています。

第2章

IEEE 演算機能

この章では、ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (「IEEE 規格」または「IEEE 754」と略される) が規定する演算モデルについて説明します。SPARC[®]、x86 のコンピュータはすべて、IEEE 演算機能を使用しています。サンのすべてのコンパイラ製品は、IEEE 演算機能をサポートしています。

IEEE 演算モデル

この節では、IEEE 754 について説明します。

IEEE の演算機能

IEEE 754 は、以下のことを規定しています。

- **単精度と倍精度の 2 つの基本浮動小数点形式。**

IEEE 単精度形式は、24 ビットの有効数字精度で、全体の大きさは 32 ビットです。

IEEE 倍精度形式は、53 ビットの有効数字精度で、全体は 64 ビットです。

- **拡張単精度と拡張倍精度の 2 つの拡張浮動小数点形式のクラス。**

IEEE 規格 はこれらの形式の精度と大きさについて正確には規定していませんが、最小の精度と大きさは定めています。たとえば、IEEE 拡張倍精度形式では、有効精度は最低 64 ビット、全体的には最低 79 ビットでなければなりません。

- 浮動小数点形式による加算、減算、乗算、除算、平方根、剰余、丸めの整数形式への代入、異なる浮動小数点形式間での変換、浮動小数点形式と整数形式間での変換、比較などの基本的な浮動小数点演算において要求される正確度。

剰余と比較の演算は正確でなければなりません。その他の各演算は、正確な結果が得られなかったり、その結果が結果先の形式に合っていない場合を除いて、結果先にその正確な結果を渡さなければなりません。結果が結果先の形式に合っていない場合、規定の丸めモードや以下に説明する規則に応じて正確な結果を最小限修正し、形式を合わせてその結果を渡さなければなりません。

- 基本的な浮動小数点形式で 10 進文字列と 2 進浮動小数点数間で変換を行う際の正確度、単調性、および同一性の要求。

指定した範囲内の引数に対する演算では、この変換は、可能な限り正確な結果を生み出します。そうでなければ、規定の丸めモードの規則に従って、正確な結果を最小限修正しなければなりません。指定した範囲外の引数に対する演算では、この変換による結果と正確な結果の差は、丸めモードに依存して指定された許容限度内にならなければなりません。

- 5 種類の IEEE 浮動小数点例外、およびこれらの例外の発生をユーザーに知らせる条件。

5 種類の浮動小数点例外とは、無効な演算、ゼロによる除算、オーバーフロー、アンダーフロー、不正確です。

- 4 つの丸めモード。最近の表現可能値がある時は、いつでも選択される “同一” の値を持つ最近似値の表現可能値への丸め、ゼロ方向への丸め、 $+\infty$ 方向への丸め、および $-\infty$ 方向への丸め。
- 丸め精度。たとえば、システムが拡張精度形式だけで結果を返す場合、結果を単精度形式と倍精度形式のいずれかの精度に丸めるように、ユーザーが指定できるようにする必要があります。

IEEE 規格は、ユーザー定義の例外のサポートも提唱しています。

IEEE 規格が規定する機能により、区間演算、例外の遡及 (そきゅう) 診断、exp や cos のような標準的な基本関数の効率的な実装、多倍精度演算といった、数値計算に便利なさまざまなツールのサポートが可能になります。

IEEE 754 の浮動小数点演算は、他のいかなる浮動小数点演算よりも、優れた数値計算の可制御性を提供します。IEEE 規格は、実装時に厳格な要求を課すことにより、数学的に洗練された移植可能なプログラムを作成する作業を容易にするだけではありません。この規格は、これよりも拡張したり、洗練した実装も認めています。

IEEE 形式

この節では、浮動小数点データをメモリーに記憶する方法について説明します。また、各 IEEE 記憶形式の精度および範囲を示します。

記憶形式

浮動小数点形式とは、浮動小数点数を表わす数値フィールド、フィールドの配置、およびその解釈からなるデータ構造です。浮動小数点の**記憶形式**は、浮動小数点数をメモリーに記憶する形式を指定します。IEEE 標準はその書式を定義していますが、記憶形式の選択は実装者に任されています。

アセンブリ言語ソフトウェアは使用する記憶形式に依存しているものもありますが、高級言語の場合は通常、浮動小数点のデータ型の言語で表記できる部分だけを処理します。これらの型は、高級言語によって異なる名前を持ち、表 2-1 に示すように IEEE 形式に対応します。

表 2-1 IEEE 形式と言語の種類

IEEE 精度	C、C++	FORTAN (SPARC のみ)
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	REAL*16

IEEE 754 は、単精度と倍精度の浮動小数点形式を規定しており、これらの基本的な 2 つの形式に対して、それぞれ拡張形式のクラスを定義しています。表 2-1 にある long double と REAL*16 という言語の種類は、IEEE 標準で定義された拡張倍精度形式の 1 クラスです。

次の節では、SPARC、x86 プラットフォームで IEEE 浮動小数点形式に使用される 3 つの記憶形式について詳しく説明します。

単精度の記憶形式

IEEE 単精度記憶形式は、23 ビットの小数部 f 、8 ビットの指数 e 、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。図 2-1 に示すように、これらのフィールドは、32 ビットワードを 1 つとして隣接して格納されます。ビット 0:22 には 23 ビット小数部 f が含まれます。ビット 0 が小数部の最下位ビット、ビット 22 が最上位ビットです。ビット 23:30 は 8 ビットのバイアス指数 e となり、ビット 23 はバイアス指数の最下位ビット、ビット 30 は最上位になります。最高位のビット 31 には符号ビット s が含まれます。

s	e (指数) [30:23]	f (小数部) [22:0]
31	30	23 22
		0

図 2-1 単精度の記憶形式

表 2-2 では、 s 、 e 、 f の 3 つのフィールドにある値と、単精度の記憶形式ビットパターンで表示される値との対応を示しています。 u は無意味です。つまり、示されたフィールドの値は、単精度記憶形式のこのビットパターンの値の決定には影響を及ぼさないということです。

表 2-2 IEEE 単精度記憶形式 のビットパターンで表現した値

単精度ビットパターン	値
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$ (正規数)
$(e = 0; f \neq 0)$ (f にあるビットのうち最低 1 つはゼロ以外)	$(-1)^s \times 2^{-126} \times 0.f$ (非正規数)
$e = 0; f = 0$ (f にあるビットはすべてゼロ)	$(-1)^s \times 0.0$ (符号付きの 0)
$s = 0; e = 255; f = 0$ (f にあるビットはすべてゼロ)	+INF (正の無限大)
$s = 1; e = 255; f = 0$ (f にあるビットはすべてゼロ)	-INF (負の無限大)
$s = u; e = 255; f \neq 0$ (f にあるビットのうち最低 1 つはゼロ以外)	NaN (非数)

表 2-3 単精度記憶形式でのビットパターンとその IEEE 値 (続き)

名前	ビットパターン (16 進形式)	対応する値
$+\infty$	7f800000	+INF (正の無限大)
$-\infty$	ff800000	-INF (負の無限大)
非数	7fc00000	NaN (数字以外)

NaN (非数) は、NaN の定義を満たすビットパターンのどれでも表現できます。表 2-3 に示した NaN の 16 進数値は、NaN を表現できる何種類ものビットパターンのうちの 1 つだけです。

倍精度の記憶形式

IEEE 倍精度値は、52 ビットの小数部 f 、11 ビットの指数 e 、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。9 ページの図 2-2 に示すように、これらのフィールドは、続けてアドレス指定した 2 つの 32 ビットワードに隣接して格納されます。

SPARC アーキテクチャでは、上位アドレス 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。一方 x86 および PowerPC アーキテクチャでは、下位アドレス 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。

$f[31:0]$ を小数の最下位 32 ビットで表わすと、ビット 0 は小数全体の最下位ビットになり、ビット 31 は 32 最下位小数ビットの最上位になります。

その他の 32 ビットワードでは、ビット 0:19 には小数部 $f[51:32]$ の 20 の最上位ビットが含まれます。ビット 0 はこれらの 20 の最上位小数ビットの最下位になり、ビット 19 は小数全体の最上位ビットになります。ビット 20:30 は 11 ビットのバイアス指数 e で、ビット 20 はバイアス指数の最下位ビット、ビット 30 が最上位になります。最高位のビット 31 には符号ビット s が含まれます。

図 2-2 では、2 つの隣接する 32 ビットワードを介したビット数は、1 つの 64 ビットワードとなり、ビット 0:51 は 52 ビット小数部 f 、ビット 52:62 は 11 ビットバイアス指数 e 、ビット 63 は符号ビット s をそれぞれ格納します。

倍精度の記憶形式について、正規数と非正規数の相違点は、正規数の場合仮数の先行ビット (2 進小数点の左にあるビット) は 1 であるのに対して、非正規数の場合は 0 であることです。倍精度記憶形式の非正規数は、IEEE 規格 754 では倍精度記憶形式のデノーマル数と呼ばれていました。

暗黙の先行仮数ビットと結合された 52 ビット小数部は、倍精度形式の正規数で 53 ビットの精度を提供します。

倍精度の記憶形式における重要なビットパターンの例を表 2-5 に示します。2 カラム目のビットパターンは、8 桁の 16 進数で表わされます。SPARC アーキテクチャでは、左は下位アドレス指定の 32 ビットワードの値、右は上位アドレス指定の 32 ビットワードの値になります。一方 x86 アーキテクチャでは、左が上位アドレスのワード、右が下位アドレスのワードになります。正の最大正規数とは、IEEE 倍精度形式で表現可能な最大の有限数です。正の最小正規数とは、IEEE 倍精度形式で、精度を落とさずに表現可能な正の最小数です。最大非正規数とは、IEEE 倍精度形式で表現可能な最大数です。正の最小非正規数とは、IEEE 倍精度形式で表現可能な正の最小数です。正の最小正規数は、アンダーフローしきい値とも呼ばれます。正の最大および最小の正規数および非正規数は、精度が失われる可能性があります。値は、以下の表のようになります。

表 2-5 倍精度記憶形式でのビットパターンとその IEEE 値

名前	ビットパターン (16 進)	対応する値
+ 0	00000000 00000000	0.0
- 0	80000000 00000000	-0.0
1	3ff00000 00000000	1.0
2	40000000 00000000	2.0
最大正規数	7fefffff ffffffff	1.7976931348623157e+308
正の最小正規数	00100000 00000000	2.2250738585072014e-308
最大非正規数	000fffff ffffffff	2.2250738585072009e-308
正の最小非正規数	00000000 00000001	4.9406564584124654e-324
+∞	7ff00000 00000000	無限大
-∞	fff00000 00000000	- 無限大
非数	7ff80000 00000000	NaN

NaN (非数) は、NaN の定義を満たすビットパターンのもれでも表現できます。表 2-5 に示した NaN の 16 進数値は、NaN を表わすのに使用できる多くのビットパターンのうちの一例です。

拡張倍精度の記憶形式 (SPARC)

浮動小数点環境の 4 倍精度記憶形式は、IEEE 定義の拡張倍精度の記憶形式に準拠します。4 倍精度記憶形式では、4 つの 32 ビットワードを占有します。112 ビット小数部 f、15 ビットバイアス指数部 e、1 ビット符号 s の 3 つのフィールドから構成されています。図 2-3 に示すように、隣接して格納されています。

最高位アドレスの 32 ビットワードには、小数部 f[31:0] の最下位 32 ビットがあります。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] が含まれます。次のワードのビット 0:15 には、小数部 f[111:96] の 16 最上位ビットが含まれ、ビット 0 はこれら 16 ビットの最下位になり、ビット 15 は小数部全体の最上位になります。ビット 16:30 には 15 ビットバイアス指数部 e が含まれ、ビット 16 はバイアス指数部の最下位、ビット 30 は最上位になります。また、ビット 31 には符号ビット s が含まれます。

図 2-3 では、4 つの隣接する 32 ビットワードを介したビット数は、1 つの 128 ビットワードとなり、ビット 0:111 は小数部 f、ビット 112:126 は 15 ビットバイアス指数 e、ビット 127 は符号ビット s をそれぞれ格納します。

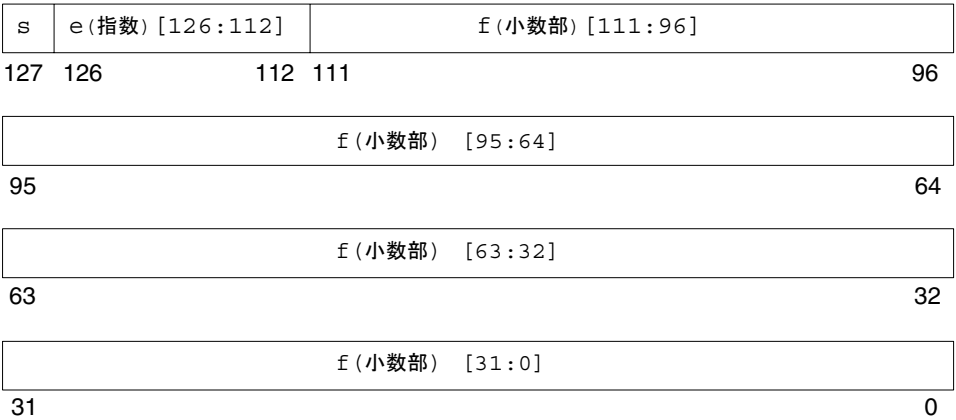


図 2-3 拡張倍精度の記憶形式 (SPARC)

これら f 、 e 、 s の 3 つのフィールドにあるビットパターンの値は、全体のビットパターンによって表現される値を決定します。

表 2-6 では、 s 、 e 、 f の 3 つのフィールドにある値と、4 倍精度の記憶形式ビットパターンで表示される値との対応を示しています。 u は無意味です。つまり、示されたフィールドの値は、拡張倍精度記憶形式のこのビットパターンの値の決定には影響を及ぼさないということです。

表 2-6 IEEE 拡張倍精度記憶形式のビットパターンで表現した値 (SPARC)

拡張倍精度ビットパターン	値
$0 < e < 32767$	$(-1)^s \times 2^{e-16383} \times 1.f$ (正規数)
$e = 0$ 、 $f \neq 0$ (f のビットのうち最低 1 つはゼロ以外)	$(-1)^s \times 2^{-16382} \times 0.f$ (非正規数)
$e = 0$ 、 $f = 0$ (f のすべてのビットがゼロ)	$(-1)^s \times 0.0$ (符号付きのゼロ)
$s = 0$ 、 $e = 32767$ 、 $f = 0$ (f のすべてのビットがゼロ)	$+\text{INF}$ (正の無限大)
$s = 1$ 、 $e = 32767$ 、 $f = 0$ (f のすべてのビットがゼロ)	$-\text{INF}$ (負の無限大)
$s = u$ 、 $e = 32767$ 、 $f \neq 0$ (f のビットのうち最低 1 つはゼロ以外)	NaN (非数)

4 倍精度の記憶形式における重要なビットパターンの例を表 2-7 に示します。第 2 カラムのビットパターンは、8 桁の 16 進数で表わされています。一番左の数字が最低位アドレスの 32 ビットワードを、一番右の数字が最高位アドレスの 32 ビットワードを示しています。正の最大正規数とは、拡張倍精度形式で表現可能な最大の有限数です。正の最小正規数とは拡張倍精度形式で、精度を落とさずに表現可能な正の最小数です。最大非正規数とは、拡張倍精度形式で表現可能な最大数です。正の最小非正規数とは、拡張倍精度形式で表現可能な正の最小数です。正の最小正規数は、アンダーフローしきい値ともよばれます。正の最大および最小の正規数および非正規数は、精度が失われる可能性があります。値は、以下の表のようになります。

表 2-7 拡張倍精度記憶形式でのビットパターンとその値 (SPARC)

名前	ビットパターン (SPARC)				対応する値
+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
最大正規数	7ffefffff	ffffffffff	ffffffffff	ffffffffff	1.1897314953572317650857593266280070e+4932
正の最小正規数	00010000	00000000	00000000	00000000	3.3621031431120935062626778173217526e-4932
最大非正規数	0000ffff	ffffffffff	ffffffffff	ffffffffff	3.3621031431120935062626778173217520e-4932
正の最小非正規数	00000000	00000000	00000000	00000001	6.4751751194380251109244389582276466e-4966
+ ∞	7fff0000	00000000	00000000	00000000	+ ∞
- ∞	ffff0000	00000000	00000000	00000000	- ∞
非数	7fff8000	00000000	00000000	00000000	NaN

表 2-7 に示した NaN の 16 進数値は、NaN を表現できる何種類ものビットパターンのうちの 1 つにすぎません。

拡張倍精度の記憶形式 (x86)

浮動小数点環境の拡張倍精度記憶形式は、IEEE 定義の拡張倍精度の記憶形式に準拠します。63 ビット小数部 f 、1 ビットの明示的先行仮数ビット j 、15 ビットバイアス指数 e 、1 ビット符号 s の 4 つのフィールドから構成されています。

x86 アーキテクチャファミリでは、これらのフィールドはアドレス指定した 8 ビットバイトが 10 個連続して格納されます。しかし、UNIX System V Application Binary Interface Intel 386 Processor Supplement (Intel ABI) では、倍拡張パラメータと結果がスタックで 3 つ連続した 32 ビットワードを占有し、14 ページの図 2-4 にあるように使用されていない最高位アドレスのワードの最上位 16 ビットを持つようにしてください。

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、ビット 0 は小数部全体の最下位ビットになり、ビット 31 は 32 最下位小数ビットの最高位になります。中央のアドレス 32 ビットワードでは、ビット 0:30 には小数部

f[62:32] の 31 最高位ビットが含まれ、ビット 0 はこれら 31 最高位小数ビットの最下位になり、ビット 30 は小数部全体の最高位になります。この中央アドレスの 32 ビットワードのビット 31 には明示的な先行仮数ビット j が含まれます。

最高アドレスの 32 ビットワードでは、ビット 0:14 に 15 ビットバイアス指数部 e が含まれており、ビット 0 はバイアス指数部の最下位ビット、ビット 14 は最高位ビットになります。また、ビット 15 には符号ビット s が含まれます。この最高アドレス 32 ビットワードの最高位 16 ビットは x86 アーキテクチャファミリでは未使用ですが、この存在は上記で説明したように、Intel ABI への準拠には不可欠です。

図 2-4 では、3 つの隣接する 32 ビットワードを介したビット数は、1 つの 96 ビットワードとなり、ビット 0:62 は 63 ビット小数部 f、ビット 63 は明示的な先行仮数ビット j、ビット 64:78 は 15 ビットバイアス指数部 e、ビット 79 は符号ビット s をそれぞれ格納します。

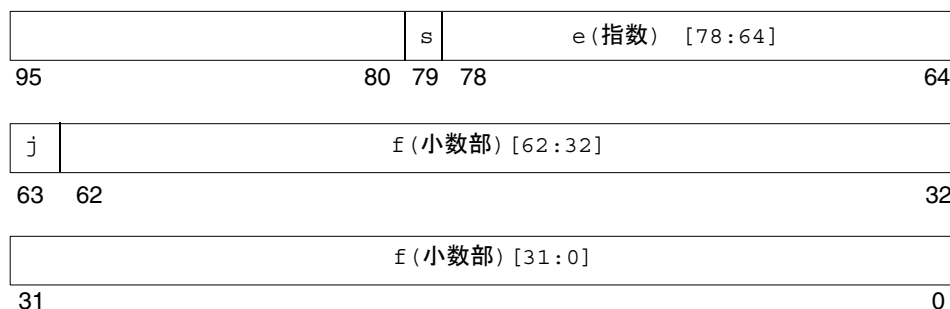


図 2-4 拡張倍精度の記憶形式 (x86)

これら f、e、s、j の 4 つのフィールドにあるビットパターンの値は、全体のビットパターンによって表現される値を決定します。

表 2-8 は、隣接する 4 つのフィールドのカウントした数値と、ビットパターンで表現される値との対応を示しています。 u は無意味です。つまり、示されたフィールドの値は、拡張倍精度記憶形式のこのビットパターンの値の決定には影響を及ぼさないということです。

表 2-8 IEEE 拡張倍精度記憶形式のビットパターンで表現した値 (x86)

拡張倍精度ビットパターン (x86)	値
$j = 0, 0 < e < 32767$	サポートしない
$j = 1, 0 < e < 32767$	$(-1)^s \times 2^{e-16383} \times 1.f$ (正規数)
$j = 0, e = 0; f \neq 0$ (f にあるビットのうち最低 1 つはゼロ以外)	$(-1)^s \times 2^{-16382} \times 0.f$ (非正規数)
$j = 1, e = 0$	$(-1)^s \times 2^{-16382} \times 1.f$ (偽の正規数)
$j = 0, e = 0, f = 0$ (f にあるすべてのビットはゼロ)	$(-1)^s \times 0.0$ (符号付きのゼロ)
$j = 1; s = 0; e = 32767; f = 0$ (f にあるすべてのビットはゼロ)	+INF (正の無限大)
$j = 1; s = 1; e = 32767; f = 0$ (f にあるすべてのビットはゼロ)	-INF (負の無限大)
$j = 1; s = u; e = 32767; f = .1uuu - uu$	QNaN (シグナルを発生しない NaN)
$j = 1; s = u; e = 32767; f = .0uuu - uu \neq 0(f$ にある u のうち最低 1 つはゼロ以外)	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は、暗黙的な先行仮数ビットを持たないので注意してください。先行仮数ビットは、拡張倍精度形式では、別のフィールド j として明示的に与えられます。しかし、 $e \neq 0$ のときは、このようなビットパターンを浮動小数点演算の演算子として使用すると、無効な演算例外が発生するという意味で、 $j = 0$ をもつパターンはどれもサポートされません。

拡張倍精度形式での連帯しない j と f の結合を**仮数**と呼びます。 $e < 32767$ および $j = 1$ 、あるいは $e = 0$ および $j = 0$ のとき、先行仮数ビット j と小数部の最上位ビットの間に小数点を挿入して仮数は形成されます。

x86 の拡張倍精度形式では、先行仮数ビット j が 0 でありバイアス指数部フィールド e も 0 であるビットパターンは非正規数を表現します。一方、先行仮数ビット j が 1 でありバイアス指数部フィールド e が 0 以外であるビットパターンは、正規数を表現します。先行仮数ビットは、指数の値から推定されるのではなく明示的に表現される

ため、この形式は非正規数のようにバイアス指数部が 0 であるが先行仮数ビットが 1 であるビットパターンも認めます。このようなビットパターンはそれぞれ、実際には、バイアス指数部フィールドが 1 である対応するビットパターン (つまり正規数) と同じ値を表現します。そのため、これらのビットパターンは「疑似デノーマル」と呼ばれます (非正規数は IEEE 規格 754 でデノーマル数と呼ばれていました)。疑似デノーマルは、x86 拡張倍精度形式をコード化した結果にすぎず、オペランドとして示されるときには対応する正規数に暗黙的に変換されます。疑似デノーマルが結果として生成されることはありません。

拡張倍精度の記憶形式における重要なビットパターンの例を表 2-9 に示します。2 カラム目のビットパターンは、4 桁の 16 進数で表わされます。この数は最上位アドレス 32 ビットワードの下位 16 ビットです (この最高位アドレス 32 ビットワードの下位 16 ビットの再呼び出しは使用されないので、その値は表示されません)。これに 8 桁の 16 進のカウント数が 2 つ続きます。この左側は中央のアドレス 32 ビットワードの値、右側は下位アドレスの 32 ビットワードの値になります。正の最大正規数とは、x86 拡張倍精度形式で表現可能な最大の有限数です。正の最小正規数とは拡張倍精度形式で、精度を落とさずに表現可能な正の最小数です。最大非正規数とは、拡張倍精度形式で表現可能な最大数です。正の最小非正規数とは、拡張倍精度形式で表現可能な正の最小数です。正の最小正規数は、アンダーフローしきい値ともよべれます。正の最大および最小の正規数および非正規数は、精度が失われる可能性があります。値は、以下の表のようになります。

表 2-9 拡張倍精度記憶形式でのビットパターンとその値 (x86)

名前	ビットパターン (x86)		対応する値
+0	0000	00000000 00000000	0.0
-0	8000	00000000 00000000	-0.0
1	3fff	80000000 00000000	1.0
2	4000	80000000 00000000	2.0
最大正規数	7ffe	fffffffe ff	1.18973149535723176505e+4932
正の最小正規数	0001	80000000 00000000	3.36210314311209350626e-4932
最大非正規数	0000	7fffffff ff	3.36210314311209350608e-4932
正の最小非正規数	0000	00000000 00000001	3.64519953188247460253e-4951
+∞	7fff	80000000 00000000	+∞
-∞	ffff	80000000 00000000	-∞

表 2-9 拡張倍精度記憶形式でのビットパターンとその値 (x86)

名前	ビットパターン (x86)	対応する値
最大小数部のあるシグナルを発生しない NaN	7fff ffffffff ffffffff	QNaN
最小小数部のあるシグナルを発生しない NaN	7fff c0000000 00000000	QNaN
最大小数部のあるシグナルを発生する NaN	7fff bfffffff ffffffff	SNaN
最小小数部のあるシグナルを発生する NaN	7fff 80000000 00000001	SNaN

NaN (非数) は、NaN の定義を満たすビットパターンのどれでも表現できます。表 2-9 に示した NaN の 16 進数値は、小数フィールドの先行 (たとえば最上位) ビットが NaN はシグナルを発生しないか (先行小数ビットは 1)、発生するか (先行小数ビットは 0) を決定するポイントを示しています。

10 進表現の範囲と精度

この節では、指定された記憶形式の範囲と精度の表記法について簡単に説明します。IEEE の単精度形式、倍精度形式、SPARC、x86 アーキテクチャでの IEEE 拡張倍精度形式の実装に準拠する範囲と精度について説明します。具体的には、範囲と精度の表記法に関する説明で IEEE の単精度形式を取りあげます。

IEEE 規格では、単精度形式で浮動小数点を表わす場合は 32 ビットを使用することを規定しています。32 個の 0 と 1 の組合せには限りがあるので、32 ビットで表現できる数だけが使用されることになります。

そこで次のような疑問がおこります。

「この特定の形式で表わすことができる最大の正の数と最小の正の数を 10 進数で表現したらどうなるか」

この疑問を次のように言い換えて、範囲の概念を説明します。

「IEEE 単精度形式で表現できる数の範囲を 10 進数で表現するとどうなるか」

IEEE の単精度形式の厳密な定義をふまえると、IEEE の単精度形式で表現できる浮動小数点数の範囲 (正の正規数に限る) は以下の通りです。

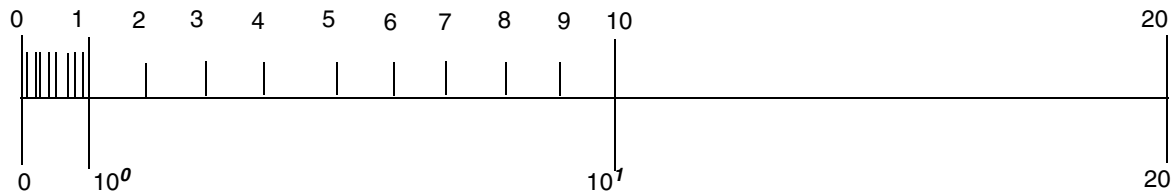
1.175...x (10⁻³⁸) から 3.402...x (10⁺³⁸) まで

次に問題になるのは、指定された形式で表現する数の精度 (正確度または有効数字と混同させないため) です。この表記法については、図と例を提示して説明します。

2 進浮動小数点演算の IEEE 標準は、単精度形式で表現できる数値の集合を規定します。この数値の集合は、2 進浮動小数点数の集合として説明されていたことを思い出してください。IEEE 単精度形式の仮数は 23 ビットで、暗黙の先行ビットと結合し、(2 進) 精度の 24 桁 (ビット) になります。

一方、以下の数直線のように数 $x = (x_1 x_2 x_3 \dots x_q) \times (10^n)$ (仮数 q は 10 進数字で表現可能な数) にマーキングをして違う数値の集合を取得します。

10進表現



2進表現

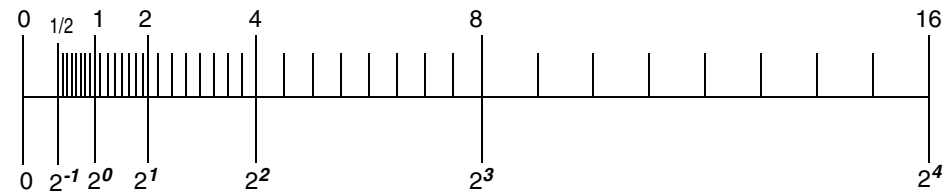


図 2-5 10 進と 2 進表現で定義された数の比較

この 2 つの集合は別のものなので注意してください。そのため、2 進数の仮数 24 に対応する仮数 10 進数を算定すると、問題を再度まとめるように求められます。

2 進表現 (コンピュータが使用する内部形式) と 10 進形式間の浮動小数点数の変換という見地で、問題をもう一度明確にしてみます。ここでは、2 進から 10 進へ変換して 2 進に戻すとともに、10 進から 2 進に変換して 10 進に戻してみます。

数値の集合はそれぞれ違うので、一般に変換は不正確だということを認識することが必要です。正しく変換が行われた場合、ある集合の数値から別の集合の数値に変換すると、変換先のセットから隣接した 2 数のうち 1 つを選択する結果になります。

最初にいくつか例をみてみます。たとえば、次の 10 進形式の数を IEEE 単精度形式で表現するとします。

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

IEEE の単精度形式で正確に表現できる実数の数には限度があり、上記形式によるすべての数値がそれらに含まれるわけではないため、通常はそのような数値を正確に表現することは不可能です。次の例を考えてみます。

$$y = 838861.2, z = 1.3$$

ここで、以下の 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、次のようになります。

```
Y: 8.38861187500E+05
Z: 1.29999995232E+00
```

y に代入された値 8.388612×10^5 と出力された値の差は、0.000000125 です。すなわち、 y よりも小数点以下 7 桁分少なくなっています。この場合、 y を IEEE の単精度形式で表わしたときの正確度は、約 6 桁または 7 桁の有効数字です。言い換えると、 y を IEEE の単精度形式で表現した場合の有効数字は約 6 桁になります。

同様に、 z に代入された値 1.3 と出力された値の差は、0.00000004768 です。すなわち、 z よりも小数点以下 8 桁分少なくなっています。この場合、 z を IEEE 単精度形式で表わしたときの正確度は、約 7 桁または 8 桁の有効数字です。つまり、 z を IEEE の単精度形式で表現した場合の有効数字は約 7 桁になります。

この問題を公式化してみます。

「10 進数の浮動小数点数 a を IEEE の単精度形式 2 進表現 b に変換し、さらに b を 10 進数の c に変換すると、 a と $a - c$ の差は何桁分になるか。」

この問題は次のように言い換えることができます。

「IEEE 単精度形式 a の有効な **10 進桁**はいくつか。言い換えると、 x を IEEE 単精度形式で表わした場合、何桁の 10 進数を正確であるとみなすべきか。」

有効 10 進数の桁数は、常に 6 から 9 です。すなわち、正確なのは 6 桁以上 9 桁以下です (変換が正確であり、無限大の桁が正確であるとみなす場合を除きます)。

言い換えると、IEEE 単精度形式で 2 進数を 10 進数に変換して、それをまた 2 進数に戻した場合、一般的にはこれらの 2 回の変換を行なった後に最初の数を得るためには、最低 9 桁の 10 進数が必要です。

値は表 2-10 のようになります。

表 2-10 各記憶形式の範囲と精度

形式	有効数字		最大の正の数	有効数字の 10 進表現
	2 進表現	最小の正の正規数		
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
拡張倍精度 (SPARC)	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

Solaris 環境における基数変換

C の `printf` や `scanf`、FORTRAN の `read`、`write`、`print` のような入出力ルーチンでは、基数変換が使用されます。これらの関数では、2 進と 10 進の数値表現の間で変換が必要です。

- 10 進から 2 進への基数変換は、従来の 10 進法の数値を読み取る場合と、それを内部的な 2 進形式で保存する場合に起きます。
- 2 進から 10 進への基数変換は、内部的な 2 進値を 10 進数の ASCII 文字列として出力する場合に起きます。

Solaris 環境では、標準 C ライブラリの `libc` にすべての言語における基数変換のための基本ルーチンが含まれています。これらのルーチンは、任意の入力形式と出力形式の間で正しい丸め変換を行うテーブル駆動型のアルゴリズムを使用します。テーブル駆動型のアルゴリズムは正確であるうえに、正しく丸められた基数変換が最悪のケースになってしまう回数を減らすことができます。

IEEE 規格は、絶対値が $10^{-44} \sim 10^{+44}$ である典型的な数値の場合、正確な丸めを要求しますが、比較的大きな指数部の場合には多少の不正確な丸めも認めています (IEEE 規格 754 の 5.6 を参照)。libc のテーブル駆動型アルゴリズムは、単精度、倍精度、および拡張倍精度形式の全範囲について正しい丸めを行います。

基数変換に関する参照マニュアルについては、付録 F を参照してください。優れた文献として、特に Coonen の論文と Sterbenz の書籍をお勧めします。

C では、10 進数文字列とバイナリ浮動小数点値間の変換は、IEEE 754 に従って、通常は正しく丸められます。変換結果は、現在の丸めモードによって指定された方向の元の値にもっとも近い結果のフォーマットで表現できます。丸めモードが、最近似値丸めモードで、元の値が結果フォーマット中の 2 つの表現可能な数字の間に正確に置かれている場合、変換結果は、最下位数字が偶数のものになります。これらの規則は、標準ライブラリルーチンを使用するプログラムによって実行されるデータ変換と同様に、コンパイラによって実行されるソースコード中の定数の変換に適用されます。

Fortran では、10 進数文字列とバイナリ浮動小数点値間の変換は、デフォルトでは、C の規則と同様に、正しく丸められます。入出力変換では、プログラムで `ROUNDING= specifier` を使用するか、または `-iorounding` フラグを指定してコンパイルすることにより、最近似値丸めモードで `round-ties-to-even` 規則が優先されます。詳細は、『Fortran ユーザーズガイド』および f95(1) マニュアルページを参照してください。

アンダーフロー

アンダーフローは、算術演算の結果が小さすぎるために、通常より大きな丸め誤差をおこさないと指定形式に格納できない場合に発生します。

アンダーフローしきい値

表 2-11 に、単精度、倍精度、および拡張倍精度の場合のアンダーフローしきい値を示します。

表 2-11 アンダーフローしきい値

宛先精度	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
拡張倍精度 (SPARC)	最小の正規数	3.3621031431120935062626778173217526e-4932
	最大の非正規数	3.3621031431120935062626778173217520e-4932
拡張倍精度 (x86)	最小の正規数	3.36210314311209350626e-4932
	最大の非正規数	3.36210314311209350590e-4932

正の非正規数は、最小の正規数とゼロの間にある数です。最小の正規数に近い 2 つの (正の) 小さい数の減算を行うと、非正規数が生じます。あるいは、最小の正規数を 2 で割ると、商は非正規数になります。

非正規数そのものは正規数より少ないビット数の精度ですが、非正規数が存在すると、小さい数を含む浮動小数点計算はより大きな精度が可能となります。数学的に正しい結果が正の最小正規数よりも小さい時に、(ゼロを答えとして返すのではなく) 非正規数を生成することは、段階的アンダーフローとして知られています。

このようなアンダーフローの結果を扱う方法はいくつかあります。以前は共通の方法として、結果をゼロにしていました。この方法は **Store 0** として知られ、IEEE 標準ができる以前はメインフレームのデフォルト値になっていました。

IEEE 規格 754 を起草した数学者およびコンピュータ設計者は、数個の代替方法を考案し、数学的に安定した解および、効率よく実装することができる標準を生成する必要性をどちらも満たすように改良しました。

IEEE 演算機能におけるアンダーフローの扱い

IEEE 規格 754 はアンダーフロー結果を扱う望ましい方法として、段階的アンダーフローを選択しています。この方法は、正規数と非正規数の 2 つの格納された値に対する表現を定義することになります。

正規浮動小数点数の IEEE 形式は次のようになっています。

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

s は符号ビット、 e はバイアス指数、 f は小数部です。数を完全に指定するには、 s 、 e 、および f を格納する必要があります。有効数字の暗黙の先行ビットは、正規数の場合 1 に定義されるので、格納する必要はありません。

格納できる最小の正の正規数は、最大の負の指数とオールゼロの小数部を持ちます。先行ビットを 1 でなくゼロにすれば、さらに小さい数でも収容することができます。倍精度形式の場合、小数部は 52 ビット長 (10 進で約 16 桁) なので、最小指数を 10^{-308} から 10^{-324} に拡張できます。これらは**非正規数**です。(アンダーフローした結果をゼロにフラッシュしないで) 非正規数を返すことが**段階的アンダーフロー**です。

非正規数が小さければ小さいほど、当然ながらゼロ以外のビット数は少なくなります。非正規数を生じる計算は、相対丸め誤差の限界が正規オペランドの計算と同じではありません。しかし、段階的アンダーフローで重要なのは、これを使用することが以下のことを意味している点です。

- アンダーフローした結果は、通常の丸め誤差から生じる精度よりも大きい絶対正確度を失うことはありません。
- 加算、減算、比較、剰余は、結果が非常に小さい場合に常に正確といえます。

非正規浮動小数点数の IEEE 形式は次のようになっています。

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

s は符号ビット、バイアス指数 e はゼロ、 f は小数部となります。暗黙の底 2 の累乗は正規形式の底より 1 大きく、小数部の暗黙の先行ビットはゼロ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利用すると、表現できる数の範囲をより小さくすることができます。値を疑わしいものにする**小ささ**ではなく、関連誤差です。非正規数を利用するアルゴリズムは、そうでないシステムより誤差の境界が小さくなります。次の節では、段階的アンダーフローの数学的正当性を示します。

段階的アンダーフローの利点

非正規数の目的は、それ以外の演算機能のように、アンダーフロー/オーバーフローを完全に避けることではありません。非正規数は、多様な計算 (代表的には、乗算の次に加算を行う場合) を配慮した上でアンダーフローを削除します。詳細について

は、James Demmel 著『Underflow and the Reliability of Numerical Software』および S.Linnainmaa 著『Combatting the Effects of Underflow and Overflow in Determining Real Roots of Polynomials』を参照してください。

演算に非正規数があると、トラップされないアンダーフロー (正確度が損なわれることを意味します) は、加算または減算では発生しません。このため、 x と y が 2 の因数の中であれば、 $x - y$ には誤差が生じません。これは、アルゴリズムの重要な場所で現在の精度を実質的に増すアルゴリズムでは絶対に必要なことです。

また、段階的アンダーフローは、アンダーフローによる誤差が通常の丸め誤差ほどひどくないことを意味しています。これは他のアンダーフロー処理方法よりもはるかに有力です。この事実は、段階的アンダーフローがもっとも正当化される理由の 1 つです。

段階的アンダーフローの誤差特性

浮動小数点の結果は、ほとんどの場合に丸められます。

$$\text{computed result} = (\text{true result}) \pm \text{roundoff}$$

丸めの最大サイズはどれくらいでしょうか。そのサイズを測る便利な方法の 1 つに「最後の場所の単位」(*unit in the last place*、*ulp* と略される) があります。標準表記における浮動小数点数の仮数の最下位ビットが**最後の場所**になります。このビットが表現する値 (たとえば、このビットを除いて表現がまったく同じである 2 つの数値間の絶対値の差) は、その数値の「最後の場所の単位」です。実際の結果を表現可能な最近似値に丸めることにより計算結果が取得される場合は、計算結果の最後の場所の単位の半分より丸め誤差が大きくなることはありません。つまり、IEEE 演算では、最近似の丸めモードによって計算結果は次のようになります。

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

ulp は相対量です。非常に大きな数値の場合、その *ulp* も同様に大きくなります。一方、小さな数値の場合にはその *ulp* は小さくなります。この関係は、*ulp* を関数として表現することにより明示化できます。 $\text{ulp}(x)$ は、浮動小数点数 x の最後の場所の単位を示します。

浮動小数点数の **ulp** は、その数値が表現される精度により異なります。たとえば、上記で説明した浮動小数点の 4 つの形式における **ulp(1)** の値は、次のようになります。

表 2-12 4 つの異なる精度における **ulp(1)**

精度	値
単精度	$\text{ulp}(1) = 2^{-23} \sim 1.192093\text{e-}07$
倍精度	$\text{ulp}(1) = 2^{-52} \sim 2.220446\text{e-}16$
拡張倍精度 (x86)	$\text{ulp}(1) = 2^{-63} \sim 1.084202\text{e-}19$
4 倍精度 (SPARC)	$\text{ulp}(1) = 2^{-112} \sim 1.925930\text{e-}34$

どのコンピュータ演算でも正確に表現できる数のごく限られています。数値が小さくなり、ゼロに近づくに従って、隣接した表現可能な数の差は狭まっていきます。逆に数値が大きくなるに従い、隣接した表現可能な数の差は広がっていきます。

たとえば、精度がわずか 3 ビットである 2 進演算を行なったとします。その場合、2 つの 2 の累乗値の間には図 2-6 に示す通り $2^3 = 8$ 個の表現可能な数があります。

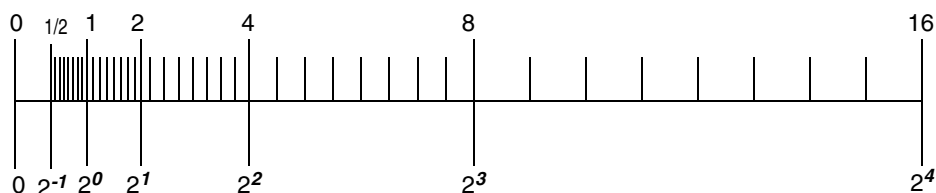


図 2-6 数直線

この数直線は、ある指数から次の指数までの数の差が、実質的に 2 倍になることを示しています。

IEEE 単精度形式の場合、2 つの最小の正の非正規数の差は約 10^{-45} ですが、2 つの最大の無限大数の差は約 10^{31} になります。

表 2-13 では、数直線上を正の無限大方向に移動していくときに、nextafter($x, +\infty$) が示す x の次に表現可能な数字を示します。

表 2-13 表現可能な単精度浮動小数点数の差

x	nextafter(x, $+\infty$)	差
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

従来の表現可能浮動小数点数の場合、不正確な結果の及ぼす最悪の影響は、計算結果の隣の表現可能数までの距離よりも大きい誤差を生じるという特性を持っていました。非正規数を表現可能数の集合に追加し、段階的アンダーフローを実装した場合、不正確な結果またはアンダーフロー結果の及ぼす最悪の影響は、計算結果の隣の表現可能数までの範囲で誤差を生じることです。

特にゼロと最小の正規数の間では、2 つの隣り合った数の差は、ゼロと最小の非正規数の差と等しくなります。非正規数があると、極限近似表現可能数までの範囲より大きい丸め誤差を生じる可能性をなくすることができます。

計算結果の表現可能な隣の数までの範囲を越えた丸め誤差を生じる計算がないため、確かな演算環境には、以下の 3 つの重要な特性があります。

- $x \neq y \Leftrightarrow x - y \neq 0$
- $(x-y) + y \approx x$ 、 x と y の大きい方の丸め誤差範囲内まで
- $1/(1/x) \approx x$ 、 x が正規数である場合、 $1/x \neq 0$ を意味する

代替のアンダーフロー機能は、Store 0 です (アンダーフロー結果をゼロにフラッシュします)。Store 0 は、 $x-y$ がアンダーフローすると 1 つめと 2 つめの特性に反します。また、 $1/x$ がアンダーフローすると、3 つめの特性に反します。

λ は、アンダーフローのしきい値でもある最小の正の正規数です。ここで、段階的アンダーフローと Store 0 の誤差特性を比較できます。

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

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

λ の最後の位置の $\frac{1}{2}$ 単位と、 λ の間には有意差があります。

段階的アンダーフローと Store 0 の 2 つの例

以下に、有名な数学的例題を 2 つ示します。最初の例は内積です。

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

段階的アンダーフローの場合、**result** は丸めと同程度に正確です。Store 0 の場合、小さな小計 (ゼロでない) を出すことができます。これは一見正しくみえますが、ほとんどの桁が間違っています。このような問題を解決するには、プログラマが詳細化によって正確度が低下すると予想できる場合、計算を各数値の比を保って何倍かにすることを認めなければなりません。

もう 1 つの例は、複素数の商を得る場合で、この場合は各数値の比を保って何倍かにする必要はありません。

$$a + i \cdot b = \frac{p + i \cdot q}{r + i \cdot s}, |r/s| \leq 1 \text{ と仮定する}$$

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

$p + i \cdot q$ と $r + i \cdot q$ がそれぞれわずかな **ulp** 以下の誤差を含む場合、丸めによる部分を考慮しても、計算結果の複素数と正しいと予想される結果には、誤差があることがわかります。誤差分析によると、 a と b の両方がアンダーフローである場合を除いて、アンダーフローがあっても誤差は $|a + i \cdot b|$ のわずかな **ulp** 以下になります。アンダーフローがゼロにフラッシュされると、どちらの例題も真ではなくなります。

複素数の商を計算するこのアルゴリズムは正確なので、段階的アンダーフローが発生しても、誤差分析を行う必要はありません。同様に Store 0 の場合、複素数の商を計算するための正確で分析しやすい有効なアルゴリズムはありません。Store 0 の場合、下位レベルの複雑で詳細な処理については、浮動小数点環境の実装側ではなくユーザー側に移行されています。

段階的アンダーフローが発生しても成功しますが、Store 0 を使うと失敗する問題の種類は、Store 0 の使用者が認識しているよりもたくさんあります。よく使用される数値演算は以下の種類に分けられます。

- 線形方程式
- 多項式
- 数値積分
- 収束加速
- 複素数除算

アンダーフローは問題か

これらの例にもかかわらず、アンダーフローがまれに問題となり、なぜ問題になるのか議論になることがあります。しかし、これは論理的には間違っています。

段階的アンダーフローの機能がないと、ユーザープログラムは暗黙の不正確なしきい値に対して注意を払わなければなりません。たとえば単精度の場合、計算のある部分でアンダーフローが発生し、アンダーフローした結果が Store 0 命令により 0 に置き換わると、正確度は、単精度指数の通常の低いほうの範囲 10^{-38} ではなく 10^{-31} 程度しか保証できません。

これは、不正確なしきい値に近づいたことを知る方法を実現するか、確かで安定したアルゴリズムの実現を放棄するか、どちらかを行わなければならないことを意味します。

ゼロ近くに収束した領域で計算が起こらないように、数値を大きくするアルゴリズムもあります。しかし、各数値計算プログラムにおいて、アルゴリズムをゼロ近くで計算が起こらないようにして、不正確なしきい値を検出する作業は困難であり、時間の浪費になります。

第3章

数学ライブラリ

この章では、Solaris オペレーティング環境と Sun ONE Studio Compiler Collection で提供されている数学ライブラリ `libm.a`、`libm.so`、`libsunmath.a` を含む数学関数の実装について説明します。この章では、各ライブラリの内容を示すとともに、このコンパイラコレクションに含まれる数学ライブラリがサポートする機能の一部 (IEEE をサポートする関数、乱数発生機構、IEEE と非 IEEE 形式間でデータを変換する関数など) について説明します。

`libm` と `libsunmath` ライブラリの内容は、Intro(3M) のマニュアルページにも挙げられています。

数学ライブラリ

`libm` 数学ライブラリには、Solaris オペレーティング環境が準拠している標準に必要な関数が含まれています。このライブラリは、静的ライブラリ `libm.a` と共有ライブラリ `libm.so` の2つの形式で Solaris に含まれています。

標準インストールをした場合の `libm` のデフォルトディレクトリは、次のとおりです。

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

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

標準インストールをした場合の `libm` のヘッダーファイルのデフォルトディレクトリは、次のとおりです。

```
/usr/include/floatingpoint.h
```

```
/usr/include/math.h
```

```
/usr/include/sys/ieeeefp.h
```

表 3-1 は、ライブラリの関数の一覧です。

表 3-1 libm の内容

型	関数名
代数関数	cbrt、hypot、sqrt
基本超越正関数	asin、acos、atan、atan2、acosh、asinh、atanh、exp、expm1、pow、log、log1p、log10、sin、cos、tan、sinh、cosh、tanh
高級超越正関数	j0、j1、jn、y0、y1、yn、erf、erfc、gamma、lgamma、gamma_r、lgamma_r
整数丸め関数	ceil、floor、rint
IEEE 規格で勧告の関数	copysign、fmod、ilogb、nextafter、remainder、scalbn、fabs
IEEE 分類の関数	isnan
旧式の浮動小数点関数	logb、scalb、significand
エラー処理関数	matherr

関数 `gamma_r` と `lgamma_r` は、`gamma` と `lgamma` の再入可能バージョンを意味するので注意してください。

動的リンクと静的リンク、およびプログラムの実行時に読み込む共有オブジェクトを決定するオプションと環境変数の詳細は、`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。

付加価値数学ライブラリ

Sun 数学ライブラリ

libsunmath ライブラリは、サンの言語製品と共に提供されるライブラリの一部です。libsunmath ライブラリには、サンの旧バージョンの libm に組み込まれていた関数が格納されています。

libsunmath がインストールされるデフォルトのディレクトリ

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

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

libsunmath のヘッダーファイルがインストールされるデフォルトのディレクトリ

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

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

次の表 3-2 は、libsunmath に含まれる関数を示しています。この表では、数学関数ごとに、C プログラムから呼び出される場合の、関数の倍精度名だけを示しています。

表 3-2 libsunmath の内容

型	関数名
30 ページの表 3-1 から の関数	matherr を除く単精度および拡張 / 4 倍精度の関数
基本超越正関数	exp2、exp10、log2、sincos
三角関数 (度数)	asind、acosd、atand、atan2d、sind、cosd、 sincosd、tand
π で基準化 (スケール) した三角関数	asinpi、acospi、atanpi、atan2pi、sinpi、cospi、 sincospi、tanpi
倍精度 π 引数還元のある三角関数	asinp、acosp、atanp、sinp、cosp、sincosp、tanp
財務関数	annuity、compound
整数丸め関数	aint、anint、rint、nint

表 3-2 libsunmath の内容 (続き)

型	関数名
IEEE 規格で勧告済みの関数	signbit
IEEE 分類の関数	fp_class、isinf、isnormal、issubnormal、iszero
有用な IEEE 値を与える関数	min_subnormal、max_subnormal、min_normal、max_normal、infinity、signaling_nan、quiet_nan
加法的乱数の生成	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_
線形合同乱数の生成	i_lcran_、i_lcrans_、i_init_lcrans_、i_get_lcrans_、i_set_lcrans_、r_lcran_、r_lcrans_、d_lcran_、d_lcrans_、u_lcrans_
桁上げ乗算乱数の生成	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_
乱数シャフル	i_shufrans_、r_shufrans_、d_shufrans_、u_shufrans_
データ変換	convert_external
制御丸めモード、および浮動小数点例外フラグ	ieee_flags
浮動小数点トラップ処理	ieee_handler、sigfpe
状態表示	ieee_retrospective
標準外演算の有効化 / 無効化	standard_arithmetic、nonstandard_arithmetic

最適化ライブラリ

一部の libm ルーチン中の最適化バージョンは、libmopt ライブラリによって提供されます。一部の libc 中のサポートルーチンは、libcopt ライブラリによって提供されます。また、SPARC® では、一部の libc サポートルーチンの代替は libcx によって提供されます。

標準インストールをした場合の libmopt、libcopt、libcx のデフォルトディレクトリは以下のとおりです。

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

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

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

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

(<arch> はアーキテクチャ固有のライブラリディレクトリを示す。SPARC では、v7、v8、v8a、v8plus、v8plusa、v8plusb、v9、v9a、v9b など。x86 プラットフォームでは、提供されるディレクトリは f80387 のみ。)

-xarch の詳細は、『Fortran ユーザーズガイド』、『C ユーザーズガイド』または『C++ ユーザーズガイド』を参照してください。

libcopt に含まれるルーチンは、ユーザーが直接呼び出すことはできません。その代わり、libc 内にサポートルーチンを置き換えて、コンパイラが使用できるようにします。

libmopt に含まれるルーチンは、libm 内の対応するルーチンに置き換えられます。libmopt バージョンの処理速度は、一般的に著しく速くなります。これは、libm バージョンは、ANSI/POSIX、SVID、X/Open、または IEEE 形式の例外のケースを処理するために構成されますが、libmopt ルーチンは IEEE 形式の例外のケースの処理のみをサポートするためです (付録 E を参照してください)。

cc を使用して libmopt と libcopt の両方とリンクするには、コマンド行で -lmopt と -lcopt オプションを指定します (-lm の直前に -lmopt を置き、-lcopt を最後に置くとともによい結果となる)。ほかのコンパイラを使用してこの 2 つのライブラリとリンクする場合は、コマンド行の任意の位置に -xlibmopt フラグを指定します。

SPARC では、ライブラリ `libcx` に 128 ビット 4 倍精度浮動小数点算術演算サポートルーチンよりも速いバージョンを含みます。これらのルーチンはユーザーが直接呼び出すことはできず、コンパイラによって呼び出されます。`-nocx` オプションが指定されていない場合、C++ コンパイラでは自動的に `libcx` と共にリンクが行われますが、C コンパイラでは行われません。C プログラムで `libcx` を使用するには、`-lcx` とともにリンクを行なってください。

`libcx` の共有バージョン (`libcx.so.1`) も提供されます。共有バージョンを実行時にあらかじめロードするには、環境変数 `LD_PRELOAD` を `libcx.so.1` ファイルのフルパス名に指定してください。パフォーマンスを最大にするには、使用しているシステムのアーキテクチャに該当する `libcx.so.1` を使用します。たとえば UltraSPARC システムでは、ライブラリがデフォルトの位置にインストールされている場合、`LD_PRELOAD` を次のように設定します。

csh:

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

sh:

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

ベクトル数学ライブラリ (SPARC のみ)

SPARC プラットフォームでは、`libmvec` ライブラリの提供するルーチンは、引数の全部のベクトルに対して共通数学関数を評価します。

標準インストールを行なった場合の `libmvec` のデフォルトディレクトリは次のとおりです。

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

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

ここで、`<arch>` は算術指定ライブラリディレクトリです。SPARCでは、これらのディレクトリは `v7`、`v8`、`v8a`、`v8plus`、`v8plusa`、`v8plusb`、`v9`、`v9a` および `v9b` を含んでいます。x86 プラットフォームでは、提供されるディレクトリは `f80387` だけです。

表 3-3 は、libmvec 中の関数を一覧しています。

表 3-3 libmvec の内容

種類	関数名
代数関数	vhypot_、vhypotf_、vc_abs_、vz_abs_、vsqrt_、 vsqrtf_、vrsqrt_、vrsqrtf
指数関数および関連した関数	vexp_、vexpf_、vlog_、vlogf_、vpow_、vpowf_、 vc_exp_、vz_exp_、vc_log_、vz_log_、vc_pow_、 vz_pow_
三角関数	vatan_、vatanf_、vatan2_、vatan2f_、vcos_、 vcosf_、vsin_、vsinf_、vsincos_、vsincosf_

libmvec_mt.a はマルチプロセッサの並列化にもとづいて、ベクトル関数の並列化バージョンを提供します。libmvec_mt.a を使用するには、-xparallel と共にリンクしなければなりません。

詳細は、libmvec(3m) および clibmvec(3m) のマニュアルページを参照してください。

libm9x 数学ライブラリ

libm9x 数学ライブラリには、C99 で規定されている数学および浮動小数点に関する関数の一部が含まれています。Forte Developer Compiler リリースでは、改良された浮動小数点例外処理をサポートするために、<fenv.h> (Floating-Point Environment、浮動小数点環境) 関数および各種の拡張プログラムがこのライブラリに含まれています。

libm9x の標準インストールの場合、デフォルトディレクトリは次のとおりです。

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

libm9x のヘッダーファイルの標準インストールの場合、デフォルトディレクトリは次のとおりです。

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

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

次の表 3-4 は、libm9x に含まれる関数を示しています (精度制御関数 fegetprec と fesetprec は、x86 プラットフォームのみに含まれています)。

表 3-4 libm9x の内容

種類	関数名
C99 規格浮動小数点環境関数	feclearexcept、fegetenv、fegetexceptflag、fegetround、feholdexcept、feraiseexcept、fesetenv、fesetexceptflag、fesetround、fetestexcept、feupdateenv
精度制御 (x86)	fegetprec、fesetprec
例外処理と遡及診断 fex_get_log_depth	fex_get_handling、fex_get_log、fex_get_log_depth、fex_getexcepthandler、fex_log_entry、fex_merge_flags、fex_set_handling、fex_set_log、fex_set_log_depth、fex_setexcepthandler

libm9x は、共有ライブラリとしてのみ提供されています。cc コンパイラは、リンク時に共有ライブラリのインストールディレクトリ内でライブラリを自動的に検索することはありません。そのため、cc を使用して libm9x とリンクする場合は、静的リンカーと実行時リンカーの両方を有効にして、ライブラリを検出する必要があります。libm9x を検出するために静的リンカーを有効にするには、次の 3 つの方法のいずれかを使用します。

- コマンド行で、-lm9x の前に -L/opt/SUNWspro/lib を指定する。
- コマンド行で、フルパス名 /opt/SUNWspro/lib/libm9x.so を指定する。
- 環境変数 LD_LIBRARY_PATH が指定するディレクトリリストに /opt/SUNWspro/lib を追加する。

libm9x を検出するために実行時リンカーを有効にするには、次の 3 つの方法のいずれかを使用します。

- リンク時に -R/opt/SUNWspro/lib を指定する。
- リンク時に環境変数 LD_RUN_PATH が指定するディレクトリリストに /opt/SUNWspro/lib を追加する。
- 実行時に環境変数 LD_LIBRARY_PATH が指定するディレクトリリストに /opt/SUNWspro/lib を追加する。

注 - 環境変数 `LD_LIBRARY_PATH` に `/opt/SUNWspro/lib` を追加すると、**Sun Performance Library** にリンクされているプログラムは、プログラムが動作するシステムにもっとも適したライブラリ以外のライブラリバージョンを使用してしまう。cc とリンクされたプログラム内で `libm9x` と **Sun Performance Library** の両方を使用する場合は、`LD_LIBRARY_PATH` に `/opt/SUNWspro/lib` を追加せず、コマンド行で `-lm9x` の前に `-xlic_lib=sunperf` を指定するだけにしてください。

ほかの **Sun ONE Studio Compiler Collection** のコンパイラはすべて、自動的に共有ライブラリのインストールディレクトリを検索します。これらのコンパイラのどれかを使用して `libm9x` とリンクするには、コマンド行で `-lm9x` と指定してください。`libm9x` は主に C および C++ プログラムでの使用を想定したのですが、**FORTRAN** プログラムでも使用できます。例は、付録 A を参照してください。

単精度、倍精度、4 倍精度

多くの数値関数は単精度、倍精度、および 4 倍精度で使用できます。他言語から異なった精度のバージョンを呼び出す例を、表 3-5 に示します。

表 3-5 単精度、倍精度、および 4 倍精度 libm 関数の呼び出し

言語	単精度	倍精度	4 倍精度
C、C++	<pre>#include <sunmath.h> float x,y,z; x = sinf(y); x = fmodf(y,z); x = max_normalf(); x = r_addran();</pre>	<pre>#include <math.h> double x,y,z; x = sin(y); x = fmod(y,z); #include <sunmath.h> double x, y, z; x = max_normal(); x = d_addran();</pre>	<pre>#include <sunmath.h> long double x,y,z; x = sinl(y); x = fmodl(y,z); x = max_normal1();</pre>
Fortran	<pre>REAL x,y,z x = sin(y) x = r_fmod(y,z) x = r_max_normal() x = r_addran()</pre>	<pre>REAL*8 x,y,z x = sin(y) x = d_fmod(y,z) x = d_max_normal() x = d_addran()</pre>	<pre>REAL*16 x,y,z x = sin(y) x = q_fmod(y,z) x = q_max_normal()</pre>

C では、単精度関数の名前は倍精度名に `f` を付加し、4 倍精度関数の名前は倍精度名に `l` を付加することで作成されます。FORTRAN 呼び出し規約は異なっているので、`libsunmath` は単精度関数、倍精度関数、および 4 倍精度関数用にそれぞれ `r_...`、`d_...`、および `q_...` バージョンを提供しています。FORTRAN 組み込み関数は、3 種類の精度のすべてについて総称で呼び出すことができます。

すべての関数が `q_...` バージョンを持っているわけではありません。libm と libsunmath 関数の名前と定義については、`<math.h>` と `<sunmath.h>` を参照してください。

FORTRAN プログラムの中で `r_...` 関数を `real`、`d_...` 関数を倍精度、`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、結果として型が不整合になることがあります。

注 - x86 バージョンの C では 4倍精度だけがサポートされます。

IEEE サポート関数

この節では、IEEE 推奨関数、有益な値を与える関数、`ieee_flags`、`ieee_retrospective`、および `standard_arithmetic` と `nonstandard_arithmetic` について説明します。関数 `ieee_handler` および `ieee_flags` についての詳細は、第 4 章を参照してください。

`ieee_functions(3m)` と `ieee_sun(3m)`

`ieee_functions(3m)` と `ieee_sun(3m)` によって記述される関数は、IEEE 規格で要求される機能または IEEE 規格の付録で推奨される機能を備えています。これらはビットマスク演算として効率的に実装されています。

表 3-6 `ieee_functions(3m)`

関数	戻り値
<code>math.h</code>	ヘッダーファイル
<code>copysign(x,y)</code>	<code>y</code> の符号ビットを持つ <code>x</code>
<code>fabs(x)</code>	<code>x</code> の絶対値
<code>fmod(x, y)</code>	<code>y</code> に関する <code>x</code> の剰余
<code>ilogb(x)</code>	整数形式である <code>x</code> の基数 2 非バイアス型指数
<code>nextafter(x,y)</code>	<code>y</code> の方向で <code>x</code> の次に表現可能な数
<code>remainder(x,y)</code>	<code>y</code> に関する <code>x</code> の剰余
<code>scalbn(x,n)</code>	$x \times 2^n$

表 3-7 ieee_sun(3m)

関数	戻り値
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
issubnormal(x)	分類関数
iszero(x)	分類関数
signbit(x)	分類関数
nonstandard_arithmetic(void)	トグルハードウェア
standard_arithmetic(void)	トグルハードウェア
ieee_retrospective(*f)	

remainder(x,y) は、IEEE 規格 754-1985 で規定された演算です。

remainder(x,y) と fmod(x,y) の相違は、fmod(x,y) が常に x と一致する符号を持つ結果を返すのに対し、remainder(x,y) が返す結果の符号は x または y の符号のどちらとも一致しないことがあるという点です。両関数とも明確な結果を返し、不明確な例外は発生しません。

表 3-8 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 3-9 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 – ユーザーは d_<関数> および q_<関数> を使用する FORTRAN プログラムの中で d_<関数> を倍精度として宣言し、q_<関数> を REAL*16 として宣言する必要があります。

ieee_values(3m)

ieee_values(3m) のマニュアルページに記述されている特殊関数によって、無限大、NaN、および最大/最小浮動小数点数のような IEEE 値が得られます。

ieee_values(3m) 関数によって記述される 10 進値と 16 進の IEEE 表現を、表 3-10、表 3-11、表 3-12、表 3-13 に示します。

表 3-10 IEEE 値: 単精度

IEEE 値	10 進値および IEEE 表現	C、C++FORTRAN
最大正規数	3.40282347e+38 7f7fffff	r = max_normalf(); r = r_max_normal()
最小正規数	1.17549435e-38 00800000	r = min_normalf(); r = r_min_normal()
最大非正規数	1.17549421e-38 007fffff	r = max_subnormalf(); r = r_max_subnormal()
最小非正規数	1.40129846e-45 00000001	r = min_subnormalf(); r = r_min_subnormal()
∞	無限大 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 3-11 IEEE 値: 倍精度

IEEE 値	10 進値および IEEE 表現	C、C++FORTRAN
最大正規数	1.7976931348623157e+308 7fefffff ffffffff	d = max_normal();d = d_max_normal()
最小正規数	2.2250738585072014e-308 00100000 00000000	d = min_normal();d = d_min_normal()
最大非正規数	2.2250738585072009e-308 000fffff ffffffff	d = max_subnormal();d = d_max_subnormal()
最小非正規数	4.9406564584124654e-324 00000000 00000001	d = min_subnormal();d = d_min_subnormal()
∞	無限大 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 7ff00000 00000001	d =signaling_nan(0);d = d_signaling_nan(0)

表 3-12 IEEE 値: 4 倍精度 (SPARC)

IEEE 値	10 進値および IEEE 表現	C、C++FORTRAN
最大正規数	1.1897314953572317650857593266280070e+4932 7ffeffff ffffffff ffffffff ffffffff	q = max_normal1();q = q_max_normal()
最小正規数	3.3621031431120935062626778173217526e-4932 00010000 00000000 00000000 00000000	q = min_normal1();q = q_min_normal()
最大非正規数	3.3621031431120935062626778173217520e-4932 0000ffff ffffffff ffffffff ffffffff	q = max_subnormal1();q = q_max_subnormal()
最小非正規数	6.4751751194380251109244389582276466e-4966 00000000 00000000 00000000 00000001	q = min_subnormal1();q = q_min_subnormal()

表 3-12 IEEE 値: 4 倍精度 (SPARC) (続き)

IEEE 値	10 進値および IEEE 表現	C、C++FORTRAN
∞	無限大 7fff0000 00000000 00000000 00000000	<code>q = infinityl(); q = q_infinity();</code>
シグナルを発生しない NaN	NaN 7fff0000 00000000 00000000 00000001	<code>q = quiet_nanl(0); q = q_quiet_nan(0);</code>
シグナルを発生する NaN	NaN 7fff8000 00000000 00000000 00000001	<code>q = signaling_nanl(0); q = q_signaling_nan(0)</code>

表 3-13 IEEE 値: 拡張倍精度 (x86)

IEEE 値	10 進値および IEEE 表現(80 ビット)	C、C++
最大正規数	1.18973149535723176509e+4932 7ffe ffffffff ffffffff	<code>x = max_normal1();</code>
最小正規数	3.36210314311209350626e-4932 0001 80000000 00000000	<code>x = min_normal1();</code>
最大非正規数	3.36210314311209350626e-4932 0000 7fffffff ffffffff	<code>x = max_subnormal1();</code>
最小非正規数	1.82259976594123730126e-4951 0000 00000000 00000001	<code>x = min_subnormal1();</code>
∞	無限大 7fff 00000000 00000000	<code>x = infinityl();</code>
シグナルを発生しない NaN	NaN 7fff c0000000 00000000	<code>x = q;</code>
シグナルを発生する NaN	NaN 7fff 80000000 00000001	<code>x = signaling_nanl(0);</code>

ieee_flags(3m)

`ieee_flags(3m)` は、以下の機能に対する SUN インタフェースを提供しています。

- 丸めの方向を照会または設定します。
- 丸めの精度を照会または設定します。

- 累積例外フラグのチェック、設定、クリアを行います。

ieee_flags(3m) 呼び出しの構文は次の通りです。

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

パラメータに対して設定できる値の ASCII 文字列を表 3-14 に示します。

表 3-14 ieee_flags のパラメータ値

パラメータ	C または C++ の型	取り得る値
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

パラメータについての詳細は、ieee_flags(3m) のマニュアルページを参照してください。

以降に、ieee_flags を使用すると変更可能な算術演算機能を簡単に説明します。ieee_flags および IEEE 例外フラグについての詳細は、第 4 章を参照してください。

mode が *direction* の時、指定された動作は現在の丸め方向に適用されます。丸め方向は、表現可能なもっとも近い数に丸める、ゼロ方向に丸める、+・方向に丸める、-・方向に丸めるのいずれかに設定できます。IEEE のデフォルトの丸め方向は、表現可能なもっとも近い数に丸めるよう設定されています。算術演算結果が隣接する2つの表現可能な数の間にあるとき、算術結果に近い値が結果となります。算術演算結果が表現可能なもっとも近い2つの数のちょうど中間にある場合には、最下位ビットがゼロである値が結果となります。このことを強調するように、表現可能なもっとも近い数への丸めは、もっとも近い偶数値への丸めとも呼ばれます。

ゼロ方向に丸める方法は、IEEE 規格以前に多くのコンピュータが採用していた方法です。これは、数学的には演算結果の切り捨てを意味します。たとえば、 $2/3$ を 6 桁の 10 進数に丸める場合、表現可能なもっとも近い数に丸めると .666667 になりますが、ゼロ方向に丸めると .666666 になります。

丸め方向の確認、クリアーまたは設定に `ieee_flags` を使用するとき、4 つの入力パラメータが取り得る値を、表 3-15 に示します。

表 3-15 `ieee_flags` による丸め方向の入力値

パラメータ	取り得る値
<code>action</code>	<code>get</code> 、 <code>set</code> 、 <code>clear</code> 、 <code>clearall</code>
<code>in</code>	<code>nearest</code> 、 <code>tozero</code> 、 <code>negative</code> 、 <code>positive</code>
<code>out</code>	<code>nearest</code> 、 <code>tozero</code> 、 <code>negative</code> 、 <code>positive</code>

`mode` が `precision` の時、指定した動作は現在の丸め精度に適用されます。x86 プラットフォームでは、丸め精度は、単精度、倍精度、拡張精度に設定できます。デフォルトの丸め精度は拡張精度です。このモードでは、浮動小数点レジスタに渡される算術演算の結果は、完全な 64 ビット精度の拡張倍精度レジスタの形式に丸められます。丸め精度が単精度または倍精度の場合は、浮動小数点レジスタに渡される算術演算の結果は、それぞれ上位の 24 ビットまたは 53 ビットに丸められます。拡張丸め精度を使用してもほとんどのプログラムでは少なくとも正確な結果が得られますが、IEEE 算術演算の論理に厳密に準拠している必要があるプログラムの中には、拡張丸め精度モードで正常に動作しないものがあります。このようなプログラムは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。

SPARC アーキテクチャを使用しているシステムでは、丸め精度は設定できません。丸め精度に関連する `ieee_flags` の呼び出しは、現在の実装では演算結果に影響を与えません。

`mode` が `exception` の時、指定された動作は現在の IEEE 例外フラグに適用されます。`ieee_flags` を使用して IEEE 例外フラグを調べ制御する方法についての詳細は、第 4 章を参照してください。

ieee_retrospective(3m)

`libsunmath` 関数 `ieee_retrospective` は、不要な例外および非標準 IEEE モードに関する情報を `stderr` に出力します。この関数は以下の事項を判断します。

- まだ処理されていない例外
- 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
- 非標準の演算処理が実施されているかどうか

これらの情報はハードウェア浮動小数点状態レジスタに格納されます。

`ieee_retrospective` は、例外フラグ、および対応するトラップが有効になっている例外についての情報を出力します。これら 2 つの異なった情報を混同しないように気を付けてください。例外フラグが発生している場合、それはプログラム実行中のある時点で発生して無視された例外です。例外に対応するトラップが有効であれば、例外は実際には発生していないことがあります。例外が発生している場合は、SIGFPE シグナルが送信されます。`ieee_retrospective` メッセージは、調査する必要がある例外についてユーザーに警告する (例外フラグが発生している場合) か、または例外がシグナルハンドラによって処理されていることをユーザーに知らせます (例外がトラップされた場合)。例外がシグナルハンドラをトラップすると、その例外はもう発生しません。例外、シグナル、トラップ、通知された例外の原因調査方法については、第 4 章を参照してください。

プログラムは、`ieee_retrospective` をいつでも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 C/C++ プログラムおよび `f95` でコンパイルされた Fortran プログラムは、自動的に `ieee_retrospective` を呼び出しません。

`f95` コンパイラはデフォルトで共通例外のトラップを可能にします。そのため、プログラムが明示的にトラップを無効にするか、SIGFPE ハンドラをインストールしない限り、このような例外が起こった場合は即座に異常終了します。`-f77` 互換モードでは、コンパイラはトラップを可能にしません。そのため、浮動小数点例外が発生すると、プログラムは実行し続け、`ieee_retrospective` 出力を通してこの例外を報告し、終了します。

この関数を呼び出すための構文を以下に示します。

```
C および C++      ieee_retrospective_(fp);
FORTRAN           call ieee_retrospective()
```

C の関数の場合、引数 `fp` には出力ファイルを指定します。FORTRAN 関数の場合、常に標準エラー出力に出力されます。

この例は、6 種類ある `ieee_retrospective` 警告メッセージの中の 4 種類を示しています。

```
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)
```

【日本語訳】

注：以下の IEEE 浮動小数点例外が発生しました：

不正確、アンダーフロー

ゼロ方向への丸め

以下の IEEE 浮動小数点例外のトラップが有効です：

オーバーフロー

詳細は、『数値計算ガイド』の `ieee_flags(3M)`,
`ieee_handler(3M)`, `ieee_sun(3m)` に関する説明を参照してください。

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

FORTRAN プログラムからの `ieee_retrospective` メッセージを抑制するには 3 つの方法があります。1 つめの方法では、処理されていない例外をすべてクリアーし、トラップを無効にし、プログラムが終了する前の、近似値に丸める、拡張精度、標準モードを復元します。これを行うには、次のように `ieee_flags`、`ieee_handler` および `standard_arithmetic` を呼び出します。

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

注 – 原因を調査せずに、処理されていない例外をクリアーすることはお勧めしません。

ieee_retrospective メッセージが表示されないようにするもう 1 つの方法は、標準エラー出力 (stderr) をファイルにリダイレクトする方法です。プログラムが ieee_retrospective 以外のメッセージを stderr に出力する場合は、この方法を使用しないでください。

3 つめの方法は、次のようにダミーの ieee_retrospective 関数をプログラムに組み込む方法です。

```
subroutine ieee_retrospective
return
end
```

nonstandard_arithmetic(3m)

第 2 章で説明したように、IEEE 演算では、段階的アンダーフローを使用して、アンダーフロー例外を処理します。一部の SPARC システム上では、演算のソフトウェアエミュレーションを通じて段階的アンダーフローが実装される場合があります。そのような場合に数多くの計算でアンダーフローが発生すると、性能を低下させる原因となることがあります。

特定のアプリケーションで上記のような状況が発生しているかどうかを調べる場合は、ieee_retrospective または ieee_flags を使用して、アンダーフロー例外の発生有無を確認し、プログラムで使用されているシステム時間をチェックします。プログラムが多大な時間をシステム呼び出しに費やし、アンダーフロー例外が発生している場合は、段階的アンダーフローが性能低下の原因と考えられます。このような場合には、IEEE 以外の演算機能を使用すると、プログラムの実行速度が上がります。

nonstandard_arithmetic 関数を呼び出した場合、ゼロへのフラッシュが高速に行われるようなハードウェアモードの SPARC 実装上では、アンダーフローした結果がゼロにフラッシュされます。ただし、段階的アンダーフローの利点が失われるため、高速になる代わりに正確性が低下します。

standard_arithmetic 関数を呼び出すと、ハードウェアがリセットされ、デフォルトの IEEE 演算機能が復元されます。これら 2 つの関数は、IEEE 754 形式のデフォルト演算機能しか利用できない実装 (SuperSPARC® など) では効果がありません。

C99 浮動小数点環境関数

この節では、C99 で規定されている `<fenv.h>` 浮動小数点環境関数について説明します。このコンパイラコレクションのリリースでは、`libm9x.so` ライブラリにこれらの関数があります。これらの関数には `ieee_flags` 関数と同じ機能が多数ありますが、より自然な C インタフェースが使用されており、C99 で定義されているため、将来は移植性がさらに向上する可能性があります。

注 – 一貫した動作を保つため、`libm9x.so` に含まれる C99 浮動小数点環境関数と例外処理の拡張機能、および `libsunmath` 内の `ieee_flags` と `ieee_handler` 関数の両方を同じプログラム内で使用することは避けてください。

例外フラグ関数

`fenv.h` ファイルは、5 つの IEEE 浮動小数点例外フラグ (`FE_INEXACT`、`FE_UNDERFLOW`、`FE_OVERFLOW`、`FE_DIVBYZERO`、および `FE_INVALID`) のそれぞれに対してマクロを定義しています。また、5 つのフラグマクロすべてのビット単位の論理和となるように、マクロ `FE_ALL_EXCEPT` を定義します。以下の説明では、*excepts* パラメータは 5 つのフラグマクロのいずれかのビット単位の論理和であるか、値 `FE_ALL_EXCEPT` です。`fegetexceptflag` 関数と `fesetexceptflag` 関数の場合は、*flagp* パラメータが型 `fexcept_t` (この型は `fenv.h` で定義されている) のオブジェクトを指すポインタでなければなりません。

C99 は、次に示す例外フラグ関数を定義しています。

表 3-16 C99 規格例外フラグ関数

関数	処理
<code>feclearexcept (excepts)</code>	指定されたフラグをクリアする。
<code>fetestexcept (excepts)</code>	指定されたフラグの設定を返す。
<code>feraiseexcept (excepts)</code>	指定された例外を発生させる。
<code>fegetexceptflag (flagp, excepts)</code>	指定された例外を <i>*flagp</i> に保存する。
<code>fesetexceptflag (flagp, excpts)</code>	指定された例外を <i>*flagp</i> 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* 引数が指定する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論理和を返します。たとえば、現在設定されているフラグだけが不正な場合、アンダーフローが発生し、ゼロ除算が行われ、次の記述が `i` を `FE_DIVBYZERO` にセットします。

```
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であれば、トラップを発生させます (例外トラップの詳細は、第 4 章を参照)。有効なトラップがない場合は、対応するフラグをセットするだけです。

`fegetexceptflag` 関数と `fesetexceptflag` 関数は、特定のフラグの状態を一時的に保存しておき、あとでその状態を復元するのに便利です。`fesetexceptflag` 関数は、トラップを発生させません。この関数は、指定されたフラグの値を復元するだけです。

丸め制御

`fenv.h` ファイルは、IEEE の 4 つの丸め方向モードである `FE_TONEAREST`、`FE_UPWARD` (正の無限大の方向)、`FE_TOWARDZERO` (負の無限大の方向)、および `FE_TOZERO` のそれぞれについてマクロを定義しています。**C99** は、丸め方向モードを制御する 2 つの関数を定義しています。`fesetround` は、現在の丸め方向をその引数 (これは上記の 4 つのマクロの 1 つでなければならない) で指定された方向に設定します。`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 プラットフォームでは、`fenv.h` ファイルは 3 つの丸め精度モード、`FE_FLTPREC` (単精度)、`FE_DBLPREC` (倍精度)、および `FE_LDBLPREC` (拡張倍精度) のそれぞれについてマクロを定義しています。これらは **C99** には含まれませんが、x86 上の `libm9x.so` は、丸め精度モードを制御する 2 つの関数を提供します。`fesetprec` は、現在の丸め精度をその引数 (これは上記の 3 つのマクロの 1 つでなければならない) で指定された精度に設定します。`fegetprec` は、現在の丸め精度に対応するマクロの値を返します。

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、標準外モード (標準外モードは SPARC の場合のみ) など、浮動小数点環境全体を表現するデータ型 `fenv_t` を定義しています。以下の説明では、`envp` パラメータは型 `fenv_t` のオブジェクトのポインタでなければなりません。

C99 は、浮動小数点環境を操作する 4 つの関数を定義しています。`libm9x.so` は、マルチスレッド対応のプログラムに便利なその他の関数を提供しています。これらの関数の概要を次の表に示します。

表 3-17 `libm9x.so` 浮動小数点環境関数

関数	処理
<code>fegetenv(envp)</code>	<code>*envp</code> に環境を保存する。
<code>fesetenv(envp)</code>	<code>*envp</code> から環境を復元する。
<code>feholdexcept(envp)</code>	<code>*envp</code> に環境を保存し、連続モードを確立する。
<code>feupdateenv(envp)</code>	<code>*envp</code> から環境を復元し、例外を発生させる。
<code>fex_merge_flags(envp)</code>	<code>*envp</code> から例外フラグの論理和をとる。

`fegetenv` 関数は、浮動小数点環境を保存します。`fesetenv` 関数は、浮動小数点環境を復元します。`fesetenv` に対する引数は、`fegetenv` または `feholdexcept` の呼び出しによって先に保存されている環境のポインタか、`fenv.h` で定義されている定数 `FE_DFL_ENV` のどちらかです。`FE_DFL_ENV` は、すべての例外フラグのクリアー、最近似への丸め (x86 では拡張倍精度への丸めも含む)、連続した例外処理モード (トラップが無効になる)、および無効にされた標準外モード (SPARC の場合) によるデフォルトの環境を表現します。

`feholdexcept` 関数は、現在の環境を保存したあとすべての例外をクリアーし、すべての例外に対して連続した例外処理モードを確立します。`feupdateenv` 関数は、保存されている環境 (`fegetenv` または `feholdexcept` の呼び出しによって保存されているもの、または定数 `FE_DFL_ENV`) を復元し、そのあとで以前の環境でフラグがセットされている例外を発生させます。それらの例外のいずれかに対して有効になったトラップが、復元される環境に含まれる場合は、トラップが発生します。それ以外の場合は、フラグがセットされます。次のコード例に示すように、これらの 2 つの関数はあわせて使用し、例外を発生させそうなサブルーチンを呼び出すことができます。

```

#include <fenv.h>

void myfunc(...) {fenv_t env;

    /* 環境を保存し、フラグをクリアし、トラップを無効にします */
    feholdexcept(&env); /* 例外を発生させる可能性がある計算を行います */
    ...
    /* 疑似例外を調べます */ if (fetestexcept(...)) { /* 例外を適切に処理
    し、それらのフラグをクリアします */ ...feclearexcept(...); } /* 環境を復
    元し、関連する例外を発生させます */ feupdateenv(&env); {

```

`fex_merge_flags` 関数は、トラップを発生させることなく、保存済みの環境の例外フラグの論理和をとり現在の環境に入れるだけです。この関数をマルチスレッド対応のプログラムで使用する、子スレッドでの計算で発生したフラグに関する情報を親スレッドに保持できます。`fex_merge_flags` の使用方法を示した例は、付録 A を参照してください。

libm と libsunmath の実装上の特徴

この節では、使用可能な `libm` と `libsunmath` の実装上の特徴について説明します。

- 限りなく精密な p を使用する引数還元、および p で基準化 (スケール) した三角関数
- IEEE 形式と IEEE 以外の形式間の浮動小数点のデータを変換するためのデータ変換ルーチン
- 乱数発生機構

アルゴリズム

SPARC システム上の `libm` および `libsunmath` 中の基本関数は、テーブル駆動型および多項式有理数の近似値のアルゴリズムによって実装されています。`x86` プラットフォーム上の `libm` および `libsunmath` 中の基本関数には、`x86` 命令セット中に提供

されている基本関数カーネル命令を使用して実装されているものと、SPARC システムで使用されているのと同じテーブル駆動型および多項式有理数の近似値アルゴリズムによって実装されているものがあります。

libm 中の共通の基本演算および libsunmath 中の単精度基本演算に対する、テーブル駆動型および多項式有理数の近似値のアルゴリズムは、最後の place 内の 1 単位 (*ulp* = unit-in-the-last-place) 中だけで正確な結果を算出します。SPARC システムでは、libsunmath 中の共通の 4 倍精度基本演算は、1 *ulp* 中で正確な結果を算出します (*expm1l* および *log1pl* 演算を除く)。*expm1l* および *log1pl* 演算は、2 *ulp* 中で正確な結果を算出します (共通の関数には、指数関数、対数関数、累乗関数、ラジアン引数の循環三角関数などがあります。双曲線三角関数や高度な超越関数のようなその他の関数は、正確度が低くなります)。これらのエラー境界は、アルゴリズムを直接分析して調べることができます。BeEF (Berkley Elementary Function) 検査プログラムを使用して、これらのルーチンの正確度を検査することもできます。BeEF は、Z.Alex Liu によって開発され、ucbtest パッケージの netlib から入手できます (<http://www.netlib.org/fp/ucbtest.tgz>)。

三角関数の引数還元

範囲 $[-\pi/4, \pi/4]$ の外側のラジアン引数に対する三角関数は、通常 $\pi/2$ の整数倍を引いて、引数を指定された範囲内に還元して計算します。

π はマシンで表現可能な数ではないので、何らかの方法で近似する必要があります。最終的に計算された三角関数の誤差は、引数還元の丸め誤差 (近似 π と丸めによる) と、還元された引数の三角関数計算の誤差で決まります。相当に大きい引数の場合でも、引数還元起因する誤差は他の誤差より大きくありませんが、これに反して相当に小さい引数の場合は、最終結果の相対誤差は引数還元誤差によって左右されます。

マシンで表現可能な大きい数は、 π より大きい範囲で分割されているという理由から、大きな引数の三角関数は本来不正確で、小さい引数はすべて比較的正确であるという誤解が広まっています。

しかし、計算した三角関数の値が急に大きくなるという固有の境界はなく、不正確な関数値が役に立たないということもありません。引数還元が一樣に行われるならば、基本的同一性と関連性が大きい引数に対しても小さい引数と同様に保護されるので、引数還元が π への近似で行われたという事実はほとんど検知できません。

libm と libsunmath 三角関数は、引数還元のため「無限に」正確な π を使用します。値 $2/\pi$ は 16 進数 916 桁まで計算され、参照用テーブルに格納されて引数還元中に使用されます。

関数 `sinpi`、`cospi`、および `tanpi` のグループ (31 ページの表 3-2 を参照) は、引数の一定範囲への縮小によって不正確になることを避けるため π で入力引数を基準化 (スケール) します。

データ変換ルーチン

`libm` と `libsunmath` には、IEEE 形式とそれ以外の形式間で 2 進浮動小数点データの変換に使用されるデータ変換ルーチン `convert_external` があります。

サポートする形式には、SPARC (IEEE)、IBM PC、VAX、IBM S/370、および Cray が使用する形式が含まれます。

Cray で生成されたデータを関数 `convert_external` を使用して、SPARC システムが希望する IEEE 形式に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。

乱数発生機構

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の各形式で、一様疑似乱数を生成する 3 つの方法を次に示します。

- `addrans(3m)` のマニュアルページに述べられた関数は、テーブル駆動型の加法的乱数ジェネレータのグループに基づいています。
- `lcrans(3m)` のマニュアルページに述べられた関数は、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に述べられた関数は、桁上げ乗算による乱数ジェネレータに基づいています。これらの関数には、64 ビット整数形式の一様疑似乱数を提供するジェネレータも含まれています。

また、`shufrans(3m)` のマニュアルページに述べられた関数をこれらの任意のジェネレータとあわせて使用すると、疑似乱数の配列を動かし、アプリケーションにより大きなランダム性を与えることができます (64 ビット整数の配列を動かす方法はありません)。

各乱数機能には、一度に 1 つ (関数呼び出しごとに 1 つ) の乱数を生成するルーチンと、一度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。一度に 1 つの乱数を生成する関数は、次の表 3-18 に示す範囲の数値を配布します。

表 3-18 単一値乱数の値の発生範囲

関数	下限	上限
i_addran_	-2147483648	2147483647
r_addran_	0	0.9999999403953552246
d_addran_	0	0.9999999999999998890
i_lcran_	1	21477483646
r_lcran_	4.656612873077392578E-10	1
d_lcran_	4.656612875245796923E-10	0.9999999995343387127
i_mwcran_	0	2147483647
u_mwcran_	0	4294967295
i_llmwcan_	0	9223372036854775807
u_llmwcan_	0	18446744073709551615
r_mwcan_	0	0.9999999403953552246
d_mwcran_	0	0.9999999999999998890

一度の呼び出しで乱数の配列全体を生成する関数を使用すると、生成される数値の範囲を指定できます。付録 A では、異なる間隔で一様に分布される乱数の配列を生成する方法を示したいくつかの例が挙げられています。

addrans および mwcrans ジェネレータは、lcrans ジェネレータよりも一般に効率的ですが、これらのジェネレータの理論は精緻ではありません。線形合同アルゴリズムの理論的な特性については、“Communications of the ACM” の 1988 年 10 月号に掲載された S. Park と K. Miller による “Random Number Generators: Good Ones Are Hard To Find” で説明されています。また、加法的乱数ジェネレータについては、Knuth の『The Art of Computer Programming』の第 2 版で説明されています。

第4章

例外と例外処理

この章では、IEEE の浮動小数点例外と、浮動小数点例外の検出、特定、処理について説明します。

SPARC[®]、x86 プラットフォーム上の Sun ONE Studio Compiler Collection のコンパイラおよび Solaris オペレーティングシステムで提供される浮動小数点環境では、IEEE 標準規格で定められているすべての例外処理機能、およびその他の推奨されている機能がサポートされています。IEEE 標準の目的の 1 つは、「IEEE 854 標準規格」の次の部分で述べられています (IEEE 854 標準規格の 18 ページを参照してください)。

... 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います。すなわち、異常な事態が発生しても、適切なフラグを設定するなど、合理的なデフォルトの応答によって対処する必要があります。

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います。ユーザーが検出、設定、クリアすることによって例外が発生したことを示すステータスフラグを実装する必要があるとしています。また、例外が発生した時に、ブロックをトラップする (通常の制御フローを中断する) 方法を実装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、プログラムで適切な方法で例外を処理するトラップハンドラを用意することもできます。この章では、IEEE 754 で定義されている例外、そのデフォルトの結果、ステータスフラグ、トラップ、例外処理をサポートする浮動小数点環境について説明します。

例外とは

例外の定義は困難ですが、W.Kahan によると以下のようになっています (W.Kahan 著『Handling Arithmetic Exceptions』を参照してください)。

算術演算例外は、不可分 (atomic) な算術演算の結果が、一般に受け入れ可能なものでないときに発生します。「不可分な」および「受け入れ可能な (acceptable)」の意味は場合によって異なります。

負数の平方根を取ろうとするのは例外であり、通常は無効な演算によって引き起こされた例外と見られます。この場合、システムでは以下のうちのいずれかのことがらが起こります。

- 例外トラップが無効である場合 (デフォルト)、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処理に関するデフォルト結果を使用して、プログラムの実行を続行します。
- 例外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設定されていると、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、プログラムが異常終了します。

IEEE の浮動小数点例外は、無効な演算、ゼロ除算、オーバーフロー、アンダーフロー、および不正確の 5 種類があります。最初の 3 つ (無効な演算、ゼロ除算、オーバーフロー) の例外は共通の例外と呼ばれており、無視することはできません。ieee_handler(3m) には共通の例外だけをトラップする簡単な方法が提供されています。他の 2 つの例外 (アンダーフロー、不正確) はより頻繁に発生します。実際、ほとんどの浮動小数点演算に不正確例外を発生しており、ほとんどの場合は無視できます。

表 4-1 は IEEE 標準規格 754 の内容を要約したものです。5 種類の浮動小数点例外および例外発生時の IEEE 演算機能環境のデフォルトの応答を定義しています。

表 4-1 IEEE 浮動小数点例外

IEEE 例外	例外の発生理由	例	トラップ未設定時のデフォルト結果
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	$0 \times \infty$ $0/0$ $\infty \times \text{REM}$ 0	シグナルを発生しない NaN

表 4-1 IEEE 浮動小数点例外 (続き)

IEEE 例外	例外の発生理由	例	トラップ未設定時の デフォルト結果
	(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する場合にもこの例外が発生する。しかし、このことは IEEE 規格には含まれない)	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)	
ゼロによる除算	有限オペランドに対する演算により結果が純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、移動先の形式で表現可能な最大数を超えている (指数部分を越えた)	倍精度: $\text{DBL_MAX} + 1.0\text{e}294\text{exp}(709.8)$ 単精度: $(\text{float})\text{DBL_MAXFLT_MAX} + 1.0\text{e}32\text{expf}(88.8)$	丸めモード (RM) と中間結果の符号に依存する RM + - RN $+\infty$ $-\infty$ RZ +max -max R- +max $-\infty$ R+ $+\infty$ -max
アンダーフロー	正確な結果、正しく丸められた結果のどちらも、宛先の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 : $\text{nextafter}(\text{min_normal}, -\infty)\text{nextafter}(\text{min_subnormal}, -\infty)\text{DBL_MIN}/3.0 \exp(-708.5)$ 単精度 : $(\text{float})\text{DBL_MINnextafterf}(\text{FLT_MIN}, -\infty)\text{expf}(-87.4)$	非正規数または 0
不正確	丸められた有効な演算結果が、真の演算結果と異なる (ほとんどの浮動小数点演算ではこの例外が発生する)	$2.0/3.0(\text{float})1.123456781\log(1.1)\text{DBL_MAX} + \text{DBL_MAX}$, オーバーフローがトラップされないとき	演算結果 (丸め、オーバーフロー、またはアンダーフローなど)

表 4-1 の注

1. 非順序付け比較: 任意の浮動小数値の組は、形式が異なっても比較することができます。次の 4 つの相互に排他的な比較演算 (より小さい、より大きい、等しい、および非順序付け) が可能です。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します。次の表は非順序付け関係のとき、どの演算子が無効な演算例外を発生するかを示したものです。

表 4-2 非順序付け比較

演算子			
数学	C, C++	F95	無効な例外 (非順序付けの場合)
=	==	.EQ.	無効
≠	!=	.NE.	無効
>	>	.GT.	無効ではない
≥	>=	.GE.	無効ではない
<	<	.LT.	無効ではない
≤	<=	.LE.	無効ではない

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。また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} 、 2^{-16382} です。IEEE の浮動小数点形式については、第 2 章を参照してください。

x86 浮動小数点環境には、IEEE 規格にはない例外 (指数が最小の非正規数オペランド例外) 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。

例外の優先順位は次のとおりです。

表 4-3 例外の優先順位

x86	SPARC および PowerPC
無効	無効
オーバーフロー	オーバーフロー
除算	除算
アンダーフロー	アンダーフロー
不正確	不正確
非正規数	

同時に発生する可能性のある標準的な例外は、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だけです。x86 では、5 つの標準的な例外のどれとでも同時に非正規数例外が発生します。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可能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トラップよりも優先されます。これらはすべて x86 の非正規数よりも優先されます。

例外の検出

IEEE 規格で要求されているように、SPARC、x86 プラットフォー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状態フラグが提供されています。どの例外が発生したかを検出するために、プログラムでこれらのフラグをテストできます。これらのフラグは、明示的に設定またはクリアすることもできます。ieee_flags 関数はこれらの例外にアクセスする方法の 1 つです。C および C++ で書き込まれたプログラムでは、libm9x.so に含まれる C99 浮動小数点環境関数により別の状態フラグが提供されています。

SPARC では、各例外に現状を示す「現在フラグ」と「累積フラグ」の 2 つが用意されています。現在の例外フラグは、最後に実行された浮動小数点の演算結果によってその都度更新されます。これらの現在の例外フラグは累積例外フラグにも累積されません。これによって、プログラムの実行開始後またはプログラムにより累積フラグが最後にクリアされた時点以降に発生し、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。浮動小数点演算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セットされますが、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、浮動小数点状態レジスタ %fsr 内に保存されます。

x86 では、累積フラグは、最初の例外発生時に設定されます。明示的にクリアしない限り、ユーザープロセスの終了までその値を保持します。

ieee_flags(3m)

ieee_flags(3m) 呼び出しの構文は次の通りです。

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

2 つ目の引数に `exception` という文字列を指定すると、プログラムは `ieee_flags(3m)` 関数を使用して、発生した例外のステータスフラグを、検査、設定、クリアします。

たとえば、**FORTTRAN** 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++ では、例外が発生したかどうかの問い合わせは以下のようにします。

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

処理が特定できた場合に出力パラメータ `out` に返される文字列は、以下の通りです。

- `not available` — 例外に関する情報が取得できません。
- `" "` (空白の文字列) — 累積例外が一度も発生していません。または `x86` の場合、非正規オペランドが唯一の累積例外です。
- 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、もっとも高い優先順位を持つ例外の名前が返されます。`in` が他の文字列に設定されている場合には、もっとも高い優先順位を持つ累積例外が返されます。

FORTTRAN 呼び出しにおける例を示します。

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

ゼロによる除算の例外が発生している場合には、`out` は `"division"` に設定されます。設定されていない場合には、もっとも優先順位の高い例外が `out` に返されます。`in` に特定の例外が指定されてない場合には、それは無視されます。

たとえば、次の呼び出しでは "all" には特別な意味はありません。

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

さらに `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、すべての例外フラグのビット単位の論理和で、それぞれのフラグは表 4-4 に示すようにビットで表わします。`sys/ieee.h` ファイルは、それぞれの例外に対応するビット位置を定義します。このビット位置はマシンによって異なり、連続 (隣接) している必要はありません。

表 4-4 例外ビット

例外	ビット位置	例外ビット
無効	<code>fp_invalid</code>	<code>i & (1 << fp_invalid)</code>
オーバーフロー	<code>fp_overflow</code>	<code>i & (1 << fp_overflow)</code>
ロー		
除算	<code>fp_division</code>	<code>i & (1 << fp_division)</code>
アンダーフロー	<code>fp_underflow</code>	<code>i & (1 << fp_underflow)</code>
ロー		
不正確	<code>fp_inexact</code>	<code>i & (1 << fp_inexact)</code>
非正規数	<code>fp_denormalized</code>	<code>i & (1 << fp_denormalized)</code> (x86 のみ)

下記の C または C++ のプログラムの一部は、i の値をデコードする方法を示しています。

```
/*
 * すべての累積例外を示す整数値をデコードします。
 * fp_inexact などは、<sys/ieee.h> 内に定義されています。
 */

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

C99 例外フラグ関数

C および C++ プログラムでは、libm9x.so に含まれる 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のテスト、セット、およびクリアができます。ヘッダーファイル fenv.h は、5 つの標準的な例外、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論理和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あ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。一貫した動作を保つため、libm9x.so に含まれる C99 浮動小数点環境関数と拡張機能、および libsunmath に含まれる ieee_flags と ieee_handler 関数の両方を同じプログラム内で使用することは避けてください。

5 つの例外フラグすべてをクリアするには、次のように記述します。

```
feclearexcept(FE_ALL_EXCEPT);
```

無限演算またはゼロ除算が発生したかどうかをテストするには、次のように記述します。

```
int i;

i = fetestexcept(FE_INVALID | FE_DIVBYZERO); if (i & FE_INVALID) /*
無効な例外が発生しました */ else if (i & FE_DIVBYZERO) /* ゼロ除算例外が
発生しました */
```

オーバーフロー例外の発生をシミュレートするには、次のように記述します (このコードは、オーバーフロートラップが有効な場合にはトラップを起こすことに注意)。

```
feraiseexcept(FE_OVERFLOW);
```

`fegetexceptflag` および `fesetexceptflag` 関数は、フラグのサブセットの保存と復元に使用します。次に、この 2 つの関数を使用した例を示します。

```
fexcept_t flags;

/* アンダーフロー、オーバーフロー、および不正確のフラグを保存します
*/ fegetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW |
FE_INEXACT); /* これらのフラグをクリアします
*/ feclearexcept(FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT); /* アン
ダーフローまたはオーバーフローの可能性のある演算を行います */ ... /* アンダー
フローまたはオーバーフローを調べます */ if (fetestexcept(FE_UNDERFLOW |
FE_OVERFLOW) != 0) {...} /* アンダーフロー、オーバーフロー、および不正確
のフラグを復元します */ fesetexceptflag(&flags, FE_UNDERFLOW |
FE_OVERFLOW, | FE_INEXACT);
```

例外の特定

プログラマが例外を考慮してプログラムを作成していないために、例外が検出されたときに、どこで例外が発生したのかが問題になることがよくあります。例外が発生した場所を特定する方法の 1 つは、プログラム中のさまざまな箇所で例外フラグをテストすることですが、この方法で正確に例外を特定するには、多くのテストと労力が必要になります。

例外の発生した場所を特定する簡単な方法は、例外トラップを有効にすることです。トラップが有効で、ある例外が発生すると、オペレーティングシステムは、SIGFPE シグナルを送ってプログラムに通知します (詳細は、signal(5) のマニュアルページを参照してください)。例外のトラップを有効にすると、デバグガで実行して SIGFPE シグナルを受信した時点でプログラム終了するか、または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、例外の発生箇所を特定することができます。SIGFPE シグナルを生成する例外については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 58 ページの表 4-1 に示されているデフォルトの結果で継続されてますが、シグナルは送られません。

デバグガを使用して例外を特定する

この節では、dbx (ソースレベルのデバグガ) と adb (アセンブリレベルのデバグガ) の使用例を参考にして、浮動小数点例外の原因と、例外発生させた命令を調べます。dbx でソースレベルのデバグガを行うには、プログラムを -g オプション付きでコンパイルする必要があります。詳細は、『dbx コマンドによるデバグガ』マニュアルを参照してください。

次の C プログラムを見てみます。

```
#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
```

NaN が出力された場合、無効な演算例外が発生した可能性があります。原因を突き止めるには、無効な演算についてのトラップを有効にするために、`-ftrap` オプションを付けて再コンパイルし、`dbx` または `adb` を使用してプログラムを実行し、SIGFPE シグナルが送信された場所で停止させることができます。もう 1 つの方法として、無効な演算に対するトラップを有効にする起動ルーチンとリンクするか、または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せずに、`dbx` または `adb` を使用することができます。

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を突き止めるには、`-g` オプションおよび `-ftrap` オプションを使って再コンパイルし、`dbx` を使用して例外が発生している場所を追跡します。

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

`-g` オプションでコンパイルすると、`dbx` のソースレベルのデバッグ機能を使用することができます。`-ftrap=invalid` を指定すると、無効な演算に対する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、SIGFPE が出されたときにプログラムを停止するように `catch fpe` コマンドを実行し、プログラムを実行します。SPARC では、結果は次のようになります。

```
example% dbx a.out
a.out の読み込み中
(dbx) catch fpe
(dbx) run
実行中 : a.out
(プロセス id 653)
シグナル FPE ( 無効浮動小数点演算 ) sqrt at 0xff37c060 で
0xff37c060: sqrt+0x0038:      ld      [%g1], %g1
関数 : sqrtm1
4  double sqrtm1(double x) { return sqrt(x) - 1.0; }
(dbx) print x
x = -4.2
(dbx)
```

この出力例では、負の数の平方根を求めようとした結果、`sqrtnl` 関数で例外が発生していることがわかります。

adb を使用して例外の原因となっている命令を特定する

adb を使用して例外の原因を特定することもできます。ただし、`dbx` とは異なり、adb ではソースファイルや行番号は特定できません。adb を使用する場合も、最初の手順は `-ftrap` を指定してプログラムを再コンパイルします。

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

次に adb を起動してプログラムを実行します。無効な演算例外が発生すると、adb は例外の原因である命令の次の命令で停止します。例外の原因である命令を見つけるには、いくつかの命令を逆アセンブルし、adb 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 アーキテクチャのシステムでは、結果は次の例のようになります。

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

この出力例は、`fsqrrd` 命令が原因で例外が発生したことを示しています。ソースレジスタを調べると、負の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 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ため、ここから逆アセンブルできます。(この出力は、プログラムが `-xlibmil` フラグを指定してコンパイルされていることを前提にしています。) 一般的な結果は次のようになります。

```
example% adb a.out
:r
SIGFPE: Arithmetic Exception (invalid floating point operation)
stopped at      sqrtml_+0x13:  faddp  %st,%st(1)
sqrtml_?8i
sqrtml_:
sqrtml_:      pushl  %ebp
               movl  %esp,%ebp
               subl  $0x24,%esp
               fldl  $0x8048b88
               movl  0x8(%ebp),%eax
               fldl  (%eax)
               fsqrt
               faddp  %st,%st(1)

$x
80387 chip is present.
cw      0x137e
sw      0x3000
cssel 0x17  ipoff 0x8acd          dataset 0x1f  dataoff 0x0

st[0]  -4.2000000000000001776356839          VALID
st[1]  -1.0                                VALID
st[2]  +0.0                                EMPTY
st[3]  +0.0                                EMPTY
st[4]  +0.0                                EMPTY
st[5]  +0.0                                EMPTY
st[6]  +0.0                                EMPTY
st[7]  +0.0                                EMPTY
```

この出力例は、`fsqrt` 命令が原因で例外が発生したことを示しています。浮動小数点レジスタを調べると、負の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。

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、主プログラムを再コンパイルすることにより、無効な演算の例外トラップを有効にしました。主プログラムの再コンパイルが不可能な場合があり、他の方法でトラップを有効にしなければならないことがあります。トラップを有効にするにはいくつかの方法があります。

`dbx` の使用中に、浮動小数点状態レジスタを直接変更することによって、トラップを有効にすることができます。`SPARC` では、この方法には注意が必要です。オペレーティングシステムでは、プログラム内で最初に使用されるまで浮動小数点ユニットは使用可能になりません。浮動小数点ユニットが最初に使用された時点で、浮動小数点状態レジスタはリセットされ、すべてのトラップは無効になります。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手動でトラップを有効にすることはできません。ここで示した例では、`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、無効な演算に対する例外トラップを有効にし、`SIGFPE` シグナルの受信時に `dbx` を停止するよう設定して、実行を継続できます。アーキテクチャのシステムでの手順は次のようになります。無効な演算例外に対するトラップを有効にするために、`assign` コマンドを使用して `%fsr` を変更していることに注意してください。

```
example% dbx a.out
a.out の読み込み中
...
(dbx) stop in sqrtm1
dbx: 警告: 'sqrtm1' はデバッグ情報を持っていません -- 最初の命令で停止します
(2) stop in sqrtm1
(dbx) run
Running: a.out
(プロセス id 693)
sqrtm1 で停止しました 0x106b8 で
0x000106b8: sqrtm1      :      save    %sp, -0x70, %sp
(dbx) assign $fsr=0x08000000
dbx: 警告: 言語不明。'ansic' と仮定
(dbx) catch fpe
(dbx) cont
シグナル FPE (無効浮動小数点演算) 関数 sqrt 0xff37c060
0xff37c060: sqrt+0x0038:      ld      [%g1], %g1
(dbx)
```

x86 では、コンパイラが各プログラム内に自動的にリンクする起動コードは、制御をメインプログラムに引き渡す前に浮動小数点ユニットを初期化します。そのため、メインプログラムが開始した後で、いつでも手動でトラップを有効にすることができます。次に、この処理のステップ例を示します。

```
example% dbx a.out
a.out の読み込み中
... etc.
(dbx) stop in main
dbx: 警告: 'main' はデバッグ情報を持っていません -- 最初の命令で停止します
(2) stop in main
(dbx) run
実行中: a.out
(プロセス id 25055)
main で停止しました 0x80508f8
0x080508f8: main      :      pushl   %ebp
(dbx) assign $fctrl=0x137e
dbx: 警告: 言語不明。'ansic' と仮定
(dbx) catch fpe
(dbx) cont
シグナル FPE (無効浮動小数点演算) 関数 setexception 0xdfbccbb0 で
0xdfbccbb0: setexception+0x0404:      fstpl   -44(%ebp)
(dbx)
```

トラップを有効にする初期化ルーチンを作成すると、メイン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、例外が発生する場合に、デバッガ上で実行することなくプログラムを中止したい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、プログラムを適切な初期化ルーチンに再リンクするだけでトラップを有効にすることができます。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 などは ieeefp.h 内で定義されています*/
    fpsetmask(FP_X_INV);
}
```

次に、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
演算例外
```

再リンクが不可能であるがプログラムは動的にリンクされているという場合は、実行時リンカーの、共有オブジェクトをプリロードする機能を使用してトラップを有効にすることができます。SPARC システムでこのように行う場合は、上記と同じ C ソースファイルを作成し、次に示す方法でコンパイルしてください。

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

次にトラップを有効にします。init.so オブジェクトのパス名を、環境変数 LD_PRELOAD によって指定されたプリロードする共有オブジェクトの一覧に追加します。次に例を示します。

```
example% env LD_PRELOAD=./init.so a.out
演算例外
```

(共有オブジェクトの作成とプリロードの詳細は、『リンカーとライブラリ』を参照してください。)

上記で説明しているように、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プリロードすることにより原則として変更できます。しかし、共有オブジェクト内の初期化ルーチンは、プリロードされる場合も明示的にリンクされる場合も、メインの実行プログラム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。続いて起動コードは、-fttrap、-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確立します。そして、メインの実行プログラム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、最後に制御をメインプログラムに渡します。そのため、SPARC では、(i) 上記の例で有効にされているトラップのような、共有オブジェクト内の初期化ルーチンにより確立される浮動小数点制御モードはすべて、無効にされないかぎりプログラムの実行中は有効な状態が継続します。(ii) 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は、共有オブジェクト内の初期化ルーチンによって確立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

して選択されるデフォルトモードは、以前に確立されているモードを無効にしません)。 (iii) メインの実行プログラムの一部である初期化ルーチンまたは、メインプログラム自体によって確立されるモードはすべて、両方とも無効にします。

x86 では状況が複雑です。新しいプロセスが開始されるたびにシステムカーネルが非デフォルトモードをいくつか使用して浮動小数点ハードウェアを初期化しますが、メインプログラムに制御を渡す前に、コンパイラにより自動提供される起動コードがこれらのモードの一部をデフォルトにリセットします。そのため、共有オブジェクト内の初期化ルーチンは、それらが浮動小数点制御モードを変更しないかぎり、無効演算、ゼロ除算、およびオーバーフロー例外に対するトラップを有効にし、丸め精度を 53 個の有効ビットに設定した状態で動作します。実行時リンカーが制御を起動コードに渡した後は、起動コードが標準 C ライブラリ libc 内のルーチン `__fpstart` を呼び出します。これにより、トラップがすべて無効になり、丸め精度が 64 個の有効ビットに設定されます。起動コードは続いて、`-fround`、`-ftrap`、または `-fprecision` フラグにより選択されている非デフォルトモードを確立し、そのあと静的にリンクされた初期化ルーチンを実行し、制御をメインプログラムに渡します。そのため、初期化ルーチンを持つ共有オブジェクトをプリロードすることにより x86 プラットフォーム上でトラップを有効にしたり丸め精度モードを変更したりするには、トラップイネーブルモードと丸め精度モードを `__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。しかし、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を実行します。次に、この処理を行うコード例を示します。

```
#include <ieeefp.h>#include <sunmath.h>#include
<sys/sysi86.h>#pragma init (trapinvalid)void trapinvalid(){/*
FP_X_INV et al は ieeefp.h で定義されます */
fpsetmask(FP_X_INV);}extern int __fltrounds(),
__flt_rounds;extern long _fp_hw;void __fpstart(){char *out;/*
System V ABI Intel プロセッササプリメントによって定義されている 標準の
__fpstart() 関数と同じ浮動小数点初期化を実行しますが、 すべてのトラップ
モードをそのままにします */__flt_rounds = __fltrounds();
sysi86(SI86FPHW, &_fp_hw);_fp_hw &= 0xff;ieee_flags("set",
"precision", "extended", &out);}
```

上記の `fpstart` ルーチンが `libsunmath` に含まれる `ieee` フラグを呼び出すことに注意してください。このように、このコードから共有オブジェクトを作成するには、`-R` および `-L` オプションを使用して共有オブジェクト `libsunmath.so` の場所を指定する必要があります。このライブラリはコンパイラコレクション製品のイントール場

所に置かれます。ここでは、コンパイラコレクションをデフォルト位置にインストールしていることを前提としています。上記のコードをコンパイルして、共有オブジェクトを作成し、この共有オブジェクトをプリロードして、トラップを可能にする手順は次の通りです。

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

シグナルハンドラを使用して例外を特定する

前の節では、例外の最初の発生を特定するためにプログラムの外で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して例外の特定の発生を検出することもできます。トラップを有効にしても、SIGFPE ハンドラを設定していない場合は、トラップされた例外の次の発生時にプログラムが異常終了します。SIGFPE ハンドラを設定している場合は、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ます。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、異常終了するか、実行を再開します。実行を再開して意味のある結果を得るには、次の節で説明する例外処理のための結果をハンドラで設定する必要があります。

`ieee_handler` を使用して、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異常終了するか、SIGFPE ハンドラを設定するように指定することができます。SIGFPE ハンドラは、低レベルの関数 `sigfpe(3)`、`singal(3c)`、`sigaction(2)` のいずれかを使用して設定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を有効にできません。浮動小数点例外は、そのトラップが有効である場合のみ SIGFPE シグナルを引き起こすことができます。

`ieee_handler(3m)`

`ieee_handler` の呼び出し時の構文は、次の通りです。

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

2 つの入力パラメータ *action* (動作) と *exception* (例外) は文字列です。3 つ目のパラメータ *handler* (ハンドラ) は、型が `sigfpe_handler_type` の関数で、`floatingpoint.h` 内に定義されています。

入力パラメータの取り得る値は、以下の通りです。

入力パラメータ	C または C++ の型	取り得る値
action	char *	get、set、clear
exception	char *	invalid、division、 overflow、underflow、 inexact、 all、common
handler	sigfpe_handler_type	ユーザー定義ルーチン SIGFPE_DEFAULT SIGFPE_IGNORE SIGFPE_ABORT

action が *set* の場合、*ieee_handler* は *exception* で指定した例外について *handler* で指定した処理関数を確立します。処理関数は、SIGFPE_DEFAULT や SIGFPE_IGNORE (いずれもデフォルトの IEEE 動作を選択する)、SIGFPE_ABORT (指定した例外が発生するとプログラムを異常終了させる)、ユーザー定義のサブルーチンのアドレス (指定した例外のいずれかが発生するとサブルーチンを起動する) のいずれかです。ユーザー定義のサブルーチンには、SA_SIGINFO フラグを設定して指定したシグナルハンドラと同じパラメータ (*sigaction*(2) のマニュアルページを参照) が渡されます。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ます。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なり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、疑似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、プログラムは指定された例外ごとにフラグをクリアする必要があります。

action が *clear* の場合、*ieee_handler* は指定した例外について現在設定されている処理関数を取り消し、例外のトラップを無効にします。これは、SIGFPE_DEFAULT を設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、要求された動作が成功した場合はゼロが返されます。動作が失敗した場合はゼロ以外の値が返されます。

action が *get* の場合、*ieee_handler* は指定した例外について現在設定されているハンドラ (ハンドラが設定されていない場合は *SIGFPE_DEFAULT*) のアドレスを返します。

以下に、*ieee_handler* の使用方法を示すコード例を示します。この C のコードでは、ゼロによる除算が発生した場合はプログラムを終了します。

```
#include <sunmath.h>
/* x86 システムでは、次の行のコメントを解除します */
/*ieee_flags("clear", "exception", "division", NULL); */
if (ieee_handler("set", "division", SIGFPE_ABORT) != 0)
    printf("ieee trapping not supported here \n");
```

FORTTRAN の場合は次のようになります。

```
#include <floatingpoint.h>
c uncomment the following line on x86 systems
c      ieee_flags('clear', 'exception', 'division', %val(0))
c      i = ieee_handler('set', 'division', SIGFPE_ABORT)
c      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");
```

FORTTRAN 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'
```

シグナルハンドラからの例外の報告

ieee_handler によって設定された 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、発生した例外の型、例外の原因である命令のアドレス、マシンの整数レジスタおよび浮動小数点レジスタの内容を示す情報が通知されます。ハンドラはこの情報を調査して、例外と例外の発生場所を示すメッセージを出力します。

システムから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 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、送られたシグナルの番号がパラメータ *sig* に格納されます。シグナル番号は sys/siginfo.h 内で定義されています。SIGFPE のシグナル番号は 8 です。

パラメータ *sip* は、シグナルに関する追加情報を記録する構造体を指します。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 (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、システムとどのようなイベントで SIGFPE シグナルが発生するかによって異なります。

sip->si_code メンバーは、表 4-5 に示す SIGFPE シグナルの型のいずれかです。これらは sys/machsig.h 内で定義されています。

表 4-5 算術例外の型

SIGFPE 型名	IEEE 型
FPE_INTDIV	
FPE_INTOVF	
FPE_FLTRES	不正確
FPE_FLTDIV	除算

表 4-5 算術例外の型 (続き)

SIGFPE 型名	IEEE 型
FPE_FLTUND	アンダーフロー
FPE_FLTINV	無効
FPE_FLTOVF	オーバーフロー

この表に示されているように、各 IEEE 浮動小数点例外の型には SIGFPE シグナル型が対応しています。整数のゼロによる除算 (FPE_INTDIV) および整数オーバーフロー (FPE_INTOVF) は SIGFPE の型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` でハンドラを設定することはできません。これらの SIGFPE 型のハンドラは `sigfpe(3)` で設定できます。ただし、整数オーバーフローは、デフォルトでは SPARC および x86 プラットフォーム のすべてのシステムで無視されます。特殊な命令によって FPE_INTOVF 型の 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、`sig->si_code` のメンバーは、SPARC システムではどのような例外が発生したかを示しますが、x86 プラットフォームではフラグが立ったもっとも優先度の高い例外を示します (非正規オペランドフラグを除く)。`sig->si_addr` のメンバーは、SPARC 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 プラットフォームではトラップされた時点の命令 (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、パラメータ `uap` 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。メンバーの定義については、`sys/reg.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して、発生した例外の型と例外の原因である命令のアドレスを報告する SIGFPE ハンドラを作成することができます。以下のコード例 4-1 で、このようなハンドラの例を示します。

コード例 4-1 SIGFPE ハンドラ

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

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

#ifdef i386
    unsigned    sw;

    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status &
        ~uap->uc_mcontext.fpregs.fp_reg_set.fpship_state.state[0];
    if (sw & (1 << fp_invalid))
        code = FPE_FLTINV;
    else if (sw & (1 << fp_division))
        code = FPE_FLTDIV;
    else if (sw & (1 << fp_overflow))
        code = FPE_FLOVF;
    else if (sw & (1 << fp_underflow))
        code = FPE_FLTUND;
    else if (sw & (1 << fp_inexact))
        code = FPE_FLTRES;
    else
        code = 0;
    addr = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.
        state[3];
#else
    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
#endif
    fprintf(stderr, "fp exception %x at address %x\n", code,
        addr);
}

int main()
{
    double    x;

    /* 共通の浮動小数点例外をトラップします */
    if (ieee_handler("set", "common", handler) != 0)
        printf("Did not set exception handler\n");
}
```

コード例 4-1 SIGFPE ハンドラ (続き)

```
/* アンダーフロー例外を発生させます(報告されません) */
x = min_normal();
printf("min_normal = %g\n", x);
x = x / 13.0;
printf("min_normal / 13.0 = %g\n", x);

/* オーバーフロー例外を発生させます(報告されます) */
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)
```

[日本語訳]

注: 以下の IEEE 浮動小数点例外が発生しました:
不正確、アンダーフロー

以下の IEEE 浮動小数点例外のトラップが有効です:
オーバーフロー、ゼロによる除算、無効な演算

詳細は、『数値計算ガイド』の `ieee_flags(3M)`, `ieee_handler(3M)` に関する説明を参照してください。

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を保存し、SIGFPE ハンドラを呼び出す前にそれをクリアします。ハンドラによって保存されない限り、累積例外フラグはハンドラから戻されたときに失われます。したがって、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)
```

【日本語訳】

注：以下の IEEE 浮動小数点例外のトラップが有効です：

オーバーフロー、ゼロによる除算、無効な演算

詳細は、『数値計算ガイド』の `ieee_handler(3M)` に関する説明を参照してください。

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が IEEE デフォルトの結果を発生させる必要はありません。上記の出力では、`max_normal * max_normal` で通知される値は、オーバーフローする演算のデフォルトの結果 (正確な符号付き無限大) ではありません。通常は、意味のある値を出す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。その方法の例は、89 ページの「例外処理」を参照してください。

libm9x.so の例外処理機能を使用して例外を検出する

C および C++ プログラムでは、`libm9x.so` に含まれる C99 浮動小数点環境関数の例外処理機能を使用し、いくつかの方法で例外を検出できます。これらの拡張機能には、`ieee_handler` による処理とまったく同様に、ハンドラを確立して同時にトラップを有効にする関数が含まれますが、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、浮動小数点例外についての遡及診断メッセージのログ記録もサポートします。

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種類を処理するいくつかのオプション (モード) の 1 つを選択できます。fex_set_handling の呼び出しの構文は次のとおりです。

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

引数 *ex* は、呼び出しを適用する一連の例外を示します。この引数は、次の表 4-6 の最初の列に示された値のビット単位の論理和でなければなりません。これらの値は、fenv.h で定義されています。

表 4-6 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	ゼロ除算
FEX_INV_ZDZ	0/0 無効演算
FEX_INV_IDI	無限大/無限大の無効演算
FEX_INV_ISI	無限大-無限大の無効演算
FEX_INV_ZMI	0 x 無限大の無効演算
FEX_INV_SQRT	負の数の平方根
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

fenv.h は、便宜上、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、ゼロ除算、およびすべての無効な演算)、および FEX_ALL (すべての例外) の各値も定義しています。

引数 *mode* は、示された例外について例外処理モードを確立することを指定します。次に可能な 5 つのモードを示します。

- **FEX_NONSTOP** モードは、IEEE 754 のデフォルトの連続動作を提供します。これは、例外のトラップを無効にしておくのと同じです。ieee_handler と異なり、fex_set_handling では、無効演算例外の一定の種類に対しデフォルト以外の処理を確立し、残る種類に IEEE のデフォルト処理を当てることができます。
- **FEX_NOHANDLER** モードは、ハンドラを提供せずに例外のトラップを有効にするのと同じです。例外が発生すると、システムは、前回インストールした SIGFPE ハンドラが存在する場合はこ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、引数 handler で指定された処理関数をインストールします。これらの例外のどれかが発生すると、ieee_handler によってインストールされたかのように、同じ引数によっ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 で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によってハンドラが呼び出されます。この引数は、整数 (値は83 ページの表 4-6 に示された値の 1 つ) と、例外の原因となった演算についての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、次の節と flex_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 である場合は、*handler* パラメータは無視されることに注意してください。

fex_set_handling は、示された例外に対して指定されたモードが確立される場合はゼロ以外を返し、それ以外ではゼロを返します (次の例では、戻り値は無視される)。

次の例は、fex_set_handling を使用して特定の種類の例外を検出する方法を示しています。0/0 例外を停止するには、次のように記述します。

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

オーバーフローとゼロ除算に対して SIGFPE ハンドラをインストールするには、次のように記述します。

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

前記の例では、前の節で示しているように、SIGFPE ハンドラに対する *sip* パラメータを介して供給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供給される例外についての情報を出力します (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)。

コード例 4-2 FEX_CUSTOM モードでインストールされたハンドラに供給される情報の出力

```
#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_cnv:
    }
```

コード例 4-2 FEX_CUSTOM モードでインストールされたハンドラに供給される情報の出力 (続き)

```
        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);
```

上記例のハンドラは、発生した例外の種類、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、例外の発生場所は示しません。例外の発生場所を見つけるには、遡及診断を使用できます。

遡及診断

`libm9x.so` の例外処理機能を使用して例外を検出するもう 1 つの方法として、浮動小数点例外についての遡及診断メッセージのログ記録を有効にできます。遡及診断のログ記録を有効にすると、システムは特定の例外についての情報を記録します。この情報には、例外の種類、その原因となった命令のアドレス、その処理方法、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しか示されません。行番号、ソースファイル名、引数の値のような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。

遡及診断ログには、発生する例外ごとの情報は含まれません。例外ごとの情報を記録するとなると、巨大なログとなり、異常な例外を抜き出すことは不可能になります。代わりに、ロギング機構は冗長なメッセージは取り除きます。メッセージは、次の 2 つの状況のいずれかにある場合、冗長と見なされます。

- 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、直前に同じ例外が記録されている。
- 例外に対して `FEX_NONSTOP` モードが有効になっており、そのフラグが直前に発生した。

具体的には、ほとんどのプログラムでは、例外のそれぞれの種類が最初に発生する場合だけ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、その例外の次の発生は、直前にログ記録された位置での発生ではない場合だけログに記録されます。

ログ記録を有効にするには、`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);
```

次の例では、遡及診断のログ記録を、前の節に示されている共有オブジェクトのプリロード機能と組み合わせています。次の C ソースファイルを作成し、それを共有オブジェクトにコンパイルし、LD_PRELOAD 環境変数内でそのパス名を指定してその共有オブジェクトをプリロードし、FTRAP 環境変数で 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停止すると同時に、各例外がどこで発生したかを示す遡及診断出力を得ることができます。

コード例 4-3 遡及診断のログ記録と共有オブジェクトのプリロード機能との組み合わせ

```
#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;
        }
    }
```

コード例 4-3 遡及診断のログ記録と共有オブジェクトのプリロード機能との組み合わせ
(続き)

```
    } while ((s0 = strtok(NULL, ",")) != NULL);

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

上記のコードをこの節の初めで示しているプログラム例とともに使用すると、次のような結果が出力されます (SPARC の場合)。

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

この出力は、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。

上記で触れたように、x86 プラットフォームにおいて共有オブジェクト内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。

典型的なログ出力を示した例については、付録 A を参照してください。また、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。

例外処理

歴史的に、(さまざまな理由から) 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ました。また、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異常終了するという環境に慣れていました。現在では、LAPACK などの高品質なソフトウェアパッケージでは、ゼロによる除算や無効な演算などの例外を回避し、入力を基準化 (スケール) してオーバーフローや結果が不正になる可能性のあるアンダーフローを除外するように設計されています。ただし、このように例外を処理する方法

は、どのような状況でも適切であるというわけではありません。例外を無視すると、あるプログラマが作成したプログラムやサブルーチンを、(ソースコードにアクセスできない) 他のプログラマが使用する場合に、問題となる場合があります。すべての例外を回避しようとする、多くのテストと分岐が必要になり、非常に手間がかかります (Demmel、Li 共著『Faster Numerical Algorithms via Exception Handling』IEEE Trans. Comput. 43、1994 年), pp. 983-992 を参照)。

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や状態フラグ、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後で例外を検出するか、発生時に解釈および処理することができます。前述のように、`ieee_flags` や C99 浮動小数点環境関数を使用して後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、発生時に例外を解釈する SIGFPE ハンドラを設定することができます。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で指定するように推奨しています。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供給している `uap` パラメータを使用して指定できます。
`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供給される `info` パラメータを使用して結果を提供できます。

C では、SIGFPE シグナルハンドラは次のように宣言します。

```
#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、そのコンピュータの整数レジスタおよび浮動小数点レジスタのコピーや、その他の例外が記述されているシステム依存の情報を格納した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解読し、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することができます。

FEX_CUSTOM モードハンドラは、次の方法で宣言できます。

```
#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、*ex* パラメータはどの種類の例外が発生したか (83 ページの表 4-6 に挙げられた値の 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示します。このデータ構造は、例外の発生原因である算術演算を表現するコードと、オペランドを記録する構造体 (利用できる場合) を含みます。また、例外がトラップされない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と、発生したはずの例外フラグのビット単位の論理和を保持する整数値も含みます。ハンドラは、この 2 つの情報を変更して異なる結果に置き換えたり、発生したフラグのセットを変更したりできます。これらのデータを変更することなくハンドラが戻る場合、プログラムは、例外がトラップされないかのように、デフォルトのトラップされない結果とフラグを使用して継続します。

次の節に、アンダーフローまたはオーバーフローになる演算を基準化 (スケール) された結果に置換する方法を示します。詳細は、付録 A を参照してください。

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、指数部がラップ (wrap) された結果をトラップハンドラによって置換できるような方法をシステムで提供するように推奨されています。指数部がラップされた結果は、指数部がその通常の範囲を超えてラップされているということを除けば、その値はオーバーフローまたはアンダーフローを起こさずに演算が行われた場合の結果と一致します。つまり、値が 2 のべき乗によって基準化 (スケール) されているということです。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、指数範囲の中央に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となった結果を割り当てるような倍率が選択されます。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基準化 (スケール) し、ラップされた指数を補正することができます。また、有効な浮動小数点フォーマットの範囲を超えてしまうような計算において、正確な結果を出すことができます (P. Sterbenz 著『Floating-Point Computation』を参照)。

SPARC アーキテクチャのシステムでは、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、アンダーフローまたは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調べ、基準化 (スケール) された結果自体を生成しなければなりません。次の例では、この手順を実行するハンドラを示します。このハンドラを UltraSPARC システムでコンパイルされたコードで使用するには、Solaris 2.6、Solaris 7、または Solaris 8 を実行するシステム 上でこのハンドラをコンパイルし、プリプロセッサトークン V8PLUS を定義します。

コード例 4-4 SPARC システムでの、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

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

#ifdef V8PLUS
/* 上位 32 ビットの浮動小数点レジスタは、uap->uc_mcontext.xrs.xrs_prt
   によって示される領域に格納されます。このポインタは、
   uap->mcontext.xrs.xrs_id == XRS_ID (sys/procfs.h で定義されてい
   る) の場合のみ有効です。*/
#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)]

/*
 *   トラップされたアンダーフローまたはオーバーフローについて、
 *   IEEE 754 のデフォルトの結果が提供されます
 */
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;
```

コード例 4-4 SPARC システムでの、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 (続き)

```
float      fs1, fs2, fd, fscl;

/* 例外の原因となっている命令を取得 */
instr = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

/* 演算コード、ソース、宛先レジスタ番号を抽出 */
opf = (instr >> 5) & 0x1ff;
rs1 = (instr >> 14) & 0x1f;
rs2 = instr & 0x1f;
rd = (instr >> 25) & 0x1f;

/* オペランドを取得 */
switch (opf & 3) {
case 1: /* single precision */
    fs1 = *(float*)&FPreg(rs1);
    fs2 = *(float*)&FPreg(rs2);
    break;

case 2: /* 倍精度 */
#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: /* 4 倍精度 */
#ifdef V8PLUS
```

コード例 4-4 SPARC システムでの、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 (続き)

```
        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;
}

/* 倍率を設定 */
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);
}

/* トラップを無効にして基準化(スケール)された結果を生成 */
fpsetmask(0);
switch (opf) {
    case 0x41: /* 単精度の加算 */
        fd = fscl * (fscl * fs1 + fscl * fs2);
        break;

    case 0x42: /* 倍精度の加算 */
        dd = dscl * (dscl * ds1 + dscl * ds2);
        break;
}
```

コード例 4-4 SPARC システムでの、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 (続き)

```
case 0x43: /* 4 倍精度の加算 */
    qd = qscl * (qscl * qs1 + qscl * qs2);
    break;

case 0x45: /* 単精度の減算 */
    fd = fscl * (fscl * fs1 - fscl * fs2);
    break;

case 0x46: /* 倍精度の減算 */
    dd = dscl * (dscl * ds1 - dscl * ds2);
    break;

case 0x47: /* 4 倍精度の減算 */
    qd = qscl * (qscl * qs1 - qscl * qs2);
    break;

case 0x49: /* 単精度の乗算 */
    fd = (fscl * fs1) * (fscl * fs2);
    break;

case 0x4a: /* 倍精度の乗算 */
    dd = (dscl * ds1) * (dscl * ds2);
    break;

case 0x4b: /* 4 倍精度の乗算 */
    qd = (qscl * qs1) * (qscl * qs2);
    break;

case 0x4d: /* 単精度の除算 */
    fd = (fscl * fs1) / (fs2 / fscl);
    break;

case 0x4e: /* 倍精度の除算 */
    dd = (dscl * ds1) / (ds2 / dscl);
    break;

case 0x4f: /* 4 倍精度の除算 */
    qd = (qscl * qs1) / (qs2 / qscl);
    break;

case 0xc6: /* 倍精度を単精度に変換 */
```

コード例 4-4 SPARC システムでの、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 (続き)

```

        fd = (float) (fsc1 * (fsc1 * ds1));
        break;

    case 0xc7: /* 4 倍精度を単精度に変換 */
        fd = (float) (fsc1 * (fsc1 * qs1));
        break;

    case 0xcb: /* 4 倍精度を倍精度に変換 */
        dd = (double) (dsc1 * (dsc1 * qs1));
        break;
}

/* 宛先に結果を格納 */
if (opf & 0x80) {
    /* 変換演算 */
    if (opf == 0xcb) {
        /* 4 倍精度を倍精度に変換 */
#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
        /* 4 倍精度/倍精度を単精度に変換 */
        *(float*)&FPreg(rd) = fd;
} else {
    /* 算術演算 */
    switch (opf & 3) {
    case 1: /* 単精度 */
        *(float*)&FPreg(rd) = fd;
        break;

        case 2: /* 倍精度 */
#ifdef V8PLUS
        if (rd & 1)
        {

```

コード例 4-4 SPARC システムでの、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 (続き)

```

        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: /* 4 倍精度 */
#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; /* オーバーフロー ; 適切な数にラップされる */
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );

```

コード例 4-4 SPARC システムでの、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 (続き)

```
a /= b; /* アンダーフロー ; 逆方向にラップされる */
printf( "%g\n", a );

x = y = 1.0e300;
x *= y; /* オーバーフロー ; 適切な数にラップされる */
printf( "%g\n", x );
x /= y;
printf( "%g\n", x );
x /= y; /* アンダーフロー ; 逆方向にラップされる */
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 などの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、以下のとおりです。

```
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)
```

【日本語訳】

注: 以下の IEEE 浮動小数点例外のトラップが有効です:

アンダーフロー、オーバーフロー

詳細は、『数値計算ガイド』の ieee_handler(3M)に関する説明を参照してください。

x86 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、浮動小数点のストア命令でアンダーフローまたはオーバーフローがトラップされる時には、ハードウェアはストアが未完了のまま

トラップを行います。また、その命令がストアおよびポップの命令である場合には、スタックのポップも行われません。このため、ストア命令においてトラップが発生した時のアンダーフローまたはオーバーフローの数を追跡するには、アンダーフローまたはオーバーフローのハンドラが基準化 (スケール) された結果を生成し、スタックを修正する必要があります。このようなハンドラを、以下の例で示します。

コード例 4-5 x86 システムでの 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

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

/* 保存された fp 環境へのオフセット */
#define CW      0      /* 制御ワード */
#define SW      1      /* ステータスワード */
#define TW      2      /* タグワード */
#define OP      4      /* 演算コード */
#define EA      5      /* オペランドのアドレス */

#define FEnv(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])

/*
 *   トラップされたアンダーフローまたはオーバーフローについて
 *   IEEE 754 デフォルトの結果を提供
 */
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    double      dscl;
    float        fscl;
    unsigned     sw, op, top;
    int          mask, e;

    /* トラップされなかった例外のフラグを保存 */
    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status;
```

コード例 4-5 x86 システムでの 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 (続き)

```
FPEnv(SW) |= (sw & (FPEnv(CW) & 0x3f));

/* 例外となった命令が記憶領域にある場合、スタックを一番上に基準化(スケール)して格納し、必要に応じてスタックをポップ */
fpsetmask(0);
op = FPEnv(OP) >> 16;
switch (op & 0x7f8) {
case 0x110:
case 0x118:
case 0x150:
case 0x158:
case 0x190:
case 0x198:
    fsc1 = scalbnf(1.0f, (sip->si_code == FPE_FLTOVF)?
        -96 : 96);
    *(float *)FPEnv(EA) = (FPreg(0) * fsc1) * fsc1;

    if (op & 8) {
        /* スタックをポップする */
        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);
```

コード例 4-5 x86 システムでの 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 (続き)

```
        *(double *)FPenv(EA) = (FPreg(0) * dscl) * dscl;
    if (op & 8) {
        /* スタックをポップする */
        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 );
}
```

コード例 4-5 x86 システムでの 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 (続き)

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

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

SPARC アーキテクチャのシステムでは、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)
```

【日本語訳】

注：以下の IEEE 浮動小数点例外のトラップが有効です：

アンダーフロー、オーバーフロー

詳細は、『数値計算ガイド』の `ieee_handler(3M)` に関する説明を参照してください。

C および C++ プログラムでは、`libm9x.so` に含まれる `fex_set_handling` 関数を使用して、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 システムでは、このようなハンドラに供給される情報には例外の原因である演算とオペランドが常に含まれています。上記に示しているように、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 では、例外を引き起こした演算、および超越命令の 1 つが例外を発生させる時点 (`info->op` パラメータが `fex_other` に設定されるなど。説明は `fenv.h` ファイルを参照) を、提供される情報が常に示すとはかぎりません。また、x86 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。

`fex_set_handling` 機能を使用すると、`FEX_CUSTOM` モードでインストールされているハンドラは、アンダーフローまたはオーバーフローする演算を IEEE 指数がラップされた結果に容易に置換できます。これらの例外のどちらかがトラップされる場合、指数がラップされた結果を配布することを示すため、ハンドラは次のようにセットできます。

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

次に、こ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; /* オーバーフロー ; 適切な数値にラップされる */

    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    a /= b; /* アンダーフロー ; 逆方向にラップされる */
    printf("%g\n", a);

    x = y = 1.0e300;
    x *= y; /* オーバーフロー ; 適切な数値にラップされる */
    printf("%g\n", x);
    x /= y;

    printf("%g\n", x);
    x /= y; /* アンダーフロー ; 逆方向にラップされる */
    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
```

付録 A

例

この付録では、よく行われる作業の例を示します。例は FORTRAN、および ANSI C で書かれており、その多くは現バージョンの `libm` と `libsunmath` に依存しています。これらの例は、Solaris オペレーティングシステム上の C および FORTRAN コンパイラでテストされています。

IEEE の演算機能

浮動小数点数の 16 進表現の検証方法を例で示します。格納されたデータの 16 進表現を見るには、デバッガを使用することもできます。

以下の C のプログラムでは、倍精度の近似値を π と単精度の無限大に出力します。

コード例 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;

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

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

    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。

```
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。

コード例 A-2 最小の正規数をそれぞれの形式で出力

```
      program print_ieee_values
c
c implicit 文は、floatingpoint 疑似組み込み関数が、
c 正しい型で宣言されているかを確認します。
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 では、対応する出力が次のようになります。

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

数学ライブラリ

この節では、数学ライブラリの関数を使用した例を示します。

乱数生成

次の例では、数の配列を生成する乱数関数を呼び出し、与えられた数の EXP を計算するのにかかる時間を測定する関数を使用します。

コード例 A-3 乱数生成

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

#define SIZE 400000

      program example
c
      implicit GENERIC (a-h,o-z)
      GENERIC x(SIZE), y, lb, ub
      real time(2), u1, u2
c
c [-ln2/2,ln2/2] における乱数での EXP を計算
      lb = -0.3465735903
      ub = 0.3465735903
c
c 乱数の配列を生成
#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 クロック開始
      call dtime(tarray)
      u1 = tarray(1)
```

コード例 A-3 乱数生成 (続き)

```
c
c 指数を計算
    do 16 i=1,SIZE
        y = exp(x(i))
    16  continue
c
c 経過時間の取得
    call dtime(time)
    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
```

前記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呼び出すようにソースコードを接尾辞 F (f ではない) のついたファイルに入れ、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。

以下の例では、`d_addrans` を使用して、ユーザーが指定した範囲で均一に分散する乱数データのブロックを発生させる方法を示します。

コード例 A-4 `d_addrans` を使用する

```
/*
 * sin への SIZE*LOOPS 乱数引数をテストします。
 * 範囲は [0、しきい値] とし、
 * しきい値 = 3E3000000000000000 (3.72529029846191406e-09) とします。
 */

#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;

    /* 乱数関数を初期化 */
    d_init_addrans_();

    /* sin について (SIZE * LOOPS) 個の引数をテスト */
    for (j = 0; j < LOOPS; j++) {

        /*
         * 長さ SIZE の乱数のベクトル x を生成し、
         * 三角関数の関数への入力として使用
         */
        n = SIZE;
        ub = upperbound.d;
        lb = 0.0;
        d_addrans_(x, &n, &lb, &ub);
    }
}
```

コード例 A-4 d_addrans を使用する (続き)

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

        /*sin(x) == x ?    x が小さい場合になる */
        for (i = 0; i < n; i++)
            if (x[i] != y[i])
                printf(
                    " OOPS: %d  sin(%18.17e)=%18.17e \n",
                    i, x[i], y[i]);
    }
    printf(" comparison ended; no differences\n");
    ieee_retrospective_();
    return(0);
}
```

IEEE が推奨する関数

次の FORTRAN の例では、IEEE 標準が推奨する関数を使用しています。

コード例 A-5 IEEE が推奨する関数

```
c
c   このプログラムでは、IEEE 推奨の関数のうちの 5 つを FORTRAN から呼び出す
c   方法を示します。
c   たとえば、y=x*2**n を行うには、ハードウェアでは数値が基数 2 で記憶され
c   るため、指数を n だけシフトします。
c   浮動小数点の演算機能に関する IEEE 規格では、これらの関数を必須にはしてい
c   ませんが、IEEE 浮動小数点環境には含めることを勧めています。
c
c   以下の項目も参照してください。
c
c   ieee_functions(3m)
c   libm_double(3f)
c   libm_single(3f)
c
c   ここでは次の 5 つの関数について例を示します：
c
c   ilogb(x): x の基数 2 でのバイアスされていない指数を整数形式で返す
c   signbit(x): 0 または 1 の符号ビットを返す
c   copysign(x,y): y の符号ビットを付けた x を返す
c   nextafter(x,y): x から y 方向に次に出現する表現可能な数値を返す
```

コード例 A-5 IEEE が推奨する関数 (続き)

```

c  scalbn(x,n): x * 2**n
c
c  関数                倍精度                単精度
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)
c
c
c  program ieee_functions_demo
c  implicit double precision (d)
c  implicit real (r)
c  double precision      x, y, z, direction
c  real                  r, s, t, r_direction
c  integer               i, scale
c
c  print *
c  print *, 'DOUBLE PRECISION EXAMPLES:'
c  print *
c
c  x = 32.0d0
c  i = id_ilogb(x)
c  write(*,1) x, i
1  format(' The base 2 exponent of ', F4.1, ' is ', I2)
c
c  x = -5.5d0
c  y = 12.4d0
c  z = d_copysign(x,y)
c  write(*,2) x, y, z
2  format(F5.1, ' was given the sign of ', F4.1,
c  *      ' and is now ', F4.1)
c
c  x = -5.5d0
c  i = id_signbit(x)
c  print *, 'The sign bit of ', x, ' is ', i
c
c  x = d_min_subnormal()
c  direction = -d_infinity()
c  y = d_nextafter(x, direction)
c  write(*,3) x
3  format(' Starting from ', 1PE23.16E3,

```

コード例 A-5 IEEE が推奨する関数 (続き)

```
-    ', 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
```

コード例 A-5 IEEE が推奨する関数 (続き)

```
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ようになります。

コード例 A-6 コード例 A-5 のプログラムからの出力

```
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 -5.5000000000000000 is 1
Starting from 4.9406564584124654-324, the next representable
number towards -Inf is 0.0000000000000000E+000
Starting from 4.9406564584124654-324, the next representable
number towards 1.0 is 9.8813129168249309-324
Scaling 2.0 by 2**3 is 16.0

SINGLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is 5
The sign bit of -5.50000 is 1
-5.5 was given the sign of 12.4 and is now 5.5
Starting from 1.4012984643248171E-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 互換オプションを指定して、f95 コンパイラを使用すると、次のメッセージが表示されます。

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

【日本語訳】

注: 以下の IEEE 浮動小数点例外が発生しました:

不正確、アンダーフロー

詳細は、『数値計算ガイド』の `ieee_flags(3M)` に関する説明を参照してください。

IEEE の特殊な値

以下の C プログラムでは、`ieee_values(3m)` の関数を呼び出しています。

```
#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` と `-lm` の両方を指定してください。

SPARC では、出力は次のようになります。

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

x86 アーキテクチャは「リトルエンディアン」であるため、x86 での出力は少々異なります (倍精度数の 16 進表現で上位と下位のワードが逆になります)。

```
quiet NaN: NaN = ffffffff 7fffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                        = ffffffff 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、関数の型を宣言するよう
に注意しなければなりません。

```
      program print_ieee_values
c
c implicit 文は、floatingpoint の疑似組み込み関数が、
c 正しい型で宣言されているかどうかを確認します。
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。

```
6.4751751194380251109244389582276466-4966
Inf
3.40282E+38
```

ieee_flags – 丸め方向

以下の例は、丸めモードをゼロ切り捨てモード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;

    /* 一般的な丸めの方向を取得する */
    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);

    /* 丸め方向の設定 */
    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);
    /*
     * printf も現在の丸め方向に影響を受けるため、
     * printf の前に元の丸め方向を復元する
     */
    if (ieee_flags("set", "direction", out_1, &dummy) != 0)
        printf("Not able to change rounding direction!\n");
    printf("\nWith rounding direction %s, \n", out_2);
    printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
        ((int *) &x)[0], ((int *) &x)[1], x);

    return 0;
}
```

SPARC では、この短いプログラムの出力は、ゼロ切り捨てモードの効果を示します。

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

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

x86 では、この短いプログラムの出力は、ゼロ切り捨てモードの効果を示します。

```
demo% cc rounding_direction.c -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 プログラムでゼロ切り捨てモードを設定します。

```
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
```

出力は以下のようになります。

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

プログラムが `-f77` 互換オプションを指定した `f95` コンパイラを使用してコンパイルされる場合、さらに以下のメッセージが出力されます。

```
demo% f95 -f77 ieee_flags_demo.f
ieee_flags_demo.f:
  MAIN ieee_flags_demo:
demo% a.out
Rounding direction is: tozero
Note: Rounding direction toward zero
See the Numerical Computation Guide, ieee_flags(3M)
[日本語訳]
注: ゼロ方向への丸め
詳細は、『数値計算ガイド』の ieee_flags(3M) に関する説明を参照してください。
```

C99 浮動小数点環境関数

次に、C99 浮動小数点環境関数をいくつか使用した例を示します。`norm` 関数は、アンダーフローと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正規形を計算し、C99 浮動小数点環境関数を使用します。メインプログラムは、遡及診断出力が示すように、アンダーフローとオーバーフローを発生させるようにスケールされたベクトルを使用してこの関数を呼び出します。

コード例 A-7 C99 浮動小数点環境関数

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

/*
 * 早すぎるアンダーフローまたはオーバーフローを避けるベクトル x の
 * ユークリッド正規形を計算します
 */
double norm(int n, double *x)
{
    fenv_t env;
    double s, b, d, t;
    int i, f;

    /* 環境を保存してフラグをクリアし、連続した例外処理を確立します */
    feholdexcept(&env);
```

コード例 A-7 C99 浮動小数点環境関数 (続き)

```

/* ドット積 x.x の計算を試みます */
d = 1.0; /* 桁移動子 */
s = 0.0;
for (i = 0; i < n; i++)
    s += x[i] * x[i];

/* アンダーフローまたはオーバーフローを調べます */
f = fetestexcept(FE_UNDERFLOW | FE_OVERFLOW);
if (f & FE_OVERFLOW) {
    /* 最初の試行がオーバーフローしたら、スケーリングを
    小
    さくしてもう一度実行してください */
    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)) {
    /* 最初の試行がアンダーフローしたら、スケーリングを
    大
    きくしてもう一度実行してください */
    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;
    }
}

/* これまでに発生したアンダーフローをすべて隠します */
feclearexcept(FE_UNDERFLOW);

/* 環境を復元し、これまでに起きたほかの例外を発生させます */
feupdateenv(&env);

/* 平方根を取得し、スケーリングを解除します */
return d * sqrt(s);
}

```

コード例 A-7 C99 浮動小数点環境関数 (続き)

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

次の例は、x86 での fesetprec 関数の結果を示しています (この関数は SPARC では使用できません)。while ループは、1 に加えられるときに完全に丸められる、2 の最大の累乗を探すことにより、使用できる精度の決定を試みています。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 のようなアーキテクチャ

では、この手法が期待どおりに動作しない場合があります。そのため、2 番目のループが示すように、`fesetprec` 関数を使用して、すべての結果が希望する精度に丸められるように指定できます。

コード例 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;
}
```

x86 システムでは、このプログラムの出力は次のようになります。

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

次のコードフラグメントは、マルチスレッド対応の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え、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する例外フラグを回復する方法を示しています (マルチスレッド対応のプログラムの記述については、『Solaris でのマルチスレッドのプログラミング』を参照)。

コード例 A-9 マルチスレッド対応のプログラムで環境関数を使用する

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

fenv_t env;

void child(void *p)
{
    /* 入口で親の環境を継承します */
    fesetenv(&env);
    ...
    /* 終了前に子の環境を保存します */
    fegetenv(&env);
}

void parent()
{
    thread_t tid;
    void *arg;
    ...
    /* 子を作成する前に親の環境を保存します */
    fegetenv(&env);
    thr_create(NULL, NULL, child, arg, NULL, &tid);
    ...
    /* 子と結合します */
    thr_join(tid, NULL, &arg);
    /* 子で発生した例外フラグを親の環境にマージします */
    fex_merge_flags(&env);
    ...
}
```

例外と例外処理

ieee_flags – 例外

一般的には、例外ビットの検証やクリアはユーザープログラムで行います。自然発生した例外フラグを検証する C プログラムを示します。

コード例 A-10 例外ビットの検証

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

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

    /* アンダーフロー例外を発生させる */
    x = max_subnormal() / 2.0;

    /* この文は、前の文が最適化されていないことを示す */
    printf("x = %g\n", x);

    /* どの例外が発生したかを判定する */
    code = ieee_flags("get", "exception", "", &out);

    /* 戻り値を解説する */
    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" は発生した例外のうちでもっとも優先度が高い */
    printf(" Highest priority exception is: %s\n", out);

    /* 1 は例外が発生したことを示し、*/
    /* 0 は例外が発生していないことを示す */
    printf("%d %d %d %d %d\n", invalid, overflow, division,
    underflow, inexact);
    ieee_retrospective_();
    return(0);
}
```

このプログラムを実行した結果の出力です。

```
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)
[日本語訳]
注：以下の IEEE 浮動小数点例外が発生しました：
      不正確、アンダーフロー
詳細は、『数値計算ガイド』の ieee_flags(3M)に関する説明を参照してください。
```

FORTRAN でも同じことが行えます。

コード例 A-11 例外ビットの検証 (FORTRAN)

```
/*
これは、次のような FORTRAN プログラム例 です：
  * アンダーフロー例外を発生させる
  * ieee_flags を使用して、どの例外が発生したかを判定する
  * ieee_flags が返す整数値を解釈する
  * 未処理の例外をすべてクリアにする
c のプリプロセッサが起動され、ヘッダーファイル floatingpoint.h が取り込
まれるようにするために、このプログラムは接尾辞 .F のファイルに保存してくださ
い。
*/
#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 アンダーフロー例外を発生させる
      x = d_max_subnormal() / 2.0

c どの例外が発生したかを判定する
      accrued = ieee_flags('get', 'exception', '', out)

c ビットシフトの組み込みを使用して、ieee_flags が返した値を解釈する
      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 最高の優先度をもつ例外が "out" に返される
      print *, "Highest priority exception is ", out

c 1 は例外が発生したことを示し、0 は例外が発生していないことを示す
      print *, inv, over, div, under, inx

c 未処理の例外をすべてクリアにする
      i = ieee_flags('clear', 'exception', 'all', out)
      end
```

この FORTRAN プログラムの出力です。

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

通常、ユーザープログラムで例外フラグを**設定**することはありませんが、できないことではありません。以下の C の例でそれを示します。

```
#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 get 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、前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
```

ieee_handler – 例外のトラップ

注 – 以下の例は、Solaris オペレーティング環境だけに適用できます。

SPARC では、例外を特定するためのシグナルハンドラを設定する FORTRAN および C プログラムの例を示します。

FORTTRAN プログラム例

コード例 A-12 アンダーフローでのトラップ (SPARC)

```
program demo

c シグナルハンドラ関数の宣言
  external fp_exc_hdl
  double precision  d_min_normal
  double precision  x

c シグナルハンドラの設定
  i = ieee_handler('set', 'common', fp_exc_hdl)
  if (i.ne.0) print *, 'ieee trapping not supported here'

c アンダーフロー例外を発生させる (トラップはされない)
  x = d_min_normal() / 13.0
  print *, 'd_min_normal() / 13.0 = ', x

c オーバーフロー例外を発生させる
c 出力された値は結果とは関係ない
  x = 1.0d300*1.0d300
  print *, '1.0d300*1.0d300 = ', x

end

c
c 浮動小数点例外処理関数
c
  integer function fp_exc_hdl(sig, sip, uap)
  integer sig, code, addr
  character label*16

c
c 構造体 /siginfo/ は <sys/siginfo.h> による siginfo_t の解釈
```

コード例 A-12 アンダーフローでのトラップ (SPARC) (続き)

```
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 FPE コードの一覧は <sys/machsig.h> を参照
c SIGFPE 名を検出
    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 発生したシグナルに関する情報を出力
    write (*,77) code, label, addr
77  format ('floating-point exception code ', i2, ', ',
*         a17, ', ', ' at address ', z8 )

    end
```


出力は次のとおりです。

```
d_min_normal() / 13.0 =      1.7115952757748-309
floating-point exception code 4, overflow      , at address
1131C
1.0d300*1.0d300 =      1.00000000000000+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_handler(3M),
ieee_flags(3M)
[日本語訳]
注: 以下の IEEE 浮動小数点例外が発生しました:
    不正確、アンダーフロー
以下の IEEE 浮動小数点例外のトラップが有効です:
    オーバーフロー、ゼロによる除算、無効な演算
詳細は、『数値計算ガイド』の ieee_flags(3M), ieee_handler(3M) に関する
説明を参照してください。
```

C プログラム例

コード例 A-13 無効な演算、ゼロ除算、オーバーフロー、アンダーフロー、不正確結果でのトラップ(SPARC)

```
/*
 * 5 つの IEEE 例外（無効な演算、ゼロ除算、オーバーフロー、
 * アンダーフロー、不正確結果）を発生させます。
 * すべての浮動小数点例外をトラップし、
 * メッセージを出力してから処理を続けます。
 * i = ieee("get", "exception", "", &out);
 * 上記の式によって発生した例外について問い合わせをすることも可能です。
 * 発生した最高の例外名が含まれ、i はすべての例外を検出するように
 * 解釈されます。
 */

#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);
```

コード例 A-13 無効な演算、ゼロ除算、オーバーフロー、アンダーフロー、不正確結果でのトラップ(SPARC) (続き)

```
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 */
```

コード例 A-13 無効な演算、ゼロ除算、オーバーフロー、アンダーフロー、不正確結果でのトラップ(SPARC) (続き)

```
    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) {
```

コード例 A-13 無効な演算、ゼロ除算、オーバーフロー、アンダーフロー、不正確結果でのトラップ(SPARC) (続き)

```
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);
}
```

この C プログラムの結果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)
注: 以下の IEEE 浮動小数点例外が発生しました:
    不正確、アンダーフロー
以下の IEEE 浮動小数点例外のトラップが有効です:
    オーバーフロー、ゼロによる除算、無効な演算
詳細は、『数値計算ガイド』の ieee_flags(3M), ieee_handler(3M)に関する
説明を参照してください。
```

SPARC では、次のプログラムは、ある例外の状況から起きたデフォルト結果を修正するための `ieee_handler` とインクルードファイルの使用方法を示しています。

コード例 A-14 例外状況から起きたデフォルト結果を修正する

```
/*
 * ゼロ除算の例外を発生させ、シグナルハンドラを使用して結果を MAXDOUBLE
 * (または MAXFLOAT) で置き換えます。
 *
 * フラグ -Xa でコンパイルします。
 */

#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;

    /*
     * ieee_handler を使用し、division_handler を IEEE 例外
     * "division" が発生したときに使用するシグナルハンドラに設定する
     */
    if (ieee_handler("set","division",division_handler)!=0) {
        printf(" IEEE trapping not supported here.\n");
    }

    /* ゼロ除算の例外を発生させる */
    x = 1.0;
    y = 0.0;
    z = x / y;

    /*
     * ユーザーが提供した値 MAXDOUBLE が IEEE のデフォルト値である
     * 無限大の代わりに置換されているかどうかを確認する
     */
    printf("double precision division: %g/%g = %g \n",x,y,z);

    /* ゼロ除算の例外を発生させる */
    r = 1.0;
    s = 0.0;
    t = r / s;
```

コード例 A-14 例外状況から起きたデフォルト結果を修正する (続き)

```
/*
 * ユーザーが提供した値 MAXFLOAT が IEEE のデフォルト値である
 * 無限大の代わりに置換されているかどうかを確認する
 */
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;

    /* 例外を発生させる命令 */
    inst = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

    /*
     * 移動先のレジスタを復号化する。
     * ビット 29:25 は、SPARC 浮動小数点命令に対して
     * 移動先レジスタを符号化する
     */
    mask = 0x1f;
    rd = (mask & (inst >> 25));

    /*
     * これは単精度命令か倍精度命令か?
     * ビット 5:6 は opcode の精度を符号化する。
     * ビット 5 が 1 であれば sp となり、それ以外は dp となる
     */
    mask = 0x1;
    single_prec = (mask & (inst >> 5));

    /* ユーザーが定義した値を移動先レジスタに入れる */
    if (single_prec) {
        uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[rd] =
            f_val_p[0];
    } else {
```

コード例 A-14 例外状況から起きたデフォルト結果を修正する (続き)

```
        uap->uc_mcontext.fpregs.fpu_fr.fpu_dregs[rd/2] = d_val;
    }
}
```

期待される出力は次のとおりです。

```
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)
【日本語訳】
注: 以下の IEEE 浮動小数点例外のトラップが有効です:
    ゼロによる除算
詳細は、『数値計算ガイド』の ieee_handler(3M) に関する説明を参照してくだ
さい。
```

ieee_handler – 例外での異常終了

特定の浮動小数点例外の場合、ieee_handler を使用して強制的にプログラムを異常終了させることができます。

```
#include <floatingpoint.h>
    program abort
c
    ieeeer = ieee_handler('set', 'division', SIGFPE_ABORT)
    if (ieeeer .ne. 0) print *, ' ieee trapping not supported'
    r = 14.2
    s = 0.0
    r = r/s
c
    print *, 'you should not see this; system should abort'
c
    end
```

libm9x.so の例外処理機能

この節では、libm9x.so が提供する例外処理機能の一部の使用方法を示した例を取りあげます。最初の例は、数値 x と係数 $a_0, a_1, \dots, a_N$ 、および $b_0, b_1, \dots, b_{N-1}$ の場合に、関数 $f(x)$ とその最初の導関数 $f'(x)$ を評価するタスクに基づいています。この場合

f は連分数 $f(x) = a_0 + b_0/(x + a_1 + b_1/(x + \dots/(x + a_{N-1} + b_{N-1}/(x + a_N)\dots))$ です。IEEE 演算では、 f の計算は簡単です。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、標準によって指定されたデフォルトの値 (正しく符号が付いた無限大) が正しい結果を出します。一方、 f' の計算は、これを評価するもっとも簡潔なフォームが、削除可能な特異点を持てるため、 f の計算よりも困難です。計算中にこれらの特異点の 1 つが出現すると、不定形 $0/0$ 、 $0 \times$ 無限大、無限大/無限大の 1 つの評価が試みられます (これらの不定形はすべて無効な演算例外を発生します)。W. Kahan は、「前置換」という機能により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 (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することを許可する) を拡張したものです。libm9x.so 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し、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* パラメータが示すデータ構造に値を格納するだけで、ハンドラは例外演算の結果に対して任意の値を供給できます。次に、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例を示します。

コード例 A-15 FEX_CUSTOM ハンドラを使用して連分数とその導関数を計算する

```
#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();
}

/*
 * x の位置で係数 a[j] と b[j] によって指定される連分数を
 * 評価し、関数値を *pf、導関数の値を *pf1 で返します
 */
void continued_fraction(int N, double *a, double *b,
                        double x, double *pf, double *pf1)
{
    fex_handler_t    oldhdl; /* ハンドラの保存または復元のため */
    volatile double  t;
```

コード例 A-15 FEX_CUSTOM ハンドラを使用して連分数とその導関数を計算する (続き)

```
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;
    /* 変数 p に対する代入と f1 の評価の間で正しい順序づけを          維
     持するため、volatile 変数 t に対する次の代入が必要です */
    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;
}

/* 次の係数、x = -3、1、4、および 5 はすべて中間例外に出会います */
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;

    ferraiseexcept(FE_INEXACT); /* 不正確な演算のログ記録を防止します */
    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);
    }
}
```

コード例 A-15 FEX_CUSTOM ハンドラを使用して連分数とその導関数を計算する (続き)

```
        printf("f(%g) = %12g, f'(%g) = %12g\n", x, f, x, f1);
    }
    return 0;
}
```

プログラムについてのコメントが順を追って付けられています。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、ゼロ除算に対する連続した例外処理と、3つの不定形に対する `FEX_CUSTOM` ハンドラを確立します。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。正しい値を供給して後続の `0*無限大` 無効演算を置換するように、`p` は関数を評価するループを通して毎回計算し直す必要があります。また、`p` はループ内で明示的に記述されていないため、コンパイラが `p` を削除しないように、`p` を `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演算 (これに対し `p` が前置換の値を提供する) より上または下に移動させないように、その演算の結果も `volatile` 変数 (プログラム内では `t` と呼ばれる) に代入します。`fex_setexcepthandler` の最後の呼び出しが、ゼロ除算と無効演算に対する本来の処理モードを復元します。

メインプログラムは、`fex_set_log` 関数を呼び出して、遡及診断のログ記録を有効にします。この呼び出しを行う前に、メインプログラムは不正確フラグを発生させます。これにより、不正確演算の記録を防ぎます (87 ページの「遡及診断」で説明しているように、`FEX_NONSTOP` モードでは例外のフラグが発生すると例外がログに記録されません)。メインプログラムはまた、共通例外に対して `FEX_ABORT` モードを確立し、`continued_fraction` によって明示的に処理されない異常な例外がプログラムを停止することを防ぎます。プログラムは、最後に数か所で特定の連分数を評価します。次の出力例が示すように、演算は中間例外に出会います。

```

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 の場合の計算 $f'(x)$ 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。)

前記のプログラムは、連分数とその導関数の評価で起きる例外を処理する方法として、もっとも効率がよい方法を示しているとは言えません。その 1 つの理由は、必要の有無にかかわらず、ループが繰り返されるごとに前置換の値を計算し直す必要があることです。この場合、前置換の値の計算には浮動小数点除算が含まれますが、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。また、ループ自体にすでに 2 つの除算が含まれており、ほとんどの SPARC および x86 プロセッサは 2 つの除算演算の例外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いのです。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 つの除算だけですむようにループを書き直すことができます。実際、前置換値の計算は除算を必要としません (このようにループを書き直すには、 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、複数の除算を含む演算のボ

トルネックは排除されますが、前置換値の計算に含まれるすべての算術演算が除かれるわけではありません。また、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、コンパイラがある種のキー操作を再要求することを防止するために欠かせませんが、コンパイラがほかの無関係な演算を再要求することも防止してしまいます。この例のように前置換を介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、更に効率よく処理することは可能でしょうか。

高速な前置換に対する特別なハードウェアサポートがない場合、この例における例外をもっとも効率よく処理するには、次の例に示すようにフラグを使用することでしょう。

コード例 A-16 例外処理にフラグを使用する

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

/*
 * x の位置で係数 a[j] と b[j] によって指定される連分数を
 * 評価し、関数値を *pf、導関数の値を *pf1 で返します
 */
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);
```

コード例 A-16 例外処理にフラグを使用する (続き)

```
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;
}
```

この例では、最初のループはデフォルトの連続モードで $f(x)$ と $f'(x)$ 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の外形をテストして $f(x)$ と $f'(x)$ を明示的に再計算します。通常、無効演算例外は発生しないため、プログラムは最初のループだけを実行します。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ないため、ループはコンパイラが可能とするかぎりの速度で実行

されます。この効率を得るためには、例外が発生するケースを処理するために、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しなければなりません。このトレードオフは、浮動小数点例外処理が直面する典型的なジレンマです。

FORTRAN プログラムでの libm9x.so の使用

libm9x.so は主に C および C++ プログラムでの使用を想定したのですが、Sun の FORTRAN 言語における相互運用性の機能を活用すれば、FORTRAN プログラムから一部の libm9x.so 関数を呼び出すことができます。

注 – 一貫した動作を保つため、libm9x.so の例外処理関数と、ieee_flags および ieee_handler 関数の両方を同じプログラム内で使用することは避けてください。

次に、前置換を使用して連分数とその導関数を評価する、FORTRAN によるプログラム例を示します (SPARC のみ)。

コード例 A-17 前置換を使用して連分数とその導関数を評価する (SPARC)

```
c
c 前置換ハンドラ
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
```

コード例 A-17 前置換を使用して連分数とその導関数を評価する (SPARC) (続き)

```
        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; 定数については <fenv.h> を参照してください。
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 x の位置で係数 a[j] と b[j] によって指定される連分数を
c 評価し、関数値を f、導関数の値を f1 で返します
c

subroutine continued_fraction(n, a, b, x, f, f1)
```


コード例 A-17 前置換を使用して連分数とその導関数を評価する (SPARC) (続き)

```

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

external fex_getexcepthandler, fex_setexcepthandler
external fex_set_handling, handler
c$pragma c(fex_getexcepthandler, fex_setexcepthandler)
c$pragma c(fex_set_handling)

c x'ff2' = FEX_DIVBYZERO | FEX_INVALID
call fex_getexcepthandler(oldhdl, %val(x'ff2'))

c x'2' = FEX_DIVBYZERO, 0 = FEX_NONSTOP
call fex_set_handling(%val(x'2'), %val(0), %val(0))

c x'b0' = FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI, 3 = FEX_CUSTOM
call fex_set_handling(%val(x'b0'), %val(3), handler)

f1 = 0.0d0
f = a(n+1)
do j = n, 1, -1
    d = x + f
    d1 = 1.0d0 + f1
    q = b(j) / d
    f1 = (-d1 / d) * q
c
c     変数 p に対する代入と f1 の評価の間で正しい順序づけ
c     を維持するため、volatile 変数 t に対する次の代入が
c     必要です

    t = f1
    p = b(j-1) * d1 / b(j)
    f = a(j) + q
end do

```

コード例 A-17 前置換を使用して連分数とその導関数を評価する (SPARC) (続き)

```
        call fex_setexcepthandler(oldhdl, %val(x'ff2'))
        return
    end
c
c   メインプログラム
c
    program cf
    integer          i
    double precision  a, b, x, f, f1
    dimension         a(5), b(4)
    data a /-1.0d0, 2.0d0, -3.0d0, 4.0d0, -5.0d0/
    data b /2.0d0, 4.0d0, 6.0d0, 8.0d0/

    external fex_set_handling
c$pragma c(fex_set_handling)

c x'ffa' = FEX_COMMON, 1 = FEX_ABORT
    call fex_set_handling(%val(x'ffa'), %val(1), %val(0))
    do i = -5, 5
        x = dble(i)
        call continued_fraction(4, a, b, x, f, f1)
        write (*, 1) i, f, i, f1
    end do
1 format('f(', I2, ') = ', G12.6, ', f''(', I2, ') = ', G12.6)
    end
```

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; 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)
【日本語訳】
注：以下の IEEE 浮動小数点例外が発生しました：
      不正確、ゼロによる除算、無効な演算
以下の IEEE 浮動小数点例外のトラップが有効です：
      オーバーフロー、ゼロによる除算、無効な演算
詳細は、『数値計算ガイド』の ieee_flags(3M), ieee_handler(3M)に関する
説明を参照してください。
```

その他

sigfpe – 整数例外のトラップ

前節では、`ieee_handler` を使用した例を示しました。一般的に、`ieee_handler` と `sigfpe` のどちらかの使用を選択する場合は、前者をお勧めします。

注 – `sigfpe` は、Solaris オペレーティング環境のみで使用可能です。

SPARCでは、たとえば、整数演算例外をトラップする際に使用するハンドラが `sigfpe` だとします。次の例は整数のゼロ除算でトラップし、結果をユーザーの指定した値に置き換えています。

コード例 A-18 整数例外のトラップ

```
/* 整数のゼロ除算例外を生成する */

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

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

int main() {
    int    a, b, c;

/*
 * sigfpe(3) を使用して、整数ゼロ除算で使用するシグナルハンドラとして
 * int_handler を設定する
 */

/*
 * 整数のゼロ除算に対するシグナルハンドラが設定されていないと、
 * 整数のゼロ除算は異常終了する
 */

    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);

/*
 * プログラムカウンタを増やす。オペレーティングシステムは、
 * これを浮動小数点例外に対して自動的に行う。
 * 整数のゼロ除算に対しては行わない。
 */

    uap->uc_mcontext.gregs[REG_PC] =
    uap->uc_mcontext.gregs[REG_nPC];
}
```

FORTRAN を呼び出す C

FORTRAN のサブルーチンを呼び出す C ドライバの例を示します。C と FORTRAN での動作に関する詳細は、適当な C および FORTRAN のマニュアルを参照してください。以下は C ドライバです (ファイル `driver.c` に保存します)。

コード例 A-19 FORTRAN を呼び出す C

```
/*
 * 以下の方法を示すデモプログラムです。
 *
 * 1. 配列引数を渡して、f77 サブルーチンを C から呼び出す方法
 * 2. 単精度 f95 関数を C から呼び出す方法
 * 3. 倍精度 f95 関数を C から呼び出す方法
 */

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;

    /* 配列を fortran 関数に渡す (配列の出力) */
    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");
    }
}
```

コード例 A-19 FORTRAN を呼び出す C (続き)

```
/* 単精度 fortran 関数を呼び出す */
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");

/* 倍精度 fortran 関数を呼び出す */
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);
}
```

ファイル drivee.f 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
```

それから、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
```


効果的なデバッグコマンド

表 A-1 は、SPARC アーキテクチャのデバッグコマンドの例です。

表 A-1 デバッグコマンド(SPARC)

処理	dbx	adb
ブレークポイントの設定		
関数	stop in myfunct	myfunct:b
行番号	stop at 29	
絶対アドレス		23a8:b
相対アドレス		main+0x40:b
ブレークポイントに来るまで実行	run	:r
ソースコードの表示	list	<pc,10?ia
浮動小数点レジスタの表示		
IEEE 単精度	print \$f0	<f0=X
等価の 10 進数	print -fx \$f0	<f0=f
IEEE 倍精度	print \$f0f1	<f0=X; <f1=X
等価の 10 進数	print -flx \$f0f1print -flx \$d0	<f0=F
全浮動小数点レジスタの表示	regs -F	\$x for f0-f15\$X for f16-f31
全レジスタの表示	regs	\$r; \$x; \$X
浮動小数点状態レジスタの表示	print -fx \$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
デバッガを終了	quit	\$q

表 A-2 は、x86 アーキテクチャのデバッグコマンドの例です。

表 A-2 デバッグコマンド (x86)

処理	dbx	adb
ブレークポイントの設定		
関数	stop in myfunct	myfunct:b
行番号	stop at 29	
絶対アドレス		23a8:b
相対アドレス		main+0x40:b
ブレークポイントに来るまで実行	run	:r
ソースコードの表示	list	<pc,10?ia
浮動小数点レジスタの表示	print \$st0... print \$st7	\$x
全レジスタの表示	examine &\$gs/19X	\$r
浮動小数点状態レジスタの表示	examine &\$fstat/X	<fstat=X または \$x
実行を継続	cont	:c
シングルステップ	step (or next)	:s
デバッガを終了	quit	\$q

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 つの方法を示します。最初の方法では、次のように記述するだけですみます。

```
myfunction:b
```

2 番目の方法では、myfunction に対応しているコード部の先頭に相当する絶対アドレスを調べた後、その絶対アドレスにブレークを設定します。

```
myfunction=X
      23a8
23a8:b
```

f95 でコンパイルされる FORTRAN プログラム内のメインサブルーチンは、adb への MAIN_ として知られています。adb の MAIN_ でブレークポイントを設定するには、次のように指定します。

```
MAIN_:b
```

浮動小数点レジスタの内容を調べる場合、dbx コマンドの &\$f0/X によって表示される 16 進数値は IEEE 表現であり、その数字の 10 進表現ではありません。SPARC では adb コマンドの \$x と %x は、16 進数表現と 10 進数値の両方を表示します。x86 では、adb コマンドの \$x は 10 進数のみを表示します。SPARC では倍精度の値については、奇数番号のレジスタの横に 10 進数値が表示されます。

プロセスが浮動小数点ユニットを使用するまで、それはオペレーティングシステムによって使用不可になっているため、デバッグしているプログラムがアクセスするまで、浮動小数点ユニットを変更することはできません。

SPARC では、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 ビットを占め (したがって 1 つのレジスタに収まる)、倍精度浮動小数点数は 64 ビットを占める (したがって倍精度数 1 つを保持するには 2 つのレジスタが使用される) ことを覚えておいてください。16 進数表現では、32 ビットは 8 桁の数字に相当します。adb で表示した浮動小数点ユニットでは、表示は次のような形で行われます。

< 浮動小数点レジスタ名 > < IEEE 16 進数の値 > < 単精度 > < 倍精度 >

SPARC では、3 番目の欄には、2 番目の欄に示された IEEE 16 進数を単精度の 10 進数に変換した値が保持されています。4 番目の欄では、レジスタの対を解釈しています。

SPARC では、f10 と f11 を 64 ビットの IEEE 倍精度数として解釈しています。1 つの倍精度値を保持するのに f10 と f11 を使うので、その値の最初の 32 ビットの (f10 行での) 7ff00000 は +NaN とは解釈されません。64 ビット 7ff00000 00000000 全体の解釈である +無限大は、意味のある解釈になります。

SPARC では、最初の 16 の浮動小数点データレジスタを表示するために使用されている `adb` コマンド `$x` は、`fsr` (浮動小数点ステータスレジスタ) も表示しています。

```
$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
```

10対応する *x86* での出力は、次のとおりです。

```
$x
80387 chip is present.
cw      0x137f
sw      0x3920
cssel 0x17 ipoff 0x2d93      dataset 0x1f dataoff 0x5740

st[0] +3.24999988079071044921875 e-1      VALID
st[1] +5.6539133243479549034419688 e73      EMPTY
st[2] +2.00000000000000008881784197      EMPTY
st[3] +1.8073218308070440556016047 e-1      EMPTY
st[4] +7.9180300235748291015625 e-1      EMPTY
st[5] +4.201639036693904927233234 e-13      EMPTY
st[6] +4.201639036693904927233234 e-13      EMPTY
st[7] +2.7224999213218694649185636      EMPTY
```

注 - (*x86* のみ) `cw` は制御ワード、`sw` はステータスワードです。

SPARC の動作と実装

この付録では、SPARC® ワークステーションで使用される浮動小数点ユニット (FPU) に関連する問題を説明します。特に、各ワークステーションに最適なコード生成フラグを確定する方法を示します。

浮動小数点ハードウェア

この節では、多数の SPARC® 浮動小数点ユニットを一覧で示し、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。浮動小数点のトラップ時に発生する状況、トラップされるアンダーフローとトラップされないアンダーフローの違い、IEEE 以外の (非標準) 算術モードの SPARC の実装での適切な処理などに関する簡単な説明については、『SPARC Architecture Manual, Version 8』の付録 N「SPARC IEEE 754 Implementation Recommendations」および『SPARC Architecture Manual, Version 9』の付録 B「IEEE std 754-1985 Requirements for SPARC-V9」を参照してください。

160 ページの表 B-1 に、SPARC ワークステーションで使用される浮動小数点ハードウェアを示します。初期の大部分の SPARC システムでは、TI または Weitek で開発されたモデルにもとづく浮動小数点ユニットが使用されています。

- TI ファミリには、TI8847 と TMS390C602A があります。
- Weitek ファミリには、1164/1165、3170、3171 があります。

これら 2 種類の FPU は他のワークステーションベンダーにライセンス提供されているため、SPARC ワークステーションによっては他の半導体メーカーのチップが使用される場合もあります。他メーカーチップの一部は、次の表にも示されています。

表 B-1 SPARC 浮動小数点オプション

FPU	説明またはプロセッサ名	適合するマシン	備考	最適な -xchip と -xarch
Weitek 1164/1165 ベースの FPU または FPU なし	カーネルが浮動小数点命令をエミュレートする	旧式	遅いため推奨できない	-xchip=old -xarch=v7
TI 8847 ベースの FPU	TI 8847; 富士通、または LSI 製コントローラ	Sun-4™ /1xx Sun-4/2xx Sun-4/3xx Sun-4/4xx SPARCstation® 1 (4/60)	1989 大部分の SPARCstation 1 ワークステーションは Weitek 3170 を搭載	-xchip=old -xarch=v7
Weitek 3170 ベースの FPU		SPARCstation 1 (4/60) SPARCstation 1+ (4/65)	1989, 1990	-xchip=old -xarch=v7
TI 602a		SPARCstation 2 (4/75)	1990	-xchip=old -xarch=v7
Weitek 3172 ベースの FPU		SPARCstation SLC (4/20) SPARCstation IPC (4/40)	1990	-xchip=old -xarch=v7
Weitek 8601 または Fujitsu 86903	統合化された CPU または FPU	SPARCstation IPX (4/50) SPARCstation ELC (4/25)	1991 IPX は 40 MHz CPU/FPU、ELC は 33 MHz を使用	-xchip=old -xarch=v7
Cypress 602	Mbus モジュールに存在	SPARCserver® 6xx	1991	-xchip=old -xarch=v7
TI TMS390S10(STP 1010)	microSPARC® -I	SPARCstation LX SPARCclassic	1992 ハードウェアに FsMULd は含まれない	-xchip=micro -xarch=v8a
Fujitsu 86904(STP1012)	microSPARC® -II	SPARCstation 4 and 5 SPARCstation Voyager	ハードウェアに FsMULd は含まれない	-xchip=micro2 -xarch=v8a

表 B-1 SPARC 浮動小数点オプション (続き)

FPU	説明またはプロセッサ名	適合するマシン	備考	最適な -xchip と -xarch
TI TMS390Z50(STP 1020A)	SuperSPARC-I	SPARCserver 6xx SPARCstation 10 SPARCstation 20 SPARCserver 1000 SPARCcenter 2000		-xchip=super -xarch=v8
STP1021A	SuperSPARC-II	SPARCserver 6xx SPARCstation 10 SPARCstation 20 SPARCserver 1000 SPARCcenter 2000		-xchip=super2 -xarch=v8
Ross RT620	hyperSPARC [®]	SPARCstation 10/HSxx SPARCstation 20/HSxx		-xchip=hyper -xarch=v8
Fujitsu 86907	TurboSPARC	SPARCstation 4 と 5		-xchip=micro2 -xarch=v8
STP1030A	UltraSPARC [®] -I	Ultra-1, Ultra-2 Ex00	V9+VIS	-xchip=ultra -xarch=v8plusa
STP1031	UltraSPARC-II	Ultra-2, E450 Ultra-30, Ultra-60, Ultra-80, Ex500 Ex000, E10000	V9+VIS	-xchip=ultra2 -xarch=v8plusa
SME1040	UltraSPARC-IIi	Ultra-5, Ultra-10	V9+VIS	-xchip=ultra2i -xarch=v8plusa
	UltraSPARC IIe	Sun Blade [™] 100	V9+VIS	-xchip=ultra2e -xarch=v8plusa
	UltraSPARC III	Sun Blade 1000 Sun Blade 2000	V9+VIS II	-xchip=ultra3 -xarch=v8plusa*

* V8plusb を指定して作成した実行ファイルは、UltraSPARC-II システムでしか動きません。どの UltraSPARC システム (I、II、III) でも実行できるようにするには、-xarch を v8plus に設定してください。

この表の最後の列は、それぞれの FPU でもっとも高速な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、2つの独立したコード生成属性を制御します。-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ます。-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ついてコンパイラが立てる仮定を決定します。SPARC 浮動小数

点ユニットはすべて、少なくとも『SPARC Architecture Manual, Version 7』で定義されている浮動小数点命令セットを実装しています。そのため、`-xarch=v7` を指定してコンパイルしたプログラムは、それ以降のプロセッサの機能を十分利用しない可能性があります、どの SPARC システムでも動作します。同様に、`-xchip` の特定の値で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 システムで動作しますが、指定されたプロセッサ以外のプロセッサを搭載したシステムでは実行速度が落ちる場合があります。

この表に挙げられた microSPARC-I よりも前の浮動小数点ユニットは、『SPARC Architecture Manual, Version 7』に定義された浮動小数点命令セットを実装しています。これらの FPU を使用したシステムで実行しなければならないプログラムは、`-xarch=v7` を指定してコンパイルする必要があります。コンパイラはこれらのプロセッサのパフォーマンス特性については特に仮定しないため、これらのプロセッサはどれも単一の `-xchip` オプション、`-xchip=old` を共有します。表 B-1 に示されたシステムの中には、現在本製品のコンパイラでサポートされていないものもあります。それらは、履歴を示すために挙げられているにすぎません。これらのシステムをサポートするコンパイラで使用するコード生成フラグについては、『数値計算ガイド』の該当する版を参照してください。

microSPARC-I および microSPARC-II 浮動小数点ユニットは、『SPARC Architecture Manual, Version 8』で定義されている浮動小数点命令セット (FsMULd および 4 倍精度命令を除く) を実装しています。`-xarch=v8` を指定してコンパイルしたプログラムはこれらのプロセッサが搭載されたシステム上で動作しますが、実装されていない浮動小数点命令はシステムカーネルがエミュレートしなければなりません。そのため、FsMULd を広範囲にわたって使用するプログラム (多数の単精度の複素数演算を行う FORTRAN プログラムなど) では、パフォーマンスが著しく低下する場合があります。このような事態を避けるには、これらのプロセッサを搭載したシステム用のプログラムを、`-xarch=v8a` を指定してコンパイルしてください。

SuperSPARC-I、SuperSPARC-II、hyperSPARC、および TurboSPARC 浮動小数点ユニットは、『SPARC Architecture Manual, Version 8』で定義されている浮動小数点命令セット (4 倍精度命令を除く) を実装しています。これらのプロセッサを搭載したシステムで最高のパフォーマンスを得るには、`-xarch=v8` を指定してコンパイルしてください。

UltraSPARC I、UltraSPARC II、UltraSPARC IIe、UltraSPARC Iii および UltraSPARC III 浮動小数点ユニットは、『SPARC Architecture Manual, Version 9』で定義されている浮動小数点命令セット (4 倍精度命令を除く) を実装しています。具体的には、これらの FPU は、32 倍精度の浮動小数点レジスタを提供します。コンパ

イラがこれらのレジスタを使用できるようにするには、`-xarch=v8plus` (32 ビット OS で動作するプログラム用) または `-xarch=v9` (64 ビット OS で動作するプログラム用) を指定してコンパイルしてください。これらのプロセッサは、標準の命令セットの拡張機能も提供しています。Visual Instruction Set (VIS) として知られる追加命令をコンパイラが自動的に生成す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。そのため、これらのプロセッサがサポートする命令セットをフルに利用するためには、`-xarch=v8plusa` (32 ビット) または `-xarch=v9a` (64 ビット) を使用してください。

`-xarch` および `-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展開されます)。デフォルトのコード生成オプションは、`-xtarget=generic` です。`-xarch`、`-xchip`、`-xtarget` の値の全一覧など、詳細については、`cc(1)`、`CC(1)`、`f95(1)` のマニュアルページとコンパイラマニュアルを参照してください。`-xarch` に関する詳細は『Fortran ユーザーズガイド』、『C ユーザーズガイド』、および『C++ ユーザーズガイド』にあります。

浮動小数点状態レジスタと待ち行列

どのバージョンの SPARC アーキテクチャを実装しているかにかかわらず、SPARC 浮動小数点ユニットはすべて、FPU に対応した状態ビットと制御ビットが入った浮動小数点状態レジスタ (FSR) を提供しています。遅延浮動小数点トラップを実装している SPARC FPU はすべて、現在実行中の浮動小数点命令についての情報を保持する浮動小数点待ち行列 (FQ) を提供しています。発生した浮動小数点例外を検出し、丸め方向、トラップ、および標準外演算モードを制御するように、FSR はユーザーソフトウェアからアクセスできます。FQ は、浮動小数点トラップの処理のために、オペレーティングシステムのカーネルによって使用されます。FQ は、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 FSR をメモリーから読み込む `LDFSR` 命令を介して、浮動小数点状態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。

<code>st</code>	<code>%fsr, [addr] ! FSR を指定されたアドレスに格納する</code>
<code>ld</code>	<code>[addr], %fsr ! FSR を指定されたアドレスから読み込む</code>

Compiler Collection のコンパイラに付属のライブラリが置かれたディレクトリに入ってい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 命令の使用を示す例が入っています。

次の図 B-1 は、浮動小数点状態レジスタのビットフィールドのレイアウトを示しています。

RD	res	TEM	NS	res	ver	ftt	qne	res	fcc	aexc	cexc
31:30	29:28	27:23	22	21:20	19:17	16:14	13	12	11:10	9:5	4:0

図 B-1 SPARC 浮動小数点状態レジスタ

SPARC アーキテクチャのバージョン 7 と 8 では、この図に示されているように 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、そのうちの下位 32 ビットはこの図と一致し、上位 32 ビットには 3 つの浮動小数点条件コードフィールドがさらに入っているだけで、大部分は未使用です。

この図中の `res` は予約済みのビットを示し、`ver` は FPU のバージョンを示す読み取り専用フィールドです。`ftt` と `qne` は、システムが浮動小数点トラップを処理する場合に使用します。残りのフィールドについては、次の表に示します。

表 B-2 浮動小数点状態レジスタフィールド

フィールド	内容
RM	丸め方向モード
TEM	トラップ有効化モード
NS	標準外モード
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、浮動小数点演算の丸め方向を指定する 2 ビットが入っています。NS ビットは、標準外演算モードを実装している SPARC FPU でこのモードを有効にします。実装していない FPU では、このビットは無視されます。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た浮動小数点条件コードが入っており、分岐演算と条件付き移動演算によって使用されます。TEM、aexc、および cexc フィールド

には、トラップの制御と、5 つの IEEE 754 浮動小数点例外のそれぞれについての累積例外フラグと現在の例外フラグの記録を行う 5 ビットが入っています。これらのフィールドは、表 B-3 に示すようにさらに分割されます。

表 B-3 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、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それぞれ意味します。

ソフトウェアサポートが必要な特別な場合

SPARC 浮動小数点ユニットは、通常はソフトウェアサポートを必要とすることなく完全にハードウェア内で命令を実行します。しかし、以下の 4 つの状況において、ハードウェアは浮動小数点命令を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。
- 命令がハードウェアによって実装されていない場合 (たとえば、Weitek 1164/1165 ベース FPU での `fsqrt [sd]`、microSPARC-I および microSPARC-II FPU での `fsmuld`、全 SPARC FPU での 4 倍精度命令など)。
- ハードウェアが命令のオペランドについて正しい結果を配布できない場合。
- 命令が IEEE 754 浮動小数点例外を引き起こし、その例外のトラップが有効である場合。

これらの状況での最初の応答はすべて、プロセスからシステムカーネルに対するトラップの発行です。システムカーネルは、トラップの原因を確かめ、適切な処置を行います (「トラップ」は、通常の制御フローの割り込みを意味します)。最初の 3 つの状況では、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。エミュレートされる命令もまた、トラップが有効になった例外を引き起こす可能性があることに注意してください。

上記の最初の3つの状況で、エミュレートされる命令がトラップが有効になった IEEE 浮動小数点例外を発生させない場合、カーネルは命令を完了します。命令が浮動小数点比較の場合は、その結果を反映させるために条件コードを更新します。命令が算術演算の場合は、宛先レジスタに対して適切な結果を配布します。カーネルは、命令によって発生した (トラップされない) 例外を反映させるために現在の例外フラグの更新も行い、それらの例外の論理和を累積例外フラグに入れます。その後カーネルは、トラップが起きた位置でプロセスの実行を継続するように手はずを整えます。

ハードウェアが実行する命令やカーネルソフトウェアがエミュレートする命令がトラップが有効になった IEEE 浮動小数点例外を発生させる場合、命令は完了されません。宛先レジスタ、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、現在の例外フラグはトラップの原因となった例外を反映するようにセットされ、カーネルは SIGFPE シグナルをプロセスに送ります。

次の疑似コードは、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ません。

```
FPop provokes a trap;
if trap type is fp_disabled, unimplemented_FPop, or
  unfinished_FPop then
  emulate FPop;
texc " FPop によって生成されるすべての IEEE 例外
if (texc and TEM) = 0 then
  f[rd] " fp_result; // fpop が算術演算の場合
  fcc " fcc_result; // ifpop が比較の場合
  cexc " texc;
  aexc " (aexc または texc);
else
  cexc " FPop によって生成されるトラップされた IEEE 例外
  throw SIGFPE;
```

多くの浮動小数点命令がカーネルによってエミュレートされなければならないとき、プログラムのパフォーマンスは大幅に低下します。この状況が発生する相対頻度は、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。

普通の状況では、fp_disabled トラップはプロセスあたり 1 度だけ発生します。システムカーネルは、プロセスがはじめて開始されるときに浮動小数点ユニットを無効にします。そのため、プロセスによって実行される最初の浮動小数点演算がトラップを発生させます。トラップを処理したあと、カーネルは浮動小数点ユニットを有効に

します。有効にされたユニットは、プロセスの間中、有効な状態が継続します。システム全体について浮動小数点ユニットを無効にすることも可能ですが、これはお勧めできません。カーネルまたはハードウェアのデバッグ目的にとどめてください。

`unimplemented_FPop` トラップは、実装していない命令に浮動小数点ユニットが会う場合に必ず発生します。現在の SPARC 浮動小数点ユニットのほとんどは、4 倍精度命令を除いて『SPARC Architecture Manual, Version 8』に定義されている命令は少なくとも実装しており、また **Compiler Collection** のコンパイラは 4 倍精度命令を生成しないため、ほとんどのシステムではこの種のトラップが発生することはありません。前述したように、`FsMULd` 命令を実装していない **microSPARC-I** および **microSPARC-II** プロセッサは例外ですので注意してください。この 2 つのプロセッサでの `unimplemented_FPop` トラップを避けるには、`-xarch=v8a` オプションを指定してプログラムをコンパイルしてください。

残る 2 種類のトラップ、`unfinished_FPop` およびトラップされた IEEE 例外は、通常、NaN、無限大、および非正規数がかかわる特殊な演算状況に関連しています。

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た IEEE 浮動小数点例外に出会うと、命令は完了されず、システムはプロセスに対して SIGFPE シグナルを配布します。プロセスがすでに SIGFPE シグナルハンドラを確立している場合にはそのハンドラが呼び出されますが、それ以外の場合プロセスは停止します。トラップを有効にするのは通常、例外発生時にプログラムを停止するためであることから、メッセージを出力してプログラムを停止するシグナルハンドラを呼び出すか、あるい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まとめ直すと、ほとんどのプログラムにおいてトラップされた IEEE 浮動小数点例外の発生が少なくなります。しかし、第 4 章で説明しているように、シグナルハンドラはトラップ命令に結果を供給して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てこの方法で処理される場合は、パフォーマンスが大幅に低下することに注意してください。

ほとんどの SPARC 浮動小数点ユニットは、少なくとも、無限オペランド、NaN オペランド、または IEEE 浮動小数点例外がかかわるケースでもトラップを起こします。これは、トラップが無効になっている場合や、トラップが有効になった例外を命令が引き起こさない場合でも同様です。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せず、`unfinished_FPop` トラップを生成し、カーネルエミュレーションソフトウェアが命令を完了するに任せる場合に発生します。`unfinished_FPop` トラップとなる条件は、SPARC FPU の種類によって異なります。たとえば、初期のほ

ほとんどの SPARC FPU や hyperSPARC FPU は、トラップが有効かどうかにかかわらずすべての IEEE 浮動小数点例外でトラップします。UltraSPARC FPU は、浮動小数点例外のトラップが有効で、しかも命令が例外を発生させるかどうかをハードウェアが決定できないという場合に「悲観的に」トラップします。一方、SuperSPARC-I、SuperSPARC-II、TurboSPARC、microSPARC-I、および microSPARC-II FPU ではすべての例外的なケースがハードウェアで処理され、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 は浮動小数点例外との組み合わせで発生するため、例外処理 (例外フラグのテスト、結果のトラップと置換、または例外発生時の停止) を行うことにより、過度のトラップの発生を避けることができます。もちろん、例外を処理する場合の労力と、例外が unfinished_FPop トラップとなるに任せる場合の労力のバランスをとるように注意してください。

非正規数と標準外演算

一部の SPARC 浮動小数点ユニットが unfinished_FPop トラップとなるもっとも一般的な状況として、非正規数が挙げられます。多くの SPARC FPU は、浮動小数点演算に非正規オペランドが含まれるか、浮動小数点演算がゼロ以外の非正規の結果 (段階的アンダーフローを起こす結果) を必ず生成する場合に常にトラップします。アンダーフローはさほど発生しませんが、プログラミングが困難です。また、アンダーフローされた中間結果の正確度は、演算の最終結果における全体的な正確度に通常はほとんど影響がありません。そのため、SPARC アーキテクチャには、非正規数がかかる unfinished_FPop トラップに関連したパフォーマンス低下を避ける方法をユーザーに提供する **標準外演算モード**が含まれています。

SPARC アーキテクチャは、標準外演算モードを明確には定義していません。SPARC アーキテクチャは、標準外演算モードが有効になるとき、このモードをサポートするプロセッサは IEEE 754 規格に準拠しない結果を生成できると述べているにすぎません。しかし、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、すべての非正規オペランドと非正規結果をゼロに置換します。1 つだけ例外があります。Weitek 1164/1165 FPU は、標準外モードで非正規結果をゼロにフラッシュするだけで、非正規オペランドをゼロとして扱うことはありません。

SPARC 実装の中には、標準外モードを提供しないものもあります。SuperSPARC-I、SuperSPARC-II、TurboSPARC、microSPARC-I、および microSPARC-II 浮動小数点ユニットは、完全にハードウェア内で非正規オペランドの処理と非正規結果の生成を

行うため、標準外演算をサポートする必要がありません (これらのプロセッサで標準外モードを有効にしても無視されます)。そのため、これらのプロセッサでは、段階的アンダーフローのために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を与えているかどうかを調べるには、まずアンダーフローが実際に発生しているかどうかを調べ、続いてプログラムがどれだけシステム時間を使用しているかをチェックする必要があります。アンダーフローの発生の有無を調べるには、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を見ることができます。FORTRAN プログラムは、デフォルトで `ieee_retrospective()` をコールします。C および C++ プログラムは、終了前に明示的に `ieee_retrospective()` を呼び出す必要があります。アンダーフローが発生すると、`ieee_retrospective()` は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出会う場合、`time` コマンドを使用してプログラム実行の時間を測ることにより、プログラムがどれだけシステム時間を使用しているか確認できます。

```
demo% /bin/time myprog > myprog.output
305.3 real      32.4 user      271.9 sys
```

システム時間 (上記例の 3 番目の数字) の値が異常に大きい場合、複数のアンダーフローが原因となっている場合があります。その場合、プログラムが段階的アンダーフローの正確度に依存していないときは、標準外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。この方法は 2 つあります。その 1 つは、プログラムの起動時に標準外モードを有効にするために、`-fns` フラグ (マクロ `-fast` と `-fnonstd` の一部として暗黙に示されます) を使用してコンパイルするものです。もう 1 つは、付加価値数学ライブラリ `libsunmath` が提供している、標準外モードを有効または無効にする 2 つの関数を使用するものです。`nonstandard_arithmetic()` を呼び出すと

標準外モードが有効になり (このモードが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。次に、これらの関数を呼び出す C と FORTRAN の構文を示します。

C, C++	<code>nonstandard_arithmetic();</code> <code>standard_arithmetic();</code>
FORTRAN	<code>call nonstandard_arithmetic()</code> <code>call standard_arithmetic()</code>



警告 – 標準外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、このモードは注意して使用する必要があります。段階的アンダーフローの詳細は、第 2 章を参照してください。

標準外演算とカーネルのエミュレーション

標準外モードを実装している SPARC 浮動小数点ユニットでは、このモードを有効にすると、ハードウェアは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。しかし、トラップされた浮動小数点命令をエミュレートするために使用されるカーネルソフトウェアは、標準外モードを実装していません。この理由の 1 つは、このモードの効果が定義されておらず実装に依存しているためです。もう 1 つの理由は、段階的アンダーフロー処理に費やす労力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労力に比べてわずかなものであるためです。

標準外モードにより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、発行されたが、コンテキストの切り替えが起きたり別の浮動小数点命令がトラップを起こしたりして完了しなかった場合など)、カーネルソフトウェアが標準の IEEE 演算を使用してそれをエミュレートします。そのため特別な状況では、標準外モードで動作しているプログラムが、システム負荷に応じて少々異なる結果を出す可能性があります。実際には、まだこのような動作は発見されていません。このような動作は、何百万回もの演算の中の 1 つの演算が段階的アンダーフローで実行されるか、それとも非段階的なアンダーフローで実行されるかということに非常に敏感なプログラムだけに影響すると思われます。

fpversion(1) 関数 — FPU に関する情報の検索

コンパイラと共に提供されている `fpversion` ユーティリティは、インストールされている CPU を識別し、プロセッサとシステムバスのクロック速度を予測します。`fpversion` は、CPU と FPU によって格納された識別情報を解釈することによって、CPU と FPU の型を確定します。また、予測可能な時間内に動作するシンプルな命令を実行するループの時間を測ることにより、それらのクロック速度を推定します。このループは、時間計測の正確度を上げるように何度も実行されるため、`fpversion` の結果は即座には出ません。実行には数秒かかります。

`fpversion` は、ホストシステムに使用する上で最適な `-xtarget` コード生成オプションも報告します。

Ultra 4 ワークステーションでは、`fpversion` は情報を次のように表示します。この表示は、タイミングやマシン構成の違いによって多少異なります。

```
demo% fpversion
A SPARC-based CPU is available.
CPU's clock rate appears to be approximately 461.1 MHz.
Kernel says CPU's clock rate is 480.0 MHz.
Kernel says main memory's clock rate is 120.0 MHz.

Sun-4 floating-point controller version 0 found.
An UltraSPARC chip is available.
FPU's frequency appears to be approximately 492.7 MHz.

Use "-xtarget=ultra2 -xcache=16/32/1:2048/64/1"
code-generation option.

Hostid = hardware_host_id
```

詳細は、`fpversion(1)` のマニュアルページを参照してください。

x86 の動作と実装

この付録では、x86 と SPARC の互換性問題のうち x86 プラットフォームで使用される浮動小数点ユニットに関連する部分について説明します。

対象となるハードウェアは、Intel 社の 80386、80486、Pentium™ マイクロプロセッサとその他のメーカーの互換マイクロプロセッサです。SPARC プラットフォームとの互換性には大きな努力が注がれていますが、SPARC とは以下のような相違点があります。

x86 では:

- 浮動小数点レジスタのサイズは 80 ビットです。数値計算の中間結果が拡張精度になるために、計算結果が異なる場合があります。-fstore フラグを使用すれば、このような矛盾が最小限に抑えられます。ただし、-fstore フラグを使用すると、性能が低下します。
- 単精度または倍精度の浮動小数点数のストアやロードが行われる度に、拡張倍精度へまたは拡張倍精度からの変換が発生します。したがって、浮動小数点数のロードやストアによって例外が発生します。
- 段階的アンダーフローが完全にハードウェア内に実装されています。標準外のモードはありません。
- fpversion ユーティリティは提供されていません。
- 拡張倍精度形式は、浮動小数点値を表現しない一定のビットパターンを認めています (15 ページの表 2-8 を参照)。ハードウェアは NaN のような「サポートされていない形式」を通常は取り扱いますが、数学ライブラリによるそのような表現の処理には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ので、無効なメモリー参照 (配列の終わりを越えた読み取りなど) が

行われるか、(C の union 構造体などを介して) メモリー内においてある型から別の型へ明示的にデータを強制変換する場合にしか作成されません。そのため、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発生しません。

浮動小数点演算について

注 - この付録は、1991 年 3 月発行の “Computing Surveys” に掲載された “Every Computer Scientist Should Know About Floating-Point Arithmetic” 稿 (David Goldberg 著) を再編集し、著作権を有する Association for Computing Machinery 社 (Copyright 1991) の許可のもとに、印刷しなおしたものです。

概要

浮動小数点演算は、概して難解な問題として受け取られることがあります。コンピュータシステムの内部には、浮動小数点演算がいたるところに存在する点を考えれば、この認識には多少、現実とのずれがあるようです。たとえば、ほとんどのコンピュータ言語に、浮動小数点のデータ型が使用されています。また PC からスーパーコンピュータに至るまであらゆるコンピュータで浮動小数点アクセラレータが使用されています。さらに浮動小数点アルゴリズムをコンパイルするために、コンパイラが呼び出されることがよくあります。現実には、オーバーフローなどの浮動小数点例外に対応していないオペレーティングシステムなど皆無であるとも言えます。この付録では、コンピュータシステムの設計担当者に対して直接的な影響を与える浮動小数点のさまざまな側面について説明します。具体的には、浮動小数点表現の背景、丸め誤差、IEEE 浮動小数点標準について説明します。最後にコンピュータ設計者が浮動小数点をどのようにサポートできるかを示す例をいくつか紹介します。

カテゴリ/サブジェクト記述子 (一次) C.0 [コンピュータシステムの編成] : 一般 - 命令セットの設計 ; D.3.4 [プログラミング言語] : プロセッサ - コンパイラ、最適化 ; G.1.0 [数値解析] : 一般 - コンピュータ演算、エラー解析、数値アルゴリズム (二次)

D.2.1 [ソフトウェアエンジニアリング]：要件/仕様－言語；D.3.4 [プログラミング言語]：正式な定義と理論－セマンティクス；D.4.1 [オペレーティングシステム]：プロセス管理－同期化

一般用語：アルゴリズム、設計、言語

その他のキーワードと表現：非正規化数、例外、浮動小数点、浮動小数点標準、段階的アンダーフロー、保護桁、NaN、オーバーフロー、相対誤差、丸め誤差、丸めモード、ulp、アンダーフロー

はじめに

コンピュータシステムの設計作業を進める上で、浮動小数点演算に関する情報が必要になることがよくあります。しかし現実には、浮動小数点演算に関する詳しい資料は意外なほど少ないものです。この問題を扱った数少ない書籍の1つである

“Floating-Point Computation” (Pat Sterbenz 著) も、すでに廃刊になっています。この付録では、コンピュータシステムの設計に大きくかわる浮動小数点演算の各側面について詳しく説明します。この付録は大きく 3 部に分けられます。177 ページの「丸め誤差」の部では、四則演算の基本的な操作に各種の丸めモードを適用した場合の影響について説明します。また ulp と相対誤差という 2 種類の丸め誤差の測定方法に関する背景情報を示します。2 部では、多くのハードウェアベンダー各社によって急速な勢いで採用されつつある IEEE 浮動小数点標準について説明します。IEEE 標準には、基本的な演算に関する丸めの方法が定義されています。IEEE 標準については、177 ページの「丸め誤差」に示す事項にもとづいて説明します。具体的には命令セットの設計、最適化コンパイラ、および例外処理について説明します。

この付録で浮動小数点について説明するときは、必ず、その説明が妥当であるという正当な理由も併せて示してあります。これは、その理由に関する説明自体が、単純な計算だけでは済ますことができない場合がよくあるからです。また説明の主旨と直接的には関係のない内容については「詳細」という項に独立して示してあります。「詳細」は読み飛ばしてもかまいません。また、この付録には特に、定理に関する「証明」が数多く紹介してあります。「証明」の終わりの箇所は■の記号を付けて表わします。また「証明」を省略した場合は、■の記号を定理の文の直後に示します。

丸め誤差

多数の実数を有限個のビットの中へ無限に圧縮していくには、近似値の表現が必要になります。整数には無限の数がありますが、プログラム中では通常、整数演算の結果は 32 ビットごとに格納されます。一方、任意の固定数ビットで、実数を伴う演算を行う場合は、どれほど多くのビットを使用しても、結果の値を正確には表現できなくなってきました。したがって、浮動小数点演算の結果に、丸め操作を加えることによって、有限の範囲内に収めて表現しなければなりません。この丸め操作のときに生じる誤差は、浮動小数点演算から切り離すことのできない特性と言えます。丸め誤差の測定方法については、179 ページの「相対誤差と ulp」を参照してください。

そもそも、通常の浮動小数点演算で丸め誤差の問題を避けることができないのだとすれば、基本的な演算処理で多少の丸め誤差が出ることに何か問題があるのでしょうか。この根本的な疑問こそ、この項で扱う主要な問題にほかなりません。181 ページの「保護桁」では保護桁について説明しています。保護桁を確保することにより、2 つの近い値を減算するときの誤差を低減することができます。保護桁は本来、IBM が 1968 年に、System/360 アーキテクチャの倍精度フォーマットとして保護用の桁を追加した (単精度フォーマットにはすでに保護桁が設定されていました) ことに対応して、既存の全マシンを改訂したことがきっかけで重要視されるようになりました。保護桁の効用を示す例を 2 つ紹介します。

IEEE 標準の仕様は、単なる保護桁の必要性を要求するだけのものではありません。四則演算のほか、平方根計算のアルゴリズムが用意され、各アルゴリズムと同じ結果を生成できる実装が必要になります。したがって、あるマシンから別のマシンにプログラムを移植した場合でも、両方のマシンで IEEE 標準がサポートされていれば、基本演算により各ビットごとに、まったく同じ結果が生成されることになります。187 ページの「正確な丸め操作」には、この正確な仕様にもとづいた別の例を示します。

浮動小数点フォーマット

これまでに、実数を表現するさまざまな形式が提案されてきました。このうちもっとも一般的な形式は浮動小数点表現¹です。浮動小数点表現では、基数 β (つねに偶数とします) と精度 p を使用します。 $\beta = 10$ で、 $p = 3$ とすると、0.1 という値は 1.00×10^{-1} と表わします。また $\beta = 2$ で、 $p = 24$ の場合、小数点数 0.1 は正確に表現することはできませんが、およそ $1.1001100110011001101 \times 2^{-4}$ となります。一般に、浮

1. これ以外の表現として、浮動小数点スラッシュ、および符号付き対数があります (Matula/Kornerup 1985、Swartzlander/Alexopoulos 1975)。

動小数点数は $\pm d.dd\dots d \times \beta^e$ と表わされます (ここで $d.dd\dots d$ は「有意桁」¹と呼ばれ、桁数は p となります)。さらに正確に言うと、 $\pm d_0.d_1d_2\dots d_{p-1} \times \beta^e$ は、次の数を表わします。

$$\pm(d_0 + d_1\beta^{-1} + \dots + d_{p-1}\beta^{-(p-1)})\beta^e, (0 \leq d_i < \beta) \quad (1)$$

浮動小数点数という用語は、上記のような形式で表現できる実数のことを指します。浮動小数点表現では、最大許容指数 $e_{\max}$ と最小許容指数 $e_{\min}$ という 2 つのパラメータを使用します。有意桁が βp 個で、許容指数が $e_{\max} - e_{\min} + 1$ とすれば、浮動小数点数は次のようにコーディングできます。

$$[\log_2(e_{\max} - e_{\min} + 1)] + [\log_2(\beta^p)] + 1$$

最後の +1 は符号ビットを表わします。ここでコーディングそのものは、重要ではありません。実数が浮動小数点数としては正確に表現されない理由が 2 つあります。もっとも一般的なケースは小数点数 0.1 の例です。これは有限の 10 進小数点数として表記されますが、2 進数では無限の反復表現になります。したがって、 $\beta = 2$ の場合、0.1 という数値は正確には 2 つの浮動小数点数の間に存在しており、いずれの値でも正しくは表現できないことになります。

また、別の状況として実数が有効範囲を超える場合、すなわち絶対値が $\beta \times \beta^{e_{\max}}$ 以上、あるいは $1.0 \times \beta^{e_{\max}}$ 以下になっているケースが考えられます。この付録では、最初の問題に限って扱います。なお、有効範囲を超える数値については、201 ページの「無限大」、および 203 ページの「非正規化数」の各項で説明します。

浮動小数点表現は、必ずしも一意の表現にはなりません。たとえば、 0.01×10^1 と 1.00×10^{-1} はいずれも 0.1 を表わします。先行桁がゼロ以外 (上記の公式 (1) で $d_0 \neq 0$ の場合) であれば、表現は「正規化されている」と言います。浮動小数点数 1.00×10^{-1} は正規化されますが、 0.01×10^1 は正規化されません。 $\beta = 2$ 、 $p = 3$ 、 $e_{\min} = -1$ 、 $e_{\max} = 2$ の場合、179 ページの図 D-1 に示すとおり、計 16 個の正規化浮動小数点数が存在します。図の太線は有意桁が 1.00 となる値を表わします。浮動小数点表現を正規化すると、一意の表現が得られます。しかし、この制限によってゼロを表現することができなくなります。ゼロを表現する一般的な方法として、 $1.0 \times \beta^{e_{\min}-1}$ を使用します。これにより、非負の実数を表わす番号が浮動小数点表現の辞書編成順に対応していると

1. この用語は、Forsythe/Moler が 1967 年、それまでの仮数に置き換わる表現として採用したものです。

いう事実が保持されます¹。指数を k ビットフィールドに格納すると、1 ビット分は 0 を表わすために予約する必要があるので、 $2^k - 1$ の値だけが指数用として確保されることになります。

なお、浮動小数点数の $\times$ は表記の一部として使用されるもので、浮動小数点の乗算記号とは異なります。 $\times$ 記号の意味は、文脈から簡単判断することができます。たとえば、 $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$ という式では、浮動小数点の乗算を 1 回だけ行うという意味になります。

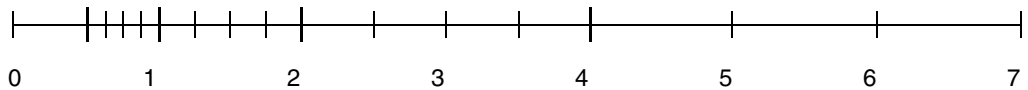


図 D-1 正規化数 $\beta = 2$ 、 $p = 3$ 、 $e_{\min} = -1$ 、 $e_{\max} = 2$

相対誤差と ulp

浮動小数点演算では本来、丸め誤差を避けることができないので、この誤差を測定する方法を確立しておくことが重要です。仮に $\beta = 10$ 、 $p = 3$ という浮動小数点フォーマットの例を考えてみます (この付録では、同じ例を使用します)。浮動小数点演算の結果が 3.12×10^{-2} の場合で、解を無限の精度に対して計算すると 0.0314 になり、最終桁に 2 ユニットの誤差があることがわかります。同じように、実数 0.0314159 を 3.14×10^{-2} として表わすと、最終桁の誤差は 0.159 となります。一般に浮動小数点数 $d.d\dots d \times \beta^e$ を使用して z を表わした場合、最終桁の誤差は $|d.d\dots d - (z / \beta^e)| \beta^{p-1}$ ユニットのようになります^{2 3}。ulp という用語は、“units in the last place” (最終桁のユニット) を略したものです。計算の結果が、正確な値にメモリーの近い浮動小数点値となる場合、0.5 ulp も誤差があることになります。浮動小数点値と実数の値の誤差を表わす場合に、**相対誤差**の近似値を求めることもできます。相対誤差とは、浮動小数点数と実数の差を実数値で割った結果のことです。たとえば、3.14159 を 3.14×10^0 によって近似表現した場合の相対誤差は $0.00159 / 3.14159 = 0.0005$ となります。

0.5 ulp に相当する相対誤差を計算する場合、実際の値にもっとも近い値を $d.dd\dots dd \times \beta^e$ で表わすと、 $0.00\dots 00 \beta' \times \beta^e$ もの誤差が出ることがあります (ここで β' は $\beta / 2$ の桁を表わし、浮動小数点数の有意桁の誤差が p ユニットの、また誤差の有意桁 p

1. ここでは、指数は有意桁の上位に格納することを前提とします。
2. z の値が $\beta^{e_{\max}+1}$ を超えない場合、あるいは $\beta^{e_{\min}}$ に満たない場合を除きます。この範囲を超える数については、ここでは扱いません。
3. z' は、 z に近似する浮動小数点数とします。 $|d.d\dots d - (z / \beta^e)| \beta^{p-1}$ は $|z' - z| / \text{ulp}(z)$ に等しくなります。誤差を測定するさらに正確な公式は $|z' - z| / \text{ulp}(z)$ です。--Ed

ユニットはゼロになります)。この誤差は $((\beta/2)\beta^{-p}) \times \beta^e$ です。 $d.dd\dots dd \times \beta^e$ の形式の数の絶対誤差はすべて同じになる一方、値はすべて β^e から $\beta \times \beta^e$ の範囲にあるので、相対誤差は $((\beta/2)\beta^{-p}) \times \beta^e / \beta^e$ と $((\beta/2)\beta^{-p}) \times \beta^e / \beta^{e+1}$ の範囲に存在します。

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2}\text{ulp} \leq \frac{\beta}{2}\beta^{-p} \quad (2)$$

特に、 0.5 ulp に相当する相対誤差は、 β を係数として可変となります。この係数のことを「ワブル」と言います。 $\epsilon = (\beta/2)\beta^{-p}$ を上式 (2) の上限に設定した場合、実数値がもっとも近い浮動小数点数に丸められると、相対誤差は必ず ϵ によって境界が設定されることになります。このことを「マシン・イプシロン」と言います。

上の例で、相対誤差は $0.00159/3.14159 = 0.0005$ となります。このような小さい数を避けるために、相対誤差は通常、係数に ϵ を乗じた数として表わします (この場合、 $\epsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = 0.005$ となります)。したがって、相対誤差は $(0.00159/3.14159)/0.005 = 0.1 \epsilon$ と表現されます。

ulp と相対誤差の違いを説明するために、実値 $x = 12.35$ を例に取ります。この値は $\tilde{x} = 1.24 \times 10^1$ により近似値として表わすことができます。誤差は 0.5 ulp で、相対誤差は 0.8ϵ です。次に $8\tilde{x}$ を計算します。正確な値は $8x = 98.8$ で、計算値は $8\tilde{x} = 9.92 \times 10^1$ です。誤差は 4.0 ulp ですが、相対誤差は依然として 0.8ϵ です。すなわち相対誤差は変わらない一方、**ulp** 誤差の測定値は 8 倍大きくなっています。一般に、基数が β の場合、**ulp** による相対誤差は最大 β の係数分だけ変動する可能性があります。逆に、上式 (2) に示すとおり、 0.5 ulp という固定誤差により β 分だけ変動する相対誤差が生じることになります。

丸め誤差を表わすもっとも自然な方法は **ulp** を使用する形式です。たとえば、もっとも近い浮動小数点数に丸めるという操作は、誤差が 0.5 ulp 以下になるということです。ただし、各種の公式が原因で起きる丸め誤差を分析する場合は、相対誤差の方が正確です。これについては 225 ページの「証明」で詳しく分析します。 ϵ は、もっとも近い浮動小数点値より、ワブル係数 β 分だけ多くなる可能性があるので、公式による誤差の計算は、 β の小さいマシンではそれだけ緊密になります。

丸め誤差の大きさだけが問題の場合、**ulp** と ϵ の差は多くても β 以内に抑えられるので、入れ換えて使用してもかまいません。たとえば、浮動小数点数に $n \text{ ulp}$ だけ誤差がある場合、汚染桁の数が $\log \beta n$ となるという意味です。計算上の相対誤差が $n \epsilon$ の場合は、次のように表わされます。

$$\text{contaminated digits} \approx \log_{\beta} n \quad (3)$$

保護桁

2つの浮動小数点数の差を計算する1つの方法として、差そのものの値を正確に求めて、これをもっとも近い値に丸めることができます。この方法は、オペランドのサイズが大きくなると、その分、コストがかかるようになります。 $p=3$ とすると $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ は次のように計算できます。

$$x = 2.15 \times 10^{12} \quad y = 0.0000000000000000125 \times 10^{12} \quad x - y = 2.149999999999999875 \times 10^{12}$$

これにより 2.15×10^{12} に丸められます。浮動小数点对応のハードウェアは通常、これらの桁をすべて占有するのではなく、固定された桁数を対象として演算を行います。確保された桁数が p の場合、これより小さいオペランドが右にシフトされると、各桁は単に破棄されます(丸められません)。 $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ は次のようになります。

$$x = 2.15 \times 10^{12} \quad y = 0.00 \times 10^{12} \quad x - y = 2.15 \times 10^{12}$$

解は、差を正確に計算した場合とまったく同じになり、この結果が丸められます。別の例を考えてみます。 $10.1 - 9.93$ は次のようになります。

$$x = 1.01 \times 10^1 \quad y = 0.99 \times 10^1 \quad x - y = 0.02 \times 10^1$$

正確な解は 0.17 なので、計算上の差は 30 ulp となり、全部の桁に誤差が生じています。なぜ、これほど大きな誤差が出るのでしょうか。

定理 1

パラメータ b と p を使用した浮動小数点フォーマットで p 桁の差を計算した場合、結果の相対誤差は最大、 $b-1$ になる。

証明

式 $x - y$ で $\beta - 1$ の相対誤差は、 $x = 1.00...0$ 、 $y = .\rho \rho \dots \rho$ ($\rho = \beta - 1$) の場合に生じます。ここで y の桁数は p です(すべて ρ)。正確な差は $x - y = \beta^{-p}$ です。ただし p 桁だけで解答を求めると、 y の最下位桁がシフトオフするので、計算上の差は β^{-p+1} です。したがって、誤差は $\beta^{-p} - \beta^{-p+1} = \beta^{-p}(\beta - 1)$ で、相対誤差は $\beta^{-p}(\beta - 1) / \beta^{-p} = \beta - 1$ です。■

$\beta = 2$ の場合、相対誤差は結果と同じくらい大きくなる場合があります。また $\beta = 10$ の場合、9 倍になる場合があります。言い換えると、 $\beta = 2$ の場合、上記の式 (3) は汚染桁の数が $\log_2(1/\varepsilon) = \log_2(2^p) = p$ になることを示します。すなわち結果の p 桁はすべて誤っていることになります。この状況を解決するために、**保護桁**として 1 桁分を追加してみます。すなわち小さい数字を $p + 1$ 桁に切り捨てて、減算の結果を p 桁に丸めます。この保護桁を使用すると、前の例は次のようになります。

$$x = 1.010 \times 10^1 \quad y = 0.993 \times 10^1 \quad x - y = 0.017 \times 10^1$$

また解も正しくなります。110 - 8.59 の場合、1 桁の保護桁を設けると、結果の相対誤差は、 ε より大きくなる場合があります。

$$x = 1.10 \times 10^2 \quad y = 0.085 \times 10^2 \quad x - y = 1.015 \times 10^2$$

これを丸めると 102 になります。一方、正しい解は 101.41 で、相対誤差は 0.006 となり、 $\varepsilon = 0.005$ より大きくなります。一般には、結果の相対誤差は ε よりわずかに大きくなります。正確に表わすと次のようになります。

定理 2

x と y が β と p のパラメータを使用したフォーマットになっている場合に、 $p + 1$ 桁 (1 保護桁) で減算を行うと、結果の相対誤差は 2ε より少なくなる。

この定理については 224 ページの「丸め誤差」で証明します。なお、 x と y は正負のいずれであってもかまわないので、上記の定理では加算を使用しています。

相殺

この最後の項の結論は、保護桁がなければ、2 つの近い値で減算をした場合に生じる相対誤差は、きわめて大きくなる可能性がある、ということです。すなわち減算 (負符号を使用した加算) を伴う式を評価した結果、相対誤差が大きくなりすぎて、桁がすべて無意味になる可能性があります (定理1)。互いに近い値を減算すると、オペランドの最上位桁がそれぞれ一致して、相互に相殺されます。この相殺には悪性と良性の 2 種類があります。

悪性の相殺は、オペランドが丸め誤差の影響を受ける場合に生じます。たとえば、二次方程式の根の公式の中にある $b^2 - 4ac$ の場合です。 b^2 と $4ac$ は浮動小数点の乗算の結果となるので、いずれも丸め誤差の影響を受けます。仮にこれらの値が、もっとも近い浮動小数点数に丸められ、精度が 0.5 ulp 以内になっているとします。減算を行うときの相殺によって正確な桁が多数消失し、丸め誤差によって汚染された桁だけが

残されることがあります。したがって、差の中に **ulp** の大きい誤差が含まれる可能性があります。たとえば、 $b = 3.34$ 、 $a = 1.22$ 、 $c = 2.28$ であると仮定します。 $b^2 - 4ac$ の正確な値は、0.0292 です。しかし、 b^2 は 11.2 に、また $4ac$ は 11.1 にそれぞれ丸められるので、最終的な解は 0.1 となり、仮に $11.2 - 11.1$ が 0.1 であったとしても、70 **ulp** もの誤差が出ます¹。減算自体が誤差の原因になったのではなく、以前の乗算で導入された誤差が明らかになったということです。

一方、**良性の相殺**は、既知の数量を減算するときにあります。 x と y に丸め誤差がない場合、保護桁を使用して減算を行うと、定理 2 により $x - y$ の差の相対誤差は非常に小さく (2ε 以下) になります。

悪性の相殺の原因となる公式は編成を変更することにより、問題を解決できる場合があります。次の二次方程式の根の公式の例を見えます。

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad (4)$$

b^2 が $b^2 \gg ac$ の場合、 $b^2 - 4ac$ には相殺が行われないので、 $\sqrt{b^2 - 4ac} \approx |b|$ となります。ただし、2 つの公式のうちのもう一方の加算 (減算) が原因となって悪性の相殺が行われます。この問題を避けるには、次のように r_1 の分子と分母を乗算して (r_2 の場合も同様に)、

$$-b - \sqrt{b^2 - 4ac}$$

次の式を求めます。

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \quad (5)$$

1. 70 ではなく 700 です。0.1 - 0.0292 = 0.0708 なので、ulp (0.0292) の誤差は 708 ulp となります。-Ed

$b^2 \gg ac$ で、 $b > 0$ の場合、公式 (4) を使用して r_1 を計算すると、相殺が行われます。したがって、 r_1 の計算には公式 (5)、また r_2 の計算には公式 (4) を使用するようしてください。一方、 $b < 0$ の場合は、 r_1 の計算に公式 (4)、また r_2 の計算に公式 (5) を使用してください。

$x^2 - y^2$ の式も悪性の相殺を伴う公式です。 $(x - y)(x + y)$ として評価の方が正確です¹。二次方程式の根の公式とは異なり、この式の方にも減算を伴いますが、丸め誤差が介在しない良性の相殺なので悪性ではありません。定理 2 により、 $x - y$ の相対誤差は多くても 2ε に抑えられます。これは、 $x + y$ にもあてはまります。相対誤差の少ない数量を乗算しても、積の相対誤差は小さくなります (224 ページの「丸め誤差」を参照)。

正確な値と計算値の混乱を避けるために、次のような表記を使用します。まず $x - y$ は、 x と y の正確な差を表わすものとします。また x から y を減算した計算値を表わす場合は、 $x \ominus y$ と表記します。同様に、 $\oplus$ 、 $\otimes$ 、 $\oslash$ は順に加算、乗算、除算の各計算値を表わします。すべて大文字の場合、特定の関数の計算値 (たとえば $\text{LN}(x)$ 、 $\text{SQRT}(x)$ など) を表わします。小文字と一般的な数学記号は、値そのもの (たとえば $\ln(x)$ 、 $\sqrt{x}$ など) を表わすものとします。

$x^2 - y^2$ の近似値を表わす場合、 $(x \ominus y) \otimes (x \oplus y)$ という表記はきわめて効果的ですが、 x と y の浮動小数点数自体が $\tilde{x}$ と $\hat{y}$ の真の値の近似値になることがあります。たとえば、 $\tilde{x}$ と $\hat{y}$ が、2 進表記では正確には表わせない 10 進数である場合があります。この場合、 $x \ominus y$ が $x - y$ の近似値を表わすにしても、真の式 $\tilde{x} - \hat{y}$ に比べ、相対誤差は大きくなるので、 $x^2 - y^2$ に対する $(x + y)(x - y)$ の利点は、それほど大きくはなくなります。 $(x + y)(x - y)$ と $x^2 - y^2$ の計算負荷はほとんど同じなので、この場合に限り $(x + y)(x - y)$ の方が有利ということになります。ただし一般には、入力近似値ではないことが多いので、悪性の相殺を良性の相殺に置き換える意味はありません。とは言え、データが正しくない場合でも、相殺を全面的に排除 (二次方程式の根の公式のように) できれば、意味があります。この付録では、あるアルゴリズムに対する浮動小数点の入力が正しく、結果も可能な限り正確に計算されることを前提とします。

$x^2 - y^2$ の式は、 $(x - y)(x + y)$ として書き換えると、悪性の相殺が良性の相殺に置き換えられるので、より正確になります。

次に、悪性の相殺を良性の相殺に置き換えることのできる公式の例を紹介します。

三角形の面積は、次のように 3 辺 a 、 b 、 c の長さで直接表わすことができます。

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = (a+b+c)/2 \quad (6)$$

1. $(x - y)(x + y)$ の式では悪性の相殺は起きないものの $x \gg y$ または $x \ll y$ の場合は $x^2 - y^2$ よりもわずかに精度が落ちます。この場合、 $x^2 - y^2$ の小さい方を計算するときに起きる丸め誤差が、最終的な減算に影響を与えることはないので、 $(x - y)(x + y)$ の丸め誤差は 3、また $x^2 - y^2$ の丸め誤差は 2 になります。

三角形が非常に平たい形状である ($a \approx b + c$) と仮定します。 $s \approx a$ となるので、式 (6) の項 ($s - a$) では、2 つの近い値 (一方に丸め誤差が含まれる可能性があります) の減算を行います。たとえば $a = 9.0$ 、 $b = c = 4.53$ の場合、正しい値は 9.03 で A は 2.342... となります。計算値 s (9.05) に 2 ulp の誤差がありますが、 A の計算値は 3.04 なので誤差は 70 ulp となります。

平坦な三角形の場合でも、正確な結果が得られるように、公式 (6) を次のように書き換えることができます (Kahan, 1986)。

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, a \geq b \geq c \quad (7)$$

a 、 b 、 c は $a \geq b \geq c$ の条件を満たさない場合は、それぞれの名前を変更してから、公式 (7) を使用します。(6) と (7) の右边が代数的に同一であることは比較的簡単にチェックできます。 a 、 b 、 c の値により、面積 2.35 が求められます。この誤差は 1 ulp で、最初の公式よりはるかに正確であることがわかります。

この例の場合、公式 (7) は公式 (6) よりはるかに正確であるので、一般に公式 (7) が有効であると言えます。

定理 3

公式 (7) を使用して三角形の面積を求めたときに起きる丸め誤差は、保護桁で減算を行い、 $e \leq 0.005$ で、平方根が $1/2$ ulp 以下の範囲で計算される限り、 11ε 以内に抑えられる。

$e < 0.005$ という条件は、事実上あらゆる浮動小数点システムで満たされます。たとえば、 $\beta = 2$ の場合、 $p \geq 8$ であれば、必ず $e < 0.005$ になり、 $\beta = 10$ であれば $p \geq 3$ で十分ということになります。

式の相対誤差に関する定理 3 の記述では、式が浮動小数点演算によって計算されることがわかります。実際に、相対誤差は式に関するものです。

$$\text{SQRT}((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4 \quad (8)$$

公式 (8) の構造は複雑に入り組んでいるので、定理の記述では、 E を丸付きの表記で表わすのではなく、 E の計算値として扱います。

誤差の範囲は悲観的すぎる場合があります。上記のような数値の例の場合、公式 (7) の計算値は 2.35 で、正しい値である 2.34216 と比較すると、相対誤差は 0.7ε となります。これは、 11ε よりはるかに小さい値です。誤差の範囲を計算する主な目的は、正確な境界を設定することではなく、公式に数値の問題が存在しないことを確認することにあります。

良性の相殺に変更できる最後の例は $(1+x)^n$ (ここで $x \ll 1$) です。この式は金融計算でよく使用されます。たとえば、年率 6% の銀行口座に毎日 100 ドルずつ預金した場合の複利計算の例を考えてみます。 $n = 365$ で、 $i = 0.06$ とすると、1 年後の貯蓄額は

$$100 \frac{(1 + i/n)^n - 1}{i/n} \text{ ドルとなります。}$$

$\beta = 2$ 、 $p = 24$ として計算すると結果は \$37615.45 となり、実際の金額の \$37614.05 とは \$1.40 の差が出ます。この問題の理由は簡単です。式 $1 + i/n$ では、0.0001643836 に 1 を加算するので、 i/n の下位ビットが失われるということです。この丸め誤差は、 $1 + i/n$ に n 乗すると、増幅します。

$(1 + i/n)^n$ という複雑な式は $e^{n \ln(1 + i/n)}$ と書き換えることができます。この場合の問題は、 x が小さい場合の $\ln(1 + x)$ の計算です。1 つの方法として $\ln(1 + x) \approx x$ の近似値を使用することができます。この場合、支払いは \$37617.26 となり、誤差は \$3.21 なので、単純な公式よりも精度が落ちます。ただし、定理 4 に示すとおり、 $(1 + x)$ をさらに正確に計算する方法があります (HP, 1982年)。この公式により、\$37614.07 という結果が得られ、誤差がわずか 2 セント以内になります。

定理 4 では、 $\text{LN}(x)$ により $\ln(x)$ の近似値が $1/2 \text{ ulp}$ 以内の誤差で求められることを前提とします。これにより、 x の値が小さいときに、 $1 \oplus x$ が、 x の下位ビットにある情報を喪失するために、 $\text{LN}(1 \oplus x)$ により $\ln(1 + x)$ の近似値が得られない問題が解決されます。すなわち、 $x \ll 1$ のときに、 $\ln(1 + x)$ の計算値は実際の値の近似値にはならないという問題です。

定理 4

$\ln(1+x)$ を次の公式で計算した場合、

$$\ln(1+x) = \begin{cases} x & \text{for } 1 \oplus x = 1 \\ \frac{x \ln(1+x)}{(1+x) - 1} & \text{for } 1 \oplus x \neq 1 \end{cases}$$

保護桁を使用して減算を行い、 $e < 0.1$ 、または $\ln$ を $1/2 \text{ ulp}$ 以内の誤差で計算すると、 $0 \leq x < \frac{3}{4}$ の場合に相対誤差は $5e$ 以内となる。

この公式は、 x がどのような値であっても適用されますが、 $x \ll 1$ の場合、すなわち本来の公式 $\ln(1+x)$ で悪性の相殺が起きる状況に限り意味があります。公式は一見、意味不明のように思われますが、これが正しく動作する理由は簡単です。 $\ln(1+x)$ を

$$x \left(\frac{\ln(1+x)}{x} \right) = x\mu(x) \quad \text{と書き換えてみます。左辺の要素はそのまま計算できます}$$

が、右辺の要素 $\mu(x) = \ln(1+x)/x$ は、 x に 1 を加算するとき大きな丸め誤差が生じます。しかし、 $\ln(1+x) \approx x$ のため μ はほとんど定数です。したがって、 x をわずかに変更するだけでも、大きな誤差を伴います。

すなわち、 $\tilde{x} \approx x$ の場合、 $x\mu(\tilde{x})$ を計算すると、 $x\mu(x) = \ln(1+x)$ のほぼ近似値が得られます。 $\tilde{x}$ と $\tilde{x} + 1$ を正しく計算できる $\tilde{x}$ の値があるでしょうか。 $1 + \tilde{x}$ はちょうど $1 \oplus x$ になるので、 $\tilde{x} = (1 \oplus x) \ominus 1$ という値です。

この項の結論として、既知の近い値を減算するとき (良性の相殺を伴う) は、保護桁により精度が保証されるということが言えます。不正確な結果をもたらす公式は、良性の相殺を使用して、数値の精度を上げることにより、書き換えることができます。ただし、この方法は保護桁を使用して減算を行う場合に限り利用できます。また保護桁を使用しても、加算器のビットが 1 つ多くなるだけなので、それほどの犠牲も伴いません。たとえば 54 ビットの倍精度加算器の場合、余分なコストは 2 % 程度で済みます。この程度のコストによって、三角形の面積を求める公式 (6) や式 $\ln(1+x)$ などの多数のアルゴリズムを実行する利点が得られます。最近のコンピュータには通常、保護桁が用意されているので、これに対応できないシステムは、ほとんどありません (Cray システムなどを除きます)。

正確な丸め操作

保護桁を使用して浮動小数点演算を行うと、正確に計算した後でもっとも近い浮動小数点数に丸めた場合よりも、精度が落ちます。後者の計算方法を「**正確な丸め操作**」と言います¹。定理 2 の前に紹介した例は、1 桁分の保護桁を確保したからといって、必ずしも正確な結果が得られるとは限らないことを示しています。前の項には、正しい動作を保証するために、保護桁を使用するアルゴリズムの例をいくつか紹介しました。ここでは、正確な丸めを行うためのアルゴリズムの例を示します。

ここまでの段階では、まだ丸めの定義を示していません。丸めの操作は、切り下げと切り上げの区切りをどこで付けるかという点 (たとえば、12.5 を 12 とするか 13 とするかという問題) を除き、難しくありません。1 つの方法として、10 までの値を四捨五入して、0、1、2、3、4 を切り捨て、5、6、7、8、9 を切り上げるという考え方があ

1. 「正確な丸め操作」は英文で言う “exactly rounded operation” または “correctly rounded operation” のことです。

ります。したがって、12.5は13として扱います。DECのVAXではこのような四捨五入が採用されています。また、5で終わる数は2つの丸め方法のちょうど境目にあるという意味で、丸めを行う回数全体の半分を切り上げ、半分を切り下げるという考え方もあります。この50%ずつ2分するという方法には、丸めの結果が少なくても、最下位の桁が偶数になっていなければならないという条件を伴います。したがって、12.5は、2が偶数であるため、13ではなく、12に切り捨てられます。切り上げと偶数への切り捨てるのどちらがよいかという問題について、ReiserとKnuthは偶数への切り捨ての方が望ましいと指摘しています(1975)。

定理 5

x と y を浮動小数点数として、 $x_0 = x$ 、 $x_1 = (x_0 \ominus y) \oplus y$ 、...、 $x_n = (x_{n-1} \ominus y) \oplus y$ と定義する。 $\oplus$ と $\ominus$ を偶数への切り捨てにより同じように丸めると、すべての n について $x_n = x$ 、また $n \geq 1$ のすべてについて $x_n = x_1$ となる。■

この結果を分類する上で、 $x = 1.00$ 、 $y = -0.555$ として $\beta = 10$ 、 $p = 3$ の場合を考えてみます。これを切り上げる場合、 $x_0 \ominus y = 1.56$ 、 $x_1 = 1.56 \ominus 0.555 = 1.01$ 、 $x_1 \ominus y = 1.01 \oplus 0.555 = 1.57$ というシーケンスになり、 $x_n = 9.45$ ($n \leq 845$) になる¹まで、 x_n の連続値が 0.01 ごとに増えていきます。偶数に切り捨てる場合、 x_n はつねに 1.00 になります。この例は、切り上げのルールを使用すると、計算結果がしだいに上方にずれていき、一方、偶数への切り捨てを使用すると、このずれが生じないことを示しています。この後の説明では、偶数への切り捨てを一貫して使用することにします。

多重精度の演算では、正確な丸めが1回だけ行われます。精度を上げるには、2つの基本的なアプローチがあります。1つは、きわめて大きい有意桁を使用した浮動小数点数です。この有意桁はワード配列に格納され、アセンブリ言語でこれらの数を操作するためのルーチンのコーディングが行われます。もう1つのアプローチは、通常の浮動小数点数の配列としてさらに高い精度の浮動小数点数を表わす方法です。この場合、無限の精度で配列要素を追加することにより、高い精度の浮動小数点数を復元します。ここでは、2つ目のアプローチについて説明します。浮動小数点数の配列を使用する利点は、高水準言語により移植可能な形でコーディングできることにあります。この反面、正確な丸めの演算が必要になります。

このシステムにおける乗算の重要な点は、各加数が xy と同じ精度を持つように、 xy の積を和として表わすことにあります。これは、 x と y を分割することによって行われます。 $x = x_h + x_l$ 、 $y = y_h + y_l$ と書くと、実際の積は $xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l$ となります。 x と y に p ビットの有意桁がある場合 $[p/2]$ ビットで x_l 、 x_h 、 y_h 、 y_l を表わすことができれば、加数の有意桁も p ビットになります。 p が偶数であれば、簡

1. $n = 845$ の場合 m 、 $x_n = 9.45$ 、 $x_n + 0.555 = 10.0$ 、 $10.0 - 0.555 = 9.45$ となります。したがって、 $n > 845$ については $x_n = x_{845}$ となります。

単に分割できます。 $x_0.x_1...x_{p-1}$ の数は、 $x_0.x_1...x_{p/2-1}$ と $0.0...0x_{p/2}...x_{p-1}$ の和として表わすことができます。一方、 p が奇数の場合、このように簡単には分割できません。ただし、負数を使用して、1ビット分を追加できます。たとえば、 $\beta = 2$ 、 $p = 5$ 、 $x = 0.10111$ の場合、 x は $x_h = 0.11$ 、および $x_l = -0.00001$ として分割できます。Dekker (1971)によると、分割方法は簡単ですが、1桁の保護桁以外にも余分に桁が必要です。

定理 6

$\beta > 2$ であっても p が偶数であるという制限のもとに、 p を浮動小数点精度として、浮動小数点数を正確に丸めるものとする。このとき、 $k = \lfloor p/2 \rfloor$ が精度の半分(切り上げ)で、 $m = \beta^k + 1$ とすると、 x は $x = x_h + x_l$ (ただし $x_h = (m \otimes x) \ominus (m \otimes x \ominus x)$ 、 $x_l = x \ominus x_h$ で、各 x_i は $\lfloor p/2 \rfloor$ ビット精度で表現可能とする)として分割できる。

この定理が実際の例にどのように適用されるかを見るために、 $\beta = 10$ 、 $p = 4$ 、 $b = 3.476$ 、 $a = 3.463$ 、 $c = 3.479$ であると仮定します。 $b^2 - ac$ をもっとも近い浮動小数点数に丸めると、0.03480になり、 $b \otimes b = 12.08$ 、 $a \otimes c = 12.05$ になります。したがって、 $b^2 - ac$ の計算値は0.03となります。これには480 ulpの誤差があります。定理6にもとづいて $b = 3.5 - 0.024$ 、 $a = 3.5 - 0.037$ 、 $c = 3.5 - 0.021$ とすると、 b^2 は $3.5^2 - 2 \times 3.5 \times 0.024 + 0.024^2$ となります。各加数はすべて正確なので、 $b^2 = 12.25 - 0.168 + 0.000576$ になります(この時点で和の評価はされていません)。同じように、 $ac = 3.52 - (3.5 \times 0.037 + 3.5 \times 0.021) + 0.037 \times 0.021 = 12.25 - 0.2030 + 0.000777$ になります。

最後に、2つの連続項ごとに減算すると、 $b^2 - ac$ の近似値が $0 \oplus 0.0350 \ominus 0.000201 = 0.03480$ として求められます。これは、正確に丸めた場合の結果と一致します。定理6で正確な丸めが必要になることを確認するために、 $p = 3$ 、 $\beta = 2$ 、 $x = 7$ 、さらに $m = 5$ 、 $mx = 35$ 、 $m \otimes x = 32$ の例を考えてみます。1桁の保護桁を使用して減算を行うと、 $(m \otimes x) \ominus x = 28$ となります。したがって、 $x_h = 4$ 、 $x_l = 3$ となるので、 x_l は $\lfloor p/2 \rfloor = 1$ ビットでは表わせません。

正確な丸めの最後の例として、 m を10で除算する場合を考えてみます。この結果、一般には $m/10$ に一致しない浮動小数点数が得られます。 $\beta = 2$ の場合、正確な丸めを使用して、 $m/10$ と10を掛けると、不思議なことに m がリストアされます。実際には、さらに一般的な事実があてはまります(Kahan)。証明は詳細にわたるものですが、このような詳しい説明まで関心のない方は、191ページの「IEEE標準」まで読み飛ばしてもかまいません。

定理 7

$\beta = 2$ のとき、 m と n が $|m| < 2^{p-1}$ の整数で、 n に $n = 2^i + 2^j$ という特殊な形式がある場合、浮動小数点演算を正確に丸めるとすれば、 $(m \oslash n) \otimes n = m$ となる。

証明

2 のべき乗でスケーリングするのは、有意桁ではなく、単に指数が変更されるだけなので、悪影響はありません。 $q = m/n$ の場合、 $2^{p-1} \leq n < 2^p$ となるように n をスケーリングし、

$\frac{1}{2} < q < 1$ となるように m をスケーリングします。したがって $2^{p-2} < m < 2^p$ となります。 m には p 個の有意ビットがあるので、2 進小数点の右に最大 1 ビットが必要です。 m の符号を変更しても悪影響はないので、 $q > 0$ とします。

$\bar{q} = m \oslash n$ の場合、定理を証明するには、次を示す必要があります。

$$|n\bar{q} - m| \leq \frac{1}{4} \quad (9)$$

これは、 m の 2 進小数点の右側に最大 1 ビットが使用されているので、 $n\bar{q}$ により m に丸められます。

中間の場合の扱いについては、 $|n\bar{q} - m| = \frac{1}{4}$ の場合に、スケーリングしていない初期の m は $|m| < 2^{p-1}$ 、また下位ビットは 0 になるので、スケーリング後の m の下位ビットも 0 になります。したがって、中間の値は m に丸められます。 $q = .q_1q_2 \dots \hat{q} \dots q_p1$ であると仮定します。 $|n\bar{q} - m|$ の近似値を求めるにはまず、 $|\hat{q} - q| = |N/2^{p+1} - m/n|$ (N は奇数の整数) を計算します。 $n = 2^i + 2^j$ 、および $2^{p-1} \leq n < 2^p$ であるから、特定の $k \leq p-2$ については $n = 2^{p-1} + 2^k$ でなければなりません。したがって、次のようになります。

$$|\hat{q} - q| = \left| \frac{nN - 2^{p+1}m}{n2^{p+1}} \right| = \left| \frac{(2^{p-1-k} + 1)N - 2^{p+1-k}m}{n2^{p+1-k}} \right|$$

分子は整数です。 N は奇数なので、実際には奇数の整数となります。したがって、 $|\hat{q} - q| \geq 1/(n2^{p+1-k})$ となります。 $q < \hat{q}$ ($q > \hat{q}$ の場合も同様) と仮定¹すると、 $n\bar{q} < m$ で、次のようになります。

1. 進数の場合、 q は $\hat{q}$ に等しくありません。-Ed

$$|m - n\bar{q}| = m - n\bar{q} = n(q - \bar{q}) = n(q - (\hat{q} - 2^{-p-1})) \leq n \left(2^{-p-1} - \frac{1}{n2^{p+1-k}} \right) \\ = (2^{p-1} + 2k)2^{-p-1} + 2^{-p-1+k} = \frac{1}{4}$$

これにより、(9) が成り立ち、定理が証明されることになります¹。■

定理は、 $2^i + 2^j$ を $\beta^i + \beta^j$ に置き換える限り、任意の基数 β について真になります。ただし β が大きくなるに従って $\beta^i + \beta^j$ の分母の差が大きくなっていきます。

ここで、本付録の最初に提示した「基本的な演算処理で多少の丸め誤差が出ることに何か問題があるか？」という根本的な疑問に立ち返ることにします。この答えは「大いに問題はある」です。その理由は、正確な基本演算では相対誤差が低く抑えられるという意味で、公式の「妥当性」を証明できるからです。

各公式にわずかな相対誤差を伴うという意味で、公式が正しいことを証明できるという理由によって、丸め誤差の問題はあるということになります。182 ページの「相殺」では、保護桁を使用するアルゴリズムによっては、上記の意味で正しい結果が得られるものと説明しました。ただし、これらの公式への入力、不正確な測定にもとづく数値である場合、定理 3 と 4 で言う境界は無意味になってきます。この理由は、 x と y が、ある測定値の近似値である場合、良性の相殺 $x - y$ が悪性になるからです。しかし、正確な計算は、定理 6 と 7 で示した正確な関係を実証できるという意味で、データが不正確な場合でも有効であることに違いはありません。各浮動小数点変数がそれぞれ、実際の値の近似値にすぎない場合でも有効であるということです。

IEEE 標準

浮動小数点演算には、IEEE 754 と IEEE 854 という 2 つの標準があります。IEEE 754 は、単精度の場合 $\beta = 2$ 、 $p = 24$ 、また倍精度の場合 $p = 53$ を使用する 2 進数標準です (IEEE 1987)。この標準により、単精度と倍精度の場合の正確なビット・レイアウトが指定されます。IEEE 854 では、 $\beta = 2$ 、または $\beta = 10$ のいずれかを使用します。IEEE 754 と異なり、IEEE 854 の場合は浮動小数点数をビットに符号化する形式は指定されません (Codyほか 1984)。 p に対する特定の値は必要ありませんが、単精度と倍精度に認められる p の値が制限されます。なお IEEE 標準という用語は、以上 2 つの標準に共通する特性を表わす場合に使用します。

1. 2 以外の基数についても読者自身で確認してみてください。-Ed

本項では、IEEE 標準の概要について説明します。以降の各項では、この標準の各特性と、これが採用された理由を示します。なお IEEE 標準がもっとも優れた浮動小数点標準であるかどうかを分析するのは本資料の主旨ではないので、あくまで標準としてそのまま受け入れ、その基本概念を説明することにします。詳細については、IEEE 標準自体の資料を参照してください (IEEE 1987、Cody ほか、1984)。

フォーマットと操作

基数

IEEE 854 で $\beta = 10$ が使用されている理由は明らかです。基数 10 は、日常的に親しみやすい数であるからです。 $\beta = 10$ は、特に計算結果を 10 進数で表わすような電卓などに適しています。

IEEE 854 で、基数が 10 以外の場合に必ず 2 を使用する理由はいくつかあります。179 ページの「相対誤差と ulp」で、理由の 1 つを説明しました。 β を 2 にすると、相対誤差として計算した場合、 0.5 ulp の丸め誤差が β の倍数で変動するので、誤差の分析結果がはるかに厳しくなり、相対誤差にもとづく誤差はほとんどの場合、単純に分析できるという理由です。また、基数を大きくした場合の有効な精度に関する理由も考えられます。 $\beta = 16$ 、 $p = 1$ と、 $\beta = 2$ 、 $p = 4$ の場合を比較してみます。いずれのシステムも有意桁は 4 ビットとします。たとえば、 $15/8$ を計算します。 $\beta = 2$ の場合、15 は 1.111×2^3 と、また $15/8$ は 1.111×2^0 と表現されます。したがって、 $15/8$ はそのまま正確に表わされます。一方、 $\beta = 16$ の場合、15 は $F \times 16^0$ (F は 15 に対応する 16 進表現) と表わします。しかし、 $15/8$ は 1×16^0 となり、正しいビットは 1 ビットしかありません。一般に、基数 16 を使用すると最大 3 ビットを失うことがあるので、 p の 16 進桁の精度は、有効精度が 2 進の $4p$ 分ではなく、 $4p - 3$ にまで低下することがあります。 β の値が大きくなるに従って、こうした問題が出てくるとすれば、IBM は System/370 に、なぜ $\beta = 16$ を採用したのでしょうか。その真意は IBM 以外に知るよしもありませんが、2 つの理由が考えられます。まず、指数の範囲が広がるという点です。System/370 の単精度では、 $\beta = 16$ と $p = 6$ が使用されています。したがって、有意桁には 24 ビットが必要になります。24 ビットは 32 ビットに収まるので、指数用の 7 ビットと符号用の 1 ビットが残ります。したがって、表現可能な数字の範囲は 16^{-2^6} から $16^{2^6} = 2^{2^8}$ になります。 $\beta = 2$ で同じ指数範囲を確保するには、指数分に 9 ビットが必要となり、有意桁にはわずか 22 ビットしか使用できません。一方、 $\beta = 16$ とすると、前述のとおり、有効精度が $4p - 3 = 21$ ビットにまで低下する

という問題があります。これも、 $\beta = 2$ で 1 ビット分、精度を上げる (本項で後述) ことができる場合は、さらに悪くなります。したがって、 $\beta = 2$ のマシンには、 $\beta = 16$ のマシンの 21 ~ 24 ビットに相当する 23 ビット分の精度が確保されます。

IBM が $\beta = 16$ を採用したもう 1 つの理由はシフトに関するものです。2 つの浮動小数点数を加算した場合に、各指数が異なっていると、基数の位置を合わせるために、いずれかの有意桁をシフトしなければならないことがあります。これが原因でパフォーマンスが低下することがあります。 $\beta = 16$ 、 $p = 1$ のシステムでは、1~15 までの数はすべて指数が同じになるので、 $\binom{15}{2} = 105$ のどの組み合わせを加算してもシフトは必要ありません。一方、 $\beta = 2$ 、 $p = 4$ のシステムでは、これらの数の指数の範囲は 0~3 なので、105 ペアのうちの 70 についてはシフトが必要になります。

最近のハードウェアでは、オペランドのサブセットに対するシフトを避けても、パフォーマンス上、それほど大きな利点は得られないので、ワブルの少ない $\beta = 2$ の方が有利な基数であると言えます。 $\beta = 2$ のもう 1 つの利点は、有意桁の余分なビットが得られる点です¹。浮動小数点数は必ず正規化されるので、有意桁の最上位ビットは必ず 1 になり、これに対応するビットを無駄にするのは意味がありません。この手法を効果的に利用したフォーマットのことを「隠れビット」と言います。177 ページの「浮動小数点フォーマット」では、0 に対する特殊な規約が必要であることを説明しました。そこで示した方法は、 $e_{\min} - 1$ の指数と、すべてゼロの有意桁により、 $1.0 \times 2^{e_{\min} - 1}$ ではなく、0 を表わすというものです。

IEEE 754 の単精度は、符号用の 1 ビット、指数用の 8 ビット、および有意桁用の 23 ビットを使用して、32 ビットをエンコードします。ただし、隠れビットを使用するので、23 ビットだけを使用してエンコードした場合でも、有意桁は 24 ビット ($p = 24$) になります。

精度

IEEE 標準では、単精度、倍精度、拡張単精度、および拡張倍精度という 4 種類のフォーマットを定義しています。IEEE754 の場合、単精度と倍精度はほぼ、大部分の浮動小数点ハードウェアで要求される機能に対応することができます。単精度では 32

1. これは、Knuth (1981, 211ページ) によると Konrad Zuse に考案されたとされていますが、Goldberg (1967) によって最初に発表されたようです。

ビットのシングルワード、また倍精度では 2 つ連続した 32 ビットワードが占有されます。拡張精度は、精度を多少上げ、指数の範囲を拡張したフォーマットです (表 D-1)。

表 D-1 IEEE 754フォーマットのパラメータ

パラメータ	フォーマット			
	単精度	拡張単精度	倍精度	拡張倍精度
p	24	≥ 32	53	≥ 64
$e_{\max}$	+127	≥ 1023	+1023	> 16383
$e_{\min}$	-126	≤ -1022	-1022	≤ -16382
指数の幅 (ビット)	8	≤ 11	11	≥ 15
フォーマットの幅 (ビット)	32	≥ 43	64	≥ 79

IEEE 標準は、拡張精度によって追加する余分なビット数の下限だけを指定します。拡張倍精度フォーマットの最低許容限度は、上の表によると 79 ビットになっていますが、便宜上 80 ビットフォーマットと呼ばれることがあります。この理由は、拡張精度のハードウェア実装では一般に隠れビットは使用しないので、79 ビットではなく、80 ビットを使用するからです。¹

IEEE 標準では、拡張精度にもっとも重点が置かれるので、倍精度に関する推奨は何もありません。代わりに、サポートされるもっとも広い基本フォーマットに対応する拡張フォーマットをインプリメントするように強く推奨されています。

拡張精度が採用された理由は、電卓によるところがあります。電卓は通常、10 桁を使用しますが、内部的には 13 桁を使用します。13 桁のうち、10 桁だけを表示することにより、電卓は一見、指数、コサインなどの関数を 10 桁までの精度で計算する「ブラックボックス」を使用しているように思われます。指数、対数、コサインなどの関数を 10 桁までの精度で比較的効率よく計算できる電卓では、余分な桁がいくつか必要になります。結局、500 ユニット程度の誤差で対数の近似値を計算する有理式を見つけるのはそれほど難しくはありません。したがって、13 桁を使用して計算することにより、10 桁で正しい解が得られるようになります。この余分な 3 ビットを隠れビットとして使用することにより、オペレータは単純なモデルによって電卓を操作することができます。

1. Kahan によると、拡張精度には 64 ビットの有意桁が確保されます。これは、サイクル時間を上げずに、Intel 8087 上でキャリーを伝播できるもっとも幅広い精度であるからです。

IEEE 標準の拡張精度も同じような作用を持っています。これにより、各ライブラリは単精度 (または倍精度) により 0.5 ulp 程度の誤差の範囲で効率的に計算できるようになります。したがって、ユーザーは単純なモデルによって各ライブラリを使用することができます。たとえば、乗算や対数の呼び出しなどに限らず、基本的な操作によっても、0.5 ulp 以内の誤差で正確に値が返されるようになります。ただし拡張精度を使用するときは、これを使用していることをユーザーにも明らかにすることが重要です。たとえば、電卓で表示する値の内部表現が、表示形式と同じ精度で丸められている場合、以降の操作の結果は隠れビットに依存するので、ユーザーには予測できなくなることがあります。

拡張精度についてさらに詳しく説明するために、IEEE 754 の単精度表現と 10 進数を変換するときの問題について考えてみます。可能であれば、10 進数を読み戻す時点で、単精度数も復元できるように、十分な桁数を確保して単精度数を出力するのが理想的です。単精度の 2 進数を復元するのに、9 桁の 10 進数があれば十分であることがわかっています (234 ページの「2 進数から 10 進数への変換」を参照してください)。10 進数を一意の 2 進表現に戻す場合、1 ulp 程度の小さい丸め誤差があっても誤った解になるので、致命的となります。拡張精度が効率的なアルゴリズムに不可欠となるような状況を次に説明します。単精度の場合、10 進数を単精度の 2 進数に変換するには、きわめて簡単な方法があります。まず、整数 N として 9 桁の 10 進数を読み取り、小数点は無視します。

表 D-1 から、 $p \geq 32$ の場合、 $10^9 < 2^{32} \approx 4.3 \times 10^9$ となるので、 N は単精度拡張フォーマットで正確に表現できることがわかります。次に N をスケーリングするのに必要な 10^p のべき乗を求めます。これは、10 進数の指数と、その時点までに無視された小数点の位置を考慮した組み合わせになります。 $10^{|P|}$ を計算します。 $|P| \leq 13$ であれば、 $10^{13} = 2^{13} 5^{13}$ で、 $5^{13} < 2^{32}$ となるので、これも正確に表現することができます。最後に、 N と $10^{|P|}$ を乗算 (または $p < 0$ の場合は除算) します。この最後の計算が正しく行われれば、もっとも近い 2 進数が求められます。最後の乗算 (除算) を正確に行う方法については、234 ページの「2 進数から 10 進数への変換」で詳しく説明しています。したがって、 $|P| \leq 13$ の場合、単精度拡張フォーマットを使用することにより、9 桁の 10 進数をもっとも近い 2 進数に変換 (すなわち正確な丸め) されることになります。また $|P| > 13$ の場合、単精度拡張フォーマットでは上のアルゴリズムに十分対応できないので、必ずしも正確に 2 進数が計算できるとは限りません。ただし Coonen (1984) により、2 進数から 10 進数への変換と、その逆の操作で、もとの 2 進数が復元されることが実証されています。

倍精度がサポートされていれば、上記のアルゴリズムは拡張単精度フォーマットではなく倍精度で実行されます。ただし、倍精度フォーマットを 17 桁の 10 進数に変換し、再度戻す場合には拡張倍精度フォーマットが必要になります。

指数

指数には負と正の 2 つの関係が考えられるので、符号を表現するための特定の方法を選択しなければなりません。符号付きの数は、符号と数量を組み合わせる方法と、2 の補数で表現する方法があります。符号/数量の形式は、IEEE フォーマットによる有意桁の符号を表わす場合に使用されます。すなわち 1 ビットを符号用として確保し、残りのビットで数量を表わします。また 2 の補数で表わす方法は整数演算のときによく使用されます。この方法では、 $[-2^{p-1}, 2^{p-1} - 1]$ の範囲にある数を 2^p の剰余に相当する非負の最小値で表わすことができます。

IEEE 2 進数標準で指数を表現する場合は、いずれの方法も使用しません。代わりに「バイアス表現」を使用します。指数を 8 ビットで格納する単精度の場合、バイアスは 127 となります (倍精度の場合は 1023 です)。 $\bar{k}$ が符号なしの整数を表わす指数ビットである場合、浮動小数点数の指数は $\bar{k} - 127$ になります。これはバイアス指数 $\bar{k}$ と区別するために、「バイアスなしの指数」と呼ばれることがあります。

表 D-1 を見ると、単精度の場合は $e_{\max} = 127$ 、 $e_{\min} = -126$ であることがわかります。 $|e_{\min}| < e_{\max}$ となるのは、最小値 ($1/2^{e_{\min}}$) の逆数がオーバーフローしないという理由によるものです。最大値の逆数がアンダーフローすることはありますが、これはオーバーフローの場合ほど深刻な問題ではありません。192 ページの「基数」では、0 を表わすのに $e_{\min} - 1$ を使用すると説明しました。また 198 ページの「特殊な数」では、 $e_{\max} + 1$ について説明します。IEEE 単精度の場合、これはバイアス指数の範囲が $e_{\min} - 1 = 127$ から $e_{\max} + 1 = 128$ にあるということになります。一方、バイアスなしの指数の範囲は、0 から 255 ままでとなり、8 ビットで表現できる非負の数に相当します。

操作

IEEE 標準では、四則演算の結果が正確に丸められます。すなわち結果を正確に計算した後、もっとも近い浮動小数点数に丸める (偶数への丸め) ということです。181 ページの「保護桁」では、正確な差や 2 つの浮動小数点数の和を計算するのは、指数が大きく異なるときにコスト高になると指摘しました。さらに、相対誤差を最小限に抑制しながら差を計算する便利な方法も紹介しました。ただし、1 桁の保護桁を使用して計算したからといって、必ずしも、正確な結果を計算した上で丸めを求めた場合と同等の結果が得られるとは限りません。

そこで 2 桁目の保護桁と 3 桁目のスティッキビットを使用することにより、1 桁の保護桁の場合より多少のコストは伴うものの、正確な計算の後に丸めを行なった場合と同じ結果を得ることができます (Goldberg 1990)。この方法により、標準をより効率的に実現できます。

算術演算の結果を完全に指定することにより、ソフトウェアの移植性が改善されます。あるマシンから別のマシンにプログラムを移植するときに、両方のマシンで IEEE 標準がサポートされていれば、その間に生成される中間結果はすべてソフトウェアバグに原因があり、演算上の差によるものでないことが確信できます。また正確な指定によって、浮動小数点に関する推論も行いやすくなります。浮動小数点に関する証明は、さまざまな演算処理が原因で起きる各種の問題に対処する必要がない場合でさえ、難しいものです。整数プログラムが正しいことを証明できるのと同じように、浮動小数点プログラムの場合も、証明する内容が計算結果の丸め誤差によって特定の条件が満たされるというものであっても、証明することができます。定理 4 は、このような証明の例を示したものです。こうした証明は、推論する操作の内容が正確に指定できれば、はるかに容易になります。いったん、あるアルゴリズムが IEEE 演算に適合していることがわかれば、IEEE 標準をサポートするマシンである限り、いずれであっても正しく動作します。

Brown (1981) は、一般的な浮動小数点ハードウェアに見られる原理を発表しました。ただし、このシステムでの証明が必ずしも、182 ページの「相殺」や 187 ページの「正確な丸め操作」に示すアルゴリズムの妥当性を裏付けることにはなりません。これらのアルゴリズムの特性がすべてのハードウェアでサポートされているとは限らないからです。さらに、Brown の原理は、単に正確な計算と丸め操作に関する単純な定義よりも、一段と複雑になっています。したがって、Brown の原理をもとに定理を導くのは、操作が単に正確な丸めであることを前提として証明する場合よりも難しくなります。

浮動小数点標準によって対応する操作の内容については、全面的な同意が確立されているわけではありません。+、-、×、/ の四則演算以外に、IEEE 標準では平方根、剰余、整数/浮動小数点の変換の各操作を正確に丸めるように規定されています。また、内部フォーマットと 10 進数の変換も正確に丸めなければなりません (ただし、きわめて大きい数の場合は除きます)。Kulisch と Miranker (1986) は、正確に指定された操作のリストに内積を追加することを提案しました。IEEE 演算にもとづいて内積を計算すると、最終的な解が大きく異なることがあると指摘されています。

たとえば、和は内積の特殊なケースなので、 $((2 \times 10^{-30} + 10^{30}) - 10^{30}) - 10^{-30}$ の和は 10^{-30} と同じになります。ただし IEEE 対応のマシンでは、計算結果が -10^{-30} となります。高速の乗算器を実現するより少ないハードウェアコストで、誤差を 1 ulp 以内に抑えて内積を計算することができます (Kirchner/Kulish, 1987)。^{1 2}

1. Kahan、および LeBlanc (1985) が提案した基本演算の 1 つとして内積を加えることには反論も見られます。
2. Kirchner は次のように指摘しています。「1 クロック・サイクルあたり 1 つの部分積ずつ、1 ulp 以内の誤差によりハードウェア上で内積を計算することができる。」

標準として規定される操作はすべて、10 進数と 2 進数の変換処理の場合を除き、正確に丸めを行う必要があります。これは、変換処理を除き、すべての操作について正確に丸められる効率的なアルゴリズムが実証されているからです。なお変換処理の場合は、もっとも効果的であるとされるアルゴリズムでも、正確な丸めによる計算に比べ、多少精度が落ちます (Coonen, 1984)。

IEEE 標準では、「テーブル・メーカーのジレンマ」があるために、超越関数は必ずしも正確に丸める必要はありません。これを説明するために、4 つの小数点の位置で指数関数のテーブルを作成する場合を考えてみます。 $\exp(1.626) = 5.0835$ とします。これを丸めると、5.083、または 5.084 のいずれになるのでしょうか。 $\exp(1.626)$ を慎重に計算すると、5.08350 となります。さらに 5.083500、次に 5.0835000 になります。 $\exp$ が超越関数である場合、 $\exp(1.626)$ が 5.083500 ...0ddd、または 5.0834999 ...9ddd のいずれであるかを判断できるまでに相当時間がかかる可能性があります。したがって、超越関数について、無限大の精度で計算した後、その結果を丸めた場合と同じような精度を求めるのは現実的ではありません。別のアプローチとして、超越関数をアルゴリズム的に指定することもできます。ただし、すべてのハードウェアアーキテクチャに適用できる 1 つのアルゴリズムが存在するわけではありません。最近のマシンで超越関数を計算する場合に、有理近似値、CORDIC¹、および大型テーブルという 3 つの手法を使用できます。いずれの手法もそれぞれ独自のクラスのハードウェアに適用されるもので、現在のところ、各種のハードウェアに幅広く対応できる単一のアルゴリズムはありません。

特殊な数

浮動小数点ハードウェアによっては、各ビットパターンが有効な浮動小数点数を表わす場合があります。IBM System/370 は、その一例です。一方、VAXTM では、「予約オペランド」という特殊な数を表わす特定のビットパターンが確保されています。この考え方は CDC 6600 に由来するもので、INDEFINITE と INFINITY という特殊な数のビットパターンが使用されます。

IEEE 標準には、この伝統を継承して、NaN (*Not a Number*) と無限大という概念があります。特殊な数がサポートされていないと、負数の平方根などの例外状況が起きた場合、計算をアボートする以外に対処する方法がありません。IBM System/370 FORTRAN では、-4 のような負数の平方根の計算に出会うと、デフォルトの動作とし

1. CORDIC とは、“Coordinate Rotation Digital Computer”の頭字語で、これにより、ほとんどの処理がシフトや加算で構成される (乗算や除算がほとんどない計算) 超越関数を計算することができます (Walther 1971)。CORDIC により、ハードウェアを追加した場合でも速度に応じて乗算器アレイに比較できるようになります。d は Intel 8087 と Motorola 68881 の両方で使用されます。

てエラーメッセージを出力します。各ビットパターンは、有効な数を表わすので、平方根の戻り値は浮動小数点数でなければなりません。System/370 FORTRAN の場合 $\sqrt{-4} = 2$ が戻されます。一方、IEEE 演算の場合は、NaN が戻されます。

IEEE 標準により、次の特殊な値が指定されます (表 D-2 を参照してください)。±0、非正規化数、±∞、および NaN (次の項で説明するとおり、NaN が存在します) です。これらの特殊な値は、 $e_{\max} + 1$ または $e_{\min} - 1$ の指数によってすべてエンコードされます (なお、0 には $e_{\min} - 1$ の指数があることはすでに指摘したとおりです)。

表 D-2 IEEE 754 の特殊な値

指数	関数	表示
$e = e_{\min} - 1$	$f = 0$	±0
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{\min}}$
$e_{\min} \leq e \leq e_{\max}$	—	$1.f \times 2^e$
$e = e_{\max} + 1$	$f = 0$	∞
$e = e_{\max} + 1$	$f \neq 0$	NaN

NaNs

これまで、0/0 や $\sqrt{-1}$ などの計算は、その時点で計算の停止の原因となる回復不能なエラーとして扱われてきました。ただし、こうした状況でも計算をそのまま続行することに意味があるケースも考えられます。たとえば、ある関数 f のゼロを求めるサブルーチン、zero(f) の例を考えてみます。従来、ゼロ検出サブルーチンが検索を行う期間として、関数が定義された間隔 $[a,b]$ をユーザーが指定しなければなりません。このサブルーチンのことを zero(f,a,b) と言います。さらに便利なゼロ検出サブルーチンでは、この追加情報を入力する必要はありません。この汎用的なゼロ検出サブルーチンは、単に関数の入力だけが要求され、領域の指定などは無意味となるような電卓などに適用できます。しかし、実際にはほとんどのゼロ検出サブルーチンで領域の指定が必要になります。その理由は簡単です。ゼロ検出サブルーチンはいくつかの値で関数 f を調べることで処理を行います。 f の領域を超えた値があると、 f に対するコードが 0/0 または $\sqrt{-1}$ になり、計算が停止して、ゼロ検出プロセスをアボートしてしまうことがあるからです。

この問題は、NaN という特殊な値を導入し、0/0 や $\sqrt{-1}$ の式の計算では、処理が停止する代わりに NaN を生成するように指定すれば、解決します。表 D-3 には、NaN の原因となる状況をいくつか示してあります。zero(f) で f の領域を超えた値があると、 f のコードにより NaN が返され、ゼロ検出サブルーチンの処理が続行します。

すなわち、`zero(f)` は誤った推測をしたことについて「とがめられる」ことはありません。この例をもとに、NaN と通常の浮動小数点数を組み合わせた場合の結果を簡単に確認することができます。`f` の最後の文は `return (-b + sqrt(d))/(2*a)` となります。 $d < 0$ の場合、`f` は NaN を返します。 $d < 0$ なので、NaN と別の数の和が NaN の場合は、`sqrt(d)` は NaN、`-b + sqrt(d)` も NaN になります。同様に、除算のいずれかのオペランドが NaN の場合、商も NaN になります。一般に、浮動小数点計算で NaN が要求された場合は、結果も NaN になります。

表 D-3 NaN が生成される操作

操作	NaN が生成される原因
+	$\infty + (-\infty)$
x	$0 \times \infty$
/	$0/0, \infty / \infty$
REM	$x \text{ REM } 0, \infty \text{ REM } y$
$\sqrt{}$	$\sqrt{x}$ ($x < 0$ のとき)

ユーザーが領域を入力する必要のないゼロ・ソルバを作成する別のアプローチとして、シグナルを使用することもできます。ゼロ検出サブルーチンは、浮動小数点例外に対するシグナルハンドラをインストールすることができます。`f` が領域外と評価され、例外が発生すると、制御はゼロ・ソルバに渡されます。このアプローチの問題点は、各言語ごとにシグナルを処理する方法が異なる (その方法がない場合もあります) ので、移植性がまったくないという点です。

IEEE 754 では、NaN は指数 $e_{\max} + 1$ とゼロ以外の有意桁を伴う浮動小数点数として表わすことがあります。実装ごとに、システム依存の情報を有意桁に入れることができます。したがって、一意の NaN が存在するわけではなく、一連の NaN で全体が構成されることになります。NaN と通常の浮動小数点数を組み合わせると、結果は NaN オペランドと同じになります。したがって、長い計算の結果が NaN の場合、有意桁にあるシステム依存の情報は、最初の NaN が生成された時点で生成される情報となります。実際に、最後の文には注意が必要です。両方のオペランドが NaN の場合、結果はいずれか一方の NaN になるので、必ずしも最初に生成された NaN であるとは限りません。

無限大

NaN を使用して、 $0/0$ や $\sqrt{-1}$ のような式が検出されたときに計算を続行させる場合と同じように、無限大によってオーバーフローが起きたときに処理を続けることができます。これは単に最大許容数を戻すよりも、はるかに安全な方法です。たとえば、 $\beta = 10$ 、 $p = 3$ 、 $e_{\max} = 98$ のときの $\sqrt{x^2 + y^2}$ を考えてみます。 $x = 3 \times 10^{70}$ 、 $y = 4 \times 10^{70}$ の場合、 x^2 はオーバーフローし、 9.99×10^{98} に置き換えられます。同じように、 y^2 、および $x^2 + y^2$ はそれぞれ順にオーバーフローして、 9.99×10^{98} に置き換えられます。最終的には、 $\sqrt{9.99 \times 10^{98}} = 3.16 \times 10^{49}$ となり、正解の 5×10^{70} とはほど遠い結果となります。IEEE 演算の場合、 x^2 の結果は、 y^2 、 $x^2 + y^2$ 、 $\sqrt{x^2 + y^2}$ と同じように、 ∞ となります。最終的な結果は ∞ となり、これは正しい解から大きくはずれた通常の浮動小数点数を戻すより、はるかに安全であると言えます¹。

0 を 0 で除算しても、NaN になります。ゼロ以外の数を 0 で割ると、 ∞ が返されます ($1/0 = \infty$ 、 $-1/0 = -\infty$)。これを区別する理由は次のとおりです。 x がある限界に接近するに従い、 $f(x) \rightarrow 0$ 、 $g(x) \rightarrow 0$ になる場合、 $f(x)/g(x)$ の値は任意になります。たとえば、 $f(x) = \sin x$ 、 $g(x) = x$ の場合、 $x \rightarrow 0$ だと $f(x)/g(x) \rightarrow 1$ ですが、 $f(x) = 1 - \cos x$ の場合、 $f(x)/g(x) \rightarrow 0$ になります。 $0/0$ を、2 つの小さい値の商を制限する条件として捉えると、 $0/0$ は任意になります。したがって、IEEE 標準で、 $0/0$ は NaN になります。ただし、 $c > 0$ の場合、 $f(x) \rightarrow c$ 、 $g(x) \rightarrow 0$ であれば、あらゆる分析関数 f と g について $f(x)/g(x) \rightarrow \pm \infty$ となります。小さい x に対して $g(x) < 0$ の場合、 $f(x)/g(x) \rightarrow -\infty$ となります。これ以外の場合は、限界が $+\infty$ になります。したがって、IEEE 標準では、 $c \neq 0$ である限り、 $c/0 = \pm \infty$ が定義されます。 ∞ の符号は、通常の c と 0 の符号に依存するので、 $-10/0 = -\infty$ 、また $-10/-0 = +\infty$ となります。オーバーフローが原因で生成される ∞ と、ゼロによる除算が原因で起きる ∞ との違いは、ステータスフラグをチェックすれば区別できます (詳細については、210 ページの「フラグ」で説明しています)。前者の場合はオーバーフロー・フラグが、また後者の場合はゼロによる除算のフラグがそれぞれセットされます。

オペランドとして無限大を生成する操作の結果を決定するルールは簡単です。無限大を有限数 x に置き換え、その限界を $x \rightarrow \infty$ に設定します。したがって、 $\lim_{x \rightarrow \infty} 3/x = 0$ なので、 $3/\infty = 0$ になります。

同様に、 $4 - \infty = -\infty$ 、 $\sqrt{\infty} = \infty$ になります。限界がない場合、結果は NaN になるので、 ∞ / ∞ も NaN になります (200 ページの表 D-3 に追加の例を示しています)。これは、 $0/0$ が NaN になると結論する場合の推論と一致します。

1. これは、微妙な点です。IEEE 演算のデフォルトではオーバーフローした数が ∞ に丸められますが、デフォルトを変更することもできます (209 ページの「丸めモード」を参照してください)。

ある部分式が NaN と評価されると、式全体の値も NaN になります。ただし、 $\pm \infty$ の場合、 $1/\infty = 0$ のようなルールがあるので、式の値は通常の浮動小数点数になる場合があります。ここでは、無限大の演算についてルールを適用する実際の例を示します。関数 $x/(x^2 + 1)$ を計算する場合を考えてみます。

これは、 x が $\sqrt{\beta}e_{\max}^{1/2}$ より大きくなるとオーバーフローするので、よい公式ではありませんが、無限大の計算により、 $1/x$ の近似値ではなく 0 が生成されるので、誤った解が得られます。ただし $x/(x^2 + 1)$ は $1/(x + x^{-1})$ と書き換えることができます。この書き換えにより、早い段階でオーバーフローが起きることはなくなり、無限大の演算により、 $x = 0 : 1/(0 + 0^{-1}) = 1/(0 + \infty) = 1/\infty = 0$ のときに正しい値が得られます。無限大の演算がないと、 $1/(x + x^{-1})$ の式では、 $x = 0$ かどうかをテストする必要があります。この場合、余分な命令が追加されるほか、パイプラインが破壊される可能性があります。この例は、無限大の演算によって、特殊なケースをチェックする必要がなくなるという一般的な事例を示したものです。ただし公式については、無限大時に、疑似的な動作 ($x/(x^2 + 1)$ のように) を示すことがないように、慎重に調べる必要があります。

符号付きゼロ

ゼロは、指数 $e_{\min} - 1$ とゼロの有意桁によって表わすことができます。符号ビットは 2 つの値を取ることができるので、ゼロにも +0 と -0 の 2 種類があります。+0 と -0 を比較する場合に、区別してしまうと、`if (x=0)` のような簡単なテストでも、 x の符号によって予測できない結果になることがあります。したがって、IEEE 標準では、 $-0 < +0$ ではなく、 $-0 = +0$ となるように定義されています。ゼロの符号をつねに無視することも可能ですが、IEEE 標準では無視されません。乗算、または除算で符号付きのゼロを伴う場合、解の符号を計算するときには通常の符号が適用されます。したがって、 $3 \cdot (+0) = +0$ 、および $+0 / -3 = -0$ になります。またゼロに符号がない場合は、 $x = \pm \infty$ の場合、 $1/(1/x) = x$ は成り立たなくなります。その理由は $1/-\infty$ と $1/+\infty$ はいずれも 0、 $1/0$ は $+\infty$ となり、符号情報が失われるからです。 $1/(1/x) = x$ の意味を保持するには、一方の無限大だけを使用します。ただしこの場合も、オーバーフローした数の符号が失われるという結果は変わりません。

符号付きゼロを使用するもう 1 つの例は、アンダーフローと、 $\log$ のような 0 で不連続となる関数に関するものです。IEEE 演算では、 $x < 0$ のときに、 $\log 0 = -\infty$ 、 $\log x$ を NaN と定義するのが当然になっています。ゼロにアンダーフローした小さい負数を表わす x の例を考えてみます。符号付きのゼロのために、 x は負数になるので、 $\log$ は NaN を返すことができます。ただし、符号なしのゼロの場合、 $\log$ 関数はアンダーフローした負数と 0 とを区別できなくなるので、 $-\infty$ を返します。ゼロで不連続になるもう 1 つの関数の例として、数値の符号を返す `signum` 関数があります。

符号付きゼロのもっとも興味のある例は、複素数演算です。

単純な例として、 $\sqrt{1/z} = 1/(\sqrt{z})$ という方程式を考えてみます。これは、 $z \geq 0$ の場合に真になります。

$z = -1$ の場合は、 $\sqrt{1/(-1)} = \sqrt{-1} = i$ 、および $1/(\sqrt{-1}) = 1/i = -i$ となります。

したがって、 $\sqrt{1/z} \neq 1/(\sqrt{z})$ となります。この問題は、平方根が複数の値となり、複素数平面全体に連続する値を選択できないという事実に由来するものです。ただし、すべて負の実数で構成されるブランチカットを考慮の対象から外せば、平方根は連続になります。それでも、 $-x + i0$ (ただし $x > 0$) の形式の負の実数について、どう扱うかという問題は残ります。符号付きゼロにより、この問題を完全な形で解決することができます。 $x + i(+0)$ の形式の数は符号と $(i\sqrt{x})$ 、またブランチカットのもう一方にある $x + i(-0)$ の形式の数は、逆の符号と $(-i\sqrt{x})$ になります。実際に、 $\sqrt{}$ を計算する公式により、次の結果が得られます。

$\sqrt{1/z} = 1/(\sqrt{z})$ となるので、 $z = -1 + i0$ であれば、 $1/z = 1/(-1 + i0) = [(-1 - i0)] / [(-1 + i0)(-1 - i0)] = (-1 - i0)/((-1)^2 - 0^2) = -1 + i(-0)$ となり、 $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$ 、一方 $1/(\sqrt{z}) = 1/i = -i$ となります。したがって、IEEE 演算では、すべての z の意味が保持されます。さらに高度な例が Kahan によって紹介されています (1987)。+0 と -0 を区別することに利点がありますが、わかりにくくなることがあります。たとえば、符号付きゼロにより、 $x = +0$ 、および $y = -0$ のときに偽となる $x = y \Leftrightarrow 1/x = 1/y$ の関係が壊れます。ただし、IEEE 委員会では、符号付きゼロを使用した場合の利点の方が欠点を上回っていると判断しています。

非正規化数

$\beta = 10$ 、 $p = 3$ 、 $e_{\min} = -98$ で正規化された浮動小数点数の例を考えてみます。 $x = 6.87 \times 10^{-97}$ と $y = 6.81 \times 10^{-97}$ は、最小の浮動小数点数 1.00×10^{-98} の 10 倍以上大きいので、通常の浮動小数点数のように思われます。しかし、これらの浮動小数点数には、 $x \neq y$! であっても、 $x \ominus y = 0$ になるという変わった特性があります。これは、 $x - y = 0.06 \times 10^{-97} = 6.0 \times 10^{-99}$ が、正規化数としては小さすぎて表現できないため、ゼロにフラッシュしなければならないという理由によるものです。この特性を保持することは、どの程度重要なのでしょう。

$$x = y \Leftrightarrow x - y = 0? \quad (10)$$

if ($x \neq y$) then $z = 1/(x - y)$ というコード・フラグメントを記述し、後で、見かけ上のゼロ除算によってプログラムをアボートさせることも簡単にできます。しかし、このようなバグ検出の方法は、心理的にも時間的にも大きな負担になります。さ

らに高度な方法として、コンピュータサイエンスの参考書には、現在のところ大規模なプログラムの妥当性を検証するのは現実的ではないものの、これを実証することを念頭にプログラムを設計すると、効率的なコードができあがると指摘されています。たとえば、不変式を取り入れるのは、仮にこれが証明の一部としては利用できない場合でも、効果的な手段です。浮動小数点コードによって、通常のコードの場合と同じように、信頼できる証明可能な事実が得られるようになります。たとえば、公式 (6) を分析する場合、 $x/2 < y < 2x \Rightarrow x \ominus y = x - y$ という事実がわかれば非常に役立ちます。同じように、公式 (10) が真であることがわかれば、信頼できる浮動小数点コードも記述しやすくなります。大部分の数にあてはまるからという前提は、何も証明することにはなりません。

IEEE 標準では、非正規化¹数を使用して公式 (10) 、およびその他の関係を証明します。非正規化数はよく議論の対象となる部分であり、754 で認可されるまでに長い期間がかかったのは、おそらくこの理由によるものと思われます。パフォーマンスの高いハードウェアはほとんどの場合、IEEE 互換であることがすなわち、非正規化数を直接サポートすることとは限りません。非正規化数を使用、あるいは生成する時点でハードウェアがトラップされ、ソフトウェアに IEEE 標準のシミュレーションを一任する形式になっています²。非正規化数の背景にある考えは、Goldberg (1967) が提案したもので、きわめて単純です。指数の $e_{\min}$ の場合、有意桁は正規化する必要がないので、 $\beta = 10$ 、 $p = 3$ 、 $e_{\min} = -98$ であれば、 0.98×10^{-98} も浮動小数点数になるので、 1.00×10^{-98} は最小の浮動小数点数ではなくなります。

$\beta = 2$ の場合に隠れビットを使用すると、 $e_{\min}$ の指数を伴う数は、暗黙に先行するビットがあるために必ず有意桁が 1.0 以上になるので、多少の支障が出てきます。この解決方法は、0 を表わす場合のそれと同じで、199 ページの表 D-2 にまとめてあります。 $e_{\min}$ 指数を使用して、非正規化数を表わすことができます。すなわち有意桁フィールドのビットが $b_1, b_2, \dots, b_{p-1}$ で、指数の値が e の場合、表現する数は $1.b_1b_2 \dots b_{p-1} \times 2^e$ となり、 $e = e_{\min} - 1$ の場合は $0.b_1b_2 \dots b_{p-1} \times 2^{e+1}$ となります。非正規化数には $e_{\min} - 1$ ではなく、 $e_{\min}$ の指数を伴うので、指数の +1 は必須になります。

本項の最初に示した $\beta = 10$ 、 $p = 3$ 、 $e_{\min} = -98$ 、 $x = 6.87 \times 10^{-97}$ 、 $y = 6.81 \times 10^{-97}$ の例を再び取り上げます。非正規化数の場合、 $x - y$ はゼロへのフラッシュは行われず、代わりに 0.6×10^{-98} という非正規化数で表わされます。この動作のことを段階的な「アンダーフロー」と言います。段階的アンダーフローを使用すれば、公式 (10) がつねに真になることを簡単に検証することができます。

1. これは、854 ではサブノーマル、754 では非正規と呼ばれます。

2. これが、標準化のもっとも複雑な側面です。頻繁にアンダーフローするプログラムは、ソフトウェアトラップを使用するハードウェアでは極端にパフォーマンスが低下します。

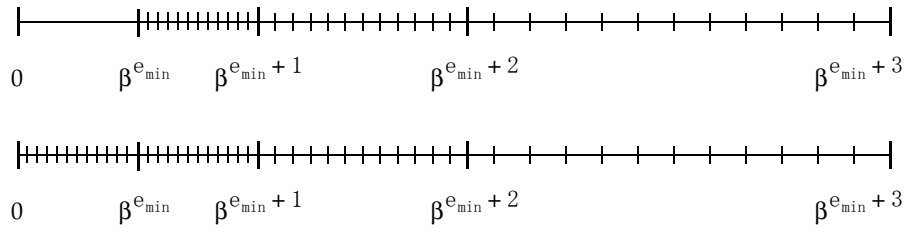


図 D-2 段階的アンダーフローとゼロフラッシュの比較

図 D-2 に、非正規化数を示します。上の数線は、正規化した浮動小数点数を表わします。0 から最小の正規化数 $1.0 \times \beta^{e_{\min}}$ までに格差があることに注意してください。浮動小数点演算の結果がこの差の範囲に入る場合は、ゼロへのフラッシュが行われます。下の数線は、一連の浮動小数点数に非正規化数を加えたときに、どう処理されるかを示したものです。この格差が埋められ、計算結果が $1.0 \times \beta^{e_{\min}}$ より少なければ、もっとも近い非正規化数で表わされます。数線に非正規化数を加算すると、隣接する浮動小数点数どうしの間隔が規則的に取られます。隣接する数どうしの間隔は同じ場合と、 β の係数によって異なる場合があります。非正規化数がなければ、 β^{p-1} から $\beta^{e_{\min}}$ までの間隔が、 β の係数による規則的な変化ではなく、 β^{p-1} の係数に従って、急激に変化することになります。このため、アンダーフローの限界に近い正規化数に対して大きな相対誤差を伴うアルゴリズムはほとんどの場合、段階的アンダーフローを使用すると、この範囲内で正しく動作するようになります。

段階的アンダーフローがなければ、 $x - y$ のような単純な式でも、上の $x = 6.87 \times 10^{-97}$ 、および $y = 6.81 \times 10^{-97}$ の例のように、正規化された入力に対して非常に大きな相対誤差を伴うことになります。大きな相対誤差は、次の例 (Demmel, 1984) に示すように、相殺がない場合でも起きる可能性があります。 $a + ib$ と $c + id$ という 2 つの複素数を除算する例を考えてみます。次の公式には、分母 $c + id$ のいずれかの要素が $\sqrt{\beta} \beta^{e_{\max}/2}$ より大きいときに、最終結果が範囲内にあっても、オーバーフローするという問題があります。

$$\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

この商を計算する場合、次の Smith の公式を使用すると、より効果的です。

$$\frac{a+ib}{c+id} = \begin{cases} \frac{a+b(d/c)}{c+d(d/c)} + i \frac{b-a(d/c)}{c+d(d/c)} & \text{if } (|d| < |c|) \\ \frac{b+a(c/d)}{d+c(c/d)} + i \frac{-a+b(c/d)}{d+c(c/d)} & \text{if } (|d| \geq |c|) \end{cases} \quad (11)$$

Smith の公式を $(2 \cdot 10^{-98} + i10^{-98}) / (4 \cdot 10^{-98} + i(2 \cdot 10^{-98}))$ に適用すると、段階的アンダーフローによって 0.5 という正しい解が得られます。ゼロ・フラッシュの場合は、0.4 になり、誤差は 100 ulp となります。これは、 $1.0 \times \beta^{\epsilon_{\min}}$ を下限とする引数の誤差範囲を保証する非正規化数としては典型的な値です。

例外、フラグ、トラップハンドラ

IEEE 演算で、ゼロによる除算、あるいはオーバーフローといった例外が起きると、デフォルトとして結果を報告し、処理を続行します。一般的なデフォルトとして、0/0 と $\sqrt{-1}$ には NaN、また 1/0 とオーバーフローには ∞ を返します。前の項では、例外が起きた場合でも、以上のデフォルト値によって処理を続行する方が望ましい例をいくつか紹介しました。いずれかの例外が起きると、ステータスフラグもセットされます。IEEE 標準の仕様では、ユーザーがステータスフラグを読み取り/書き込みできる手段を提供しなければなりません。フラグは、いったんセットされると、明示的にクリアしない限り、そのままセット状態になるという意味で「スティッキー」な特性があります。本来の無限大を表わす 1/0 と、オーバーフローとの違いを区別するには、フラグをテストする以外に方法はありません。

例外が起きたときに、処理を続行することが妥当でない場合もあります。201 ページの「無限大」では、 $x/(x^2+1)$ の例を示しました。

$x > \sqrt{\beta^{\epsilon_{\max}}/2}$ の場合、分母は無限大になるので、最終的な解が 0 という、まったく誤った結果になります。この公式の場合、 $1/(x+x^{-1})$ と書き換えれば問題は解決しますが、式の書き換えによって必ずしも問題が解決しない場合もあります。IEEE 標準では、トラップハンドラをインストールできるような仕様が強く推奨されています。例外が起きると、フラグをセットする代わりにトラップハンドラが呼び出され、トラップハンドラが返す値を操作の結果として使用します。ステータスフラグをクリアまたはセットするという判断は、トラップハンドラに一任されます。これ以外の場合、フラグの値は未定義にすることができます。

IEEE 標準では、例外をその種類によって、オーバーフロー、アンダーフロー、ゼロによる除算、無効な操作、および不正確な操作という 5 つのクラスに分類しています。例外の各クラスごとに、個別のステータスフラグが用意されています。最初の 3 つの例外の意味はその名が示すとおりです。無効な操作とは、200 ページの表 D-3 に示す状況と、NaN を伴うあらゆる比較操作のことです。無効操作例外の原因となる操作で

は、デフォルトとして NaN が返されます。ただし逆は真ではありません。ある操作に対するオペランドのいずれかが NaN の場合、結果も NaN になりますが、操作の内容が 200 ページの表 D-3 に示す条件のいずれかに一致しない限り、無効操作の例外は起きません¹。

表 D-4 IEEE 754の例外¹

例外	トラップが無効な場合の結果	トラップハンドラの引数
オーバーフロー	$\pm \infty$ または $\pm x_{\max}$	$\text{round}(x2^{-\alpha})$
アンダーフロー	$0, \pm 2^{e_{\min}}$ または denormal	$\text{round}(x2^{\alpha})$
ゼロによる除算	∞	オペランド
無効な操作	NaN	オペランド
不正確な操作	$\text{round}(x)$	$\text{round}(x)$

1. x は操作の正確な結果、単精度は $\alpha=192$ 、倍精度は $\alpha=1536$ 、 $x_{\max}=1.11 \dots 11 \times 2^{e_{\max}}$ となります。

$\beta = 10$ 、 $p = 3$ のシステムで、 $3.5 \otimes 4.2 = 14.7$ は正確ですが、 $3.5 \otimes 4.3 = 15.0$ は正確ではないので (正しくは $3.5 \cdot 4.3 = 15.05$)、不正確操作の例外が起きます。234 ページの「2 進数から 10 進数への変換」では、不正確操作の例外を使用したアルゴリズムについて説明しています。また表 D-4 に、5 種類の例外の動作をまとめてあります。

不正確な操作の例外が頻繁に起きる場合は、実装上の問題があるものと思われます。浮動小数点ハードウェアに独自のフラグが用意されておらず、代わりにオペレーティングシステムに割り込んで浮動小数点例外を報告する形式の場合は、不正確な操作の例外が起きたときのコストはきわめて高くなります。この問題は、ソフトウェアが保持するステータスフラグを使用すれば、解決します。例外が初めて起きた時点で、該当するクラスのソフトウェアフラグをセットし、浮動小数点ハードウェアに、対応する例外クラスをマスクするように指示します。これ以降の例外はすべて、オペレーティングシステムへの介入を伴わずに、処理されるようになります。ユーザーがステータスフラグをリセットすると、ハードウェアマスクが再度イネーブルになります。

1. 操作に「トラップ」する NaN が伴わない限り、無効操作の例外は起きません。詳細については、IEEE Std 754-1985 の 6.2 項を参照してください。-Ed

トラップハンドラ

トラップハンドラを使用する意味は、まず下位の互換性を保持することにあります。例外が起きた時点でアボートとなる旧バージョンのコードでは、プロセスを打ち切るためのトラップハンドラをインストールすることができます。これは特に、`do s until (x ≥ 100)` のようなループを使用するコードに有効になります。NaN は、`<`、`≤`、`>`、`≥`、`=` (`≠` はなし) を使用した数に比較されると必ず偽を返すので、`x` が NaN になると、このコードは無限ループに入ります。

オーバーフローする可能性のある $\prod_{i=1}^n x_i$ のような積を計算するときに使用されるトラップハンドラをさらに別の用途として利用することもできます。対数を使用して、 $\exp(\sum \log x_i)$ を計算する方法です。このアプローチの問題は、精度に欠けることに加え、オーバーフローが起きない場合でも単純な $\prod x_i$ 式よりもコストが大きくなる点にあります。またこれらの問題を避けるためのオーバーフロー/アンダーフロー・カウンティングというトラップハンドラを使用する方法もあります (Sterbenz、1974)。

この考え方は次のとおりです。まずゼロに初期化される汎用カウンタがあります。 $p_k = \prod_{i=1}^k x_i$ が特定の k に対してオーバーフローすると必ず、トラップハンドラがカウンタの値を 1 だけ増やし、指数をラップしてオーバーフローの値を返します。IEEE 754 の単精度では、 $e_{\max} = 127$ となるので、 $p_k = 1.45 \times 2^{130}$ であれば、オーバーフローを起こし、トラップハンドラが呼び出されます。トラップハンドラは指数をもとの範囲にラップし、 p_k を 1.45×2^{-62} (下記参照) に変更します。同様に、 p_k がアンダーフローすると、カウンタの値が減り、負の指数が正の指数にラップされます。乗算がすべて終わった時点でカウンタがゼロの場合、最終的な積は p_n になります。カウンタが正であれば積はオーバーフローし、負であればアンダーフローします。部分積がいずれも範囲を超えていない場合は、トラップハンドラは呼び出されず、計算により余分なコストが発生することはありません。オーバーフローまたはアンダーフローが起きた場合でも、 p_k はそれぞれ、全面的な精度の乗算を使用して p_{k-1} から計算されているので、対数で計算した場合よりもはるかに正確です。Barnett (1987) は、オーバーフロー/アンダーフロー・カウンタの全面精度によって、前出の表で該当する誤差を検出できる公式を紹介しました。

IEEE 754 の仕様では、オーバーフロー/アンダーフロー・トラップハンドラが呼び出された時点で、これがラップの結果を引き数として渡されます。オーバーフローに対するラップは、結果を無限の精度で計算し、これを 2α で除算した後、対応する精度に丸める操作と定義されます。アンダーフローの場合は、結果を 2α で乗算します。指数 α は単精度の場合は 192、倍精度の場合は 1536 となります。これは、 1.45×2^{130} が 1.45×2^{-62} に変換されるからです。

丸めモード

IEEE 標準では、それぞれの操作を正確に計算し、その結果が丸められる (2 進数から 10 進数への変換は除く) ので、正確でない結果が生成されると必ず、丸めが行われます。丸めとは本来、もっとも近い値で表わすということです。これ以外に標準には、0 への丸め、 $+\infty$ への丸め、 $-\infty$ への丸めという 3 種類の丸めモードがあります。 $-\infty$ への丸めを整数操作への変換とともに使用すると、変換処理は切り下げ関数として行われます。一方 $+\infty$ に丸めると、切り上げ関数になります。0 への丸め、または $-\infty$ への丸めが有効になっているときに、正の値がオーバーフローすると、デフォルトの結果が $+\infty$ ではなく、表現可能な最大値になるので、丸めモードはオーバーフローに影響を与えることになります。同様に、 $+\infty$ への丸め、または 0 への丸めが有効になっているときに負の値がオーバーフローすると、負の最大値が生成されます。

丸めモードの 1 つの応用例として、インタバルの計算があります (もう 1 つの応用例は、234 ページの「2 進数から 10 進数への変換」で示します)。インタバルの計算を使用する場合、2 つの値 x 、 y の和をインタバル $[z, \bar{z}]$ として扱います。ここで $\underline{z}$ は $x \oplus y$ を $-\infty$ に向かって丸め、 $\bar{z}$ は $x \oplus y$ を $+\infty$ に向かって丸めるという意味です。加算の正確な結果は、インタバル $[z, \bar{z}]$ の範囲にあります。丸めモードがなければ、インタバルの計算は通常、 $\underline{z} = (x \oplus y)(1 - \epsilon)$ と $\bar{z} = (x \oplus y)(1 + \epsilon)$ (ここで ϵ はマシン・イプシロン) によって実現します¹。

この結果、インタバルのサイズが多めに見積もられます。インタバル計算の結果はインタバルとして計算されるので、通常、操作に対する入力もインタバルで指定します。2 つのインタバル $[x, \bar{x}]$ と $[y, \bar{y}]$ を加算すると、結果は $[z, \bar{z}]$ (ここで $\underline{z}$ は丸めモードを $-\infty$ にセットした $x \oplus y$ 、また $\bar{z}$ は丸めモードを $+\infty$ にセットした $\underline{z} \oplus \bar{y}$) になります。

インタバル計算を使用して浮動小数点演算を行うと、最終的な解は、正確な計算結果を含むインタバルとして得られます。これは、インタバルが広すぎる場合 (よく起きることです)、正しい解がそのインタバル内の任意の値になるので、それほど便利ではありません。インタバルの計算は、多重精度の浮動小数点パッケージと併用すれば、さらに意味を持ちます。まず、特定の精度 p とともに計算を行います。インタバル計算の結果により、最終的な解が不正確であることがわかれば、最終的なインタバルが適切なサイズになるまで段階的に精度を上げながら、何度か計算しなおします。

1. x と y の両方が負の場合、 $\underline{z}$ が $\bar{z}$ より大きくなる場合があります。

フラグ

IEEE標準では、いくつかのフラグとモードがサポートされています。すでに説明したとおり、5種類の例外(アンダーフロー、オーバーフロー、ゼロによる除算、無効な操作、不正確な操作)に対してステータスフラグが用意されています。また丸めモードには、もっとも近い値への丸め、 $+\infty$ への丸め、 0 への丸め、 $-\infty$ への丸めの4種類があります。それぞれの例外に対してイネーブル・モード・ビットを1つ確保するように強くお勧めします。本項では、各モードとフラグをどのように有効に利用できるかを示す例をいくつか紹介します。さらに高度な例については、234ページの「2進数から10進数への変換」に示しています。

x^n (ここで n は整数) を計算するサブルーチンの例を考えてみます。 $n > 0$ の場合は、次のような単純なルーチンが使用できます。

```
PositivePower(x,n) {
  while (n is even) {
    x = x*x
    n = n/2
  }
  u = x
  while (true) {
    n = n/2
    if (n==0) return u
    x = x*x
    if (n is odd) u = u*x
  }
}
```

一方、 $n < 0$ の場合に x^n を計算するさらに正確な方法は、PositivePower ($1/x$, $-n$)ではなく、 $1/\text{PositivePower}(x, -n)$ を呼び出すことです。この理由は、最初の式の場合、乗算を行う n の値ごとに、 $1/x$ の除算で得られた丸め誤差が含まれることになるからです。2つ目の式では、正確な値の計算 (x) にもとづいて、最後に行う除算で丸め誤差が1回だけ伴います。しかし、この方法にも欠点があります。

PositivePower (x , $-n$) がアンダーフローになると、アンダーフロー・トラップハンドラが呼び出されるか、あるいはアンダーフロー・ステータスフラグがセットされます。この動作は、 x^n がアンダーフローした時点で、 x^n はオーバーフローになるか、範囲内に収まるかのいずれかになるので、誤りです¹。しかし、IEEE 標準ではユーザーがすべてのフラグにアクセスできるので、サブルーチンにより簡単にこれを修正することができます。単に、オーバーフロー/アンダーフロー・イネーブル・ビットをオフにして、オーバーフロー/アンダーフロー・ステータスビットを保存するだけのことです。この後、 $1/\text{PositivePower}(x, -n)$ を計算します。オーバーフロー、 $1.x < 1, n < 0$ で x^n が、アンダーフローの限界 $2^{e_{\min}}$ よりわずかに小さい場合は、 $x^n \approx 2^{-e_{\min}} < 2^{e_{\max}}$ となるので、IEEE の精度はすべて $-e_{\min} < e_{\max}$ となることにより、オーバーフローは起きません。

アンダーフローのいずれのステータスビットもセットされなければ、トラップ・イネーブル・ビットとともにステータスビットがリストアされます。いずれかのステータスビットがセットされた場合は、フラグがリストアされ、PositivePower (1/x, -n) を使用して再計算します。この場合は、適切な例外が起きます。

フラグを使用するもう 1 つのケースは、 $\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}$ という公式で $\arccos$ を計算するときに起きます。 $\arctan(\infty)$ の評価が $\pi/2$ となる場合、 $\arccos(-1)$ は、無限大の演算に従って $2 \cdot \arctan(\infty) = \pi$ に正しく評価されます。ただし、 $(1-x)/(1+x)$ の計算により、 $\arccos(-1)$ は例外にならなくても、ゼロによる除算例外のフラグがセットされます。この問題は簡単に解決できます。まずゼロによる除算フラグの値を保存してから、 $\arccos$ を計算し、計算が終了した後、もとの値をリストアします。

システムの側面

コンピュータシステムのあらゆる側面を設計する場合に、浮動小数点に関する知識が必要になります。コンピュータアーキテクチャには通常、浮動小数点命令が組み込まれているので、コンパイラでこれらの浮動小数点命令を生成すると同時に、オペレーティングシステムでは、各浮動小数点命令に対して例外条件が発生した場合の処置を判断しなければなりません。一般に、数値解析の解説書を読んでも、基本的な内容はソフトウェアのユーザーや開発者を対象に書かれているので、コンピュータシステムの設計者にはそれほど参考にはなりません。設計上の方針が、どの程度予期せぬ方向に発展するかを示す実例として、次の BASIC プログラムを考えてみます。

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
    else print "Not Equal"
```

Borland 社の Turbo Basic を IBM PC 上で実行しながら、“Not Equal” と出力するプログラムです。この例の分析は、次の項で行います。

なお、上記のような特殊な例は浮動小数点数と同列に扱うことはできないという指摘もあると思われます。しかし、いずれの例も特定の範囲 E の中に収まるものという意味で、同じであると解釈してください。これは、多くの答えになると同時に、多くの疑問も呈するので、万能的な解答にはならないのは言うまでもありません。 E の値はどのようになるのでしょうか。 $x < 0$ と $y > 0$ の値が E の範囲内にある場合、それぞれ異

なる符号があるのに、実際にこれが等価であると言えるでしょうか。さらに、 $a \sim b \Leftrightarrow |a - b| < E$ というルールで定義される関係は、 $a \sim b$ と $b \sim c$ が $a \sim c$ を意味するものではないので、等価の関係とみなすことはできません。

命令セット

あるアルゴリズムで、正確な結果を生成するために、精度の高いショートバーストが必要になることがあります。

一例として、二次方程式の根の公式 $(-b \pm \sqrt{b^2 - 4ac}) / 2a$ があげられます。231 ページの「定理 4 の証明」に説明するとおり、 $b^2 \approx 4ac$ のとき、丸め誤差により二次方程式の根の公式で求めた根の桁の半分が汚染される可能性があります。 $b^2 - 4ac$ の部分計算を倍精度で行うことにより、根の倍精度ビットの半分 (すなわち単精度ビットの全部) が保持されます。

a 、 b 、 c の各値が単精度フォーマットのときに、2 つの単精度数を取り、倍精度の結果を生成する乗算命令があれば、 $b^2 - 4ac$ を倍精度で計算するのは簡単です。2 つの p 桁数を正確に丸めるには、処理の進行に従ってビットが破棄される可能性はあるものの、積の $2p$ ビット全体を生成するために乗数が必要になります。したがって、単精度オペランドから倍精度の積を計算するハードウェアは通常、単精度の乗数の場合に比べコストが高く、また倍精度の乗数よりもコストが低くなります。これにもかかわらず、最近の命令セットはオペランドと同じ精度の結果を生成する命令だけを提供する傾向があります¹。

2 つの単精度オペランドを組み合わせて倍精度の結果を生成する命令が、二次方程式の根の公式に限って適用される場合、命令セットに追加しても意味がありません。しかし、この命令にはさまざまな使い方があります。次の一次方程式の系を求める問題を考えてみます。

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

...

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

1. これはおそらく、設計者が浮動小数点命令の精度が実際の操作からは独立した直行関係の命令セットを好む傾向を反映したものです。乗算で特別なケースを認めることにより、この直行関係がなくなります。

これは、次のように $Ax = b$ として行列形式に書き換えることができます。

$$A = \begin{bmatrix} a_{11} & a_{12} \dots & a_{1n} \\ a_{21} & a_{22} \dots & a_{2n} \\ a_{n1} & a_{n2} \dots & a_{nn} \end{bmatrix}$$

$x^{(1)}$ の解をガウスの消去などの方法で計算すると仮定します。結果の精度を「反復的改善」の方法で上げることができます。

最初の計算を行います。

$$\xi = Ax^{(1)} - b \quad (12)$$

次に系を計算します。

$$Ay = \xi \quad (13)$$

$x^{(1)}$ が正確な計算である場合、 ξ は y と同様にゼロ・ベクタになります。一般に、 ξ と y の計算は丸め誤差を招きやすいので、 $Ay \approx \xi \approx Ax^{(1)} - b = A(x^{(1)} - x)$ となります (ただし x は真の解 (未知) です)。 $y \approx x^{(1)} - x$ となるので、解の概算は次のようになります。

$$x^{(2)} = x^{(1)} - y \quad (14)$$

(12)、(13)、(14) のステップをそれぞれ $x^{(1)}$ を $x^{(2)}$ に、また $x^{(2)}$ を $x^{(3)}$ に置き換えて繰り返すことができます。 $x^{(i+1)}$ が $x^{(i)}$ より正確であるという指摘は正式なものではありません。詳細については、Golub/Van Loan (1989) を参照してください。

反復的改善を実行すると、 ξ は、近似値ではあるものの正確には一致しない浮動小数点数の差に相当する要素を含むベクタとなるので、悪性の相殺の問題が出てきます。したがって、反復的改善は、 $\xi = Ax^{(1)} - b$ を倍精度で計算しない限り、それほど意味がありません。これも、完全な倍精度の結果が必要とされるところで、2 つの単精度数 (A と $x^{(1)}$) の積を計算するケースに相当します。

要約すると、2つの浮動小数点数を乗算し、オペランドの2倍の精度で積を返す命令は、浮動小数点命令セットに追加することができます。このことがコンパイラに与える影響について次に説明します。

言語とコンパイラ

コンパイラと浮動小数点の相互関係については、Farnum (1988) の資料に詳しく説明されています。本項の大部分は、この資料から抜粋したものです。

あいまいさ

言語の定義によって、プログラムに関する文が証明されるように言語の意味を正確に規定できるのが理想的です。このことは、言語の整数部分についてはあてはまるものの、浮動小数点に関する限り、言語の定義自体があいまいであることも事実です。おそらく、言語の設計者の意識の中に、浮動小数点には丸め誤差が付きものであるため、証明できるものは何もないという思い込みがあるように思われます。仮にこれが本当であるなら、これまでの説明は、この論拠の誤った認識について実証してきたこととなります。本項では、言語定義に共通して見られるあいまいさについて、その対処方法などを中心に説明します。

驚くべきことに、言語によっては x が浮動小数点の変数 (たとえば $3.0/10.0$) である場合、 $10.0 * x$ が現れるたびにすべて同じ値になるという明確な指定のないものがよくあります。たとえば、Brown のモデルにもとづく Ada では、浮動小数点演算は Brown の原理だけを満たしていれば十分なので、式が多数の値を持っていてもかまわないことになります。浮動小数点をあいまいな形で捉えるという点は、浮動小数点の演算ごとに結果を正確に定義する必要のある IEEE モデルとはまったく異なるところです。IEEE モデルでは、 $(3.0/10.0) * 10.0$ が 3 になることを証明 (定理 7) できますが、Brown のモデルではこれを証明できません。

大部分の言語定義に見られるもう 1 つのあいまいさは、オーバーフロー、アンダーフロー、その他の例外が起きたときの処理方法に関するものです。IEEE 標準では、例外の動作が正確に指定されるので、この標準をモデルとして使用した言語では、この点に関するあいまいさの問題を回避できます。

かつこの解釈の仕方もあいまいです。浮動小数点数は丸め誤差を伴うため、代数の結合法則は必ずしも浮動小数点数には適用できません。たとえば、 $x = 10^{30}$ 、 $y = -10^{30}$ 、 $z = 1$ とした場合の $(x+y)+z$ と $x+(y+z)$ の解はまったく異なります。前者の場合は 1 で、後者は 0 になります。かつこを残すことの重要性は決して強調すぎることはありません。定理 3、4 および 6 に示したアルゴリズムはすべて、かつこの位置に依

存します。たとえば、定理 6 の公式 $x_h = mx - (mx - x)$ は、もしかっこのがなければ $x_h = x$ になり、アルゴリズム全体が無意味になります。かっこの意味を尊重する必要のない言語の定義は、浮動小数点の計算にはほとんど無意味です。

部分式の評価が正確に定義されない言語も数多くあります。仮に `ds` が倍精度、`x` と `y` が単精度であると仮定します。`ds + x*y` という式で、積は単精度と倍精度のどちらで計算されるのでしょうか。また別の例で、`x + m/n` (ここで `m` と `n` はともに整数) の式は整数と浮動小数点数のいずれで計算されるのでしょうか。この問題には 2 つの方法で対処できますが、いずれも完全な解決策にはなりません。1 つは、式の中の変数をすべて同じタイプとして扱う方法です。これは簡単な解決方法ですが、欠点もいくつかあります。まず、Pascal のような言語では、部分範囲のタイプによって部分範囲変数と整数の変数を併用できるので、単精度変数と倍精度変数の混用を禁止してもほとんど意味がありません。また、定数に関する問題もあります。`0.1*x` という式の場合、ほとんどの言語では `0.1` が単精度の定数と解釈されます。ここで仮に、プログラマが浮動小数点変数の宣言をすべて、単精度から倍精度に変更しようと考えたとします。`0.1` をそのまま単精度定数として扱うと、コンパイル時にエラーになります。この場合、プログラマは浮動小数点定数をすべて探して、これを変更する必要があります。

2 つ目のアプローチは、式の混用を認めることにより、部分式の評価に関するルールを提供する方法です。これには参考になる例がいくつかあります。C 言語本来の定義では、浮動小数点式はすべて倍精度で計算しなければなりません

(Kernighan/Ritchie, 1978)。この条件は、本項の最初に紹介した例のような偏差をもたらす原因となります。`3.0/7.0` は倍精度で計算されますが、`q` が単精度変数の場合、商は単精度に丸めて格納されます。`3/7` は 2 進の循環小数になるので、倍精度による計算値は、単精度で格納された値とは一致しくなくなります。したがって、 $q = 3/7$ の計算は失敗します。これは、すべての式をもっとも高い精度で計算しても必ずしも効果的ではないことを示すものです。

もう 1 つの例は内積に関するものです。内積に多数の項が含まれる場合、和の丸め誤差が大きくなる可能性があります。この丸め誤差を減少させる方法として、倍精度の和を累積することができます (この方法については、219 ページの「最適化プログラム」で詳しく説明しています)。`d` が倍精度変数で、`x[i]` と `y[i]` が単精度配列の場合、内積のループは `d = d + x[i] * y[i]` のようになります。単精度で乗算を行うと、倍精度変数に追加する直前に積が単精度に切り捨てられるので、倍精度による累積の利点が得られません。

前の 2 つの例に適用されるルールは、ある式のあらゆる変数中でもっとも高い精度で計算するというものです。そうすると、`q = 3.0/7.0` 全体が単精度で計算され¹、真の論理値が与られます。一方、`d = d + x[i] * y[i]` は倍精度で計算されるので、倍精度 1. ここでは、`3.0` が単精度の定数、`3.0D0` が倍精度の定数であることを前提とします。

度の累積による利点が全面的に生かされます。ただし、このルールはあらゆるケースにそのまま適用できるわけではありません。dx と dy が倍精度変数の場合、 $y = x + \text{single}(dx-dy)$ の式には倍精度の変数が含まれますが、オペランドが両方とも単精度であることに加え、結果も単精度になるので、倍精度で和を求めても意味がありません。

さらに複雑な部分式に関する評価ルールは次のとおりです。まず、それぞれの操作に対し、仮の精度として対応するオペランドのもっとも高い精度を割り当てます。この割り当ては、式のツリー構造で葉の部分から根の方向に行います。次に、根から葉の方向に二次パスを設定します。このパスでは、各操作に最高の精度と、親から要求される精度を仮に割り当てます。 $q = 3.0/7.0$ の場合、葉の部分は単精度になるので、操作はすべて単精度で実行されます。 $d = d + x[i] * y[i]$ の場合は、乗算の一時的な精度が単精度になりますが、二次パスでは親の操作から倍精度オペランドが要求されるので倍精度に昇格します。 $y = x + \text{single}(dx-dy)$ では、加算が単精度で実行されます。Farnum (1988) の資料では、このアルゴリズムのインプリメントがそれほど難しくないことが実証されています。

このルールの欠点は、部分式の評価が、中に組み込まれた式に依存する点にあります。これは、複雑なシーケンスの原因になることがあります。たとえば、プログラムのデバッグ作業を行なっているときに、部分式の値を確認したい場合があります。この場合、プログラム中の部分式は、これが組み込まれた式に依存するので、単にデバッガに部分式を入力して評価するように指示することはできません。最後に、部分式には次の問題があります。10 進数定数から 2 進数定数への変換も 1 つの演算となるので、評価ルールも 10 進数定数の解釈に影響を与えるということです。これは 0.1 のように 2 進数では正確に表現できない定数の場合に特に重要になります。

その他、組み込まれている動作の 1 つとして言語に指数が含まれている場合にも、あいまいさが現れます。基本的な四則演算の場合とは異なり、指数の値は必ずしも明確には表現されない場合があります (Kahan/Coonen, 1982)。** が指数関数の演算子である場合、 $(-3) ** 3$ の値は -27 となります。ただし $(-3.0) ** 3.0$ などの場合は問題になります。** 演算子が整数のべき乗をチェックすると、 $(-3.0) ** 3.0$ を $-3.0^3 = -27$ として計算します。一方、 $x^y = e^{y \log x}$ の公式を使用して実際の引数に ** を定義すると、log 関数によって結果が NaN ($x < 0$ のとき $\log(x) = \text{NaN}$ という定義にもとづく) になることがあります。これに FORTRAN の CLOG 関数を使用すると、ANSI FORTRAN の標準により、 $\text{CLOG}(-3.0)$ が $i\pi + \log 3$ (ANSI 1978) と定義されるので、解は -27 になります。Ada 言語では、整数のべき乗に限って指数を定義することにより、この問題を回避しています。一方、ANSI FORTRAN では、負数を実際の乗数でべき乗することはできません。

FORTRANの標準では、次のように規定されます。

「結果を数学的に定義できない算術演算は禁止される」

しかし、IEEE 標準に $\pm \infty$ という概念が導入されたことにより、「数学的に定義できない」という表現はまったく意味をなさなくなりました。1つの定義として、201ページの「無限大」に示した方法を適用できます。たとえば、 a^b の値を決定する場合に、 $x \rightarrow 0$ に従ってそれぞれ $f(x) \rightarrow a$ 、 $g(x) \rightarrow b$ という特性を持つ非定数の解析関数 f と g の例を考えてみます。 $f(x)^{g(x)}$ はつねに同じ極限に近づくとすれば、これが a^b の値となります。この定義により、 $2^\infty = \infty$ が成立します。 1.0^∞ の場合、 $f(x) = 1$ 、 $g(x) = 1/x$ のときに、限界は 1 に近づきますが、 $f(x) = 1 - x$ 、 $g(x) = 1/x$ のときは限界が e^{-1} となります。したがって、 1.0^∞ は NaN となります。また 0^0 の場合は、 $f(x)^{g(x)} = e^{g(x) \log f(x)}$ となります。 f と g は解析関数なので、 0 、 $f(x) = a_1 x^1 + a_2 x^2 + \dots$ 、および $g(x) = b_1 x^1 + b_2 x^2 + \dots$ のとき値 0 を取ります。したがって、 $\lim_{x \rightarrow 0} g(x) \log f(x) = \lim_{x \rightarrow 0} x \log (x(a_1 + a_2 x + \dots)) = \lim_{x \rightarrow 0} x \log(a_1 x) = 0$ になります。さらにすべての f と g に対して、 $f(x)g(x) \rightarrow e^0 = 1$ となります。すなわち $0^0 = 1$ になります^{1 2}。この定義を使用すると、すべての引数に対する指数関数の定義があいまいになります。特に (-3.0) $** 3.0$ は -27 と定義されることになります。

IEEE 標準

先の191ページの「IEEE 標準」では、IEEE 標準の主な特長について説明しました。しかし IEEE 標準では、これらの特長をプログラミング言語から利用できるかについては何も保証されていません。したがって、この標準をサポートする浮動小数点ハードウェアと、C、Pascal、FORTRAN などのプログラミング言語の間に不一致が生じるのはよくあることです。IEEE の機能によっては、サブルーチン・ライブラリの呼び出しによってアクセスできるものもあります。たとえば、IEEE 標準には、平方根は正確に丸めなければならないという条件があります。これに対応して、平方根関数がハードウェア上で直接インプリメントされていることがよくあります。この機能には、ライブラリにある平方根ルーチンを使用して簡単にアクセスすることができます。しかし、標準の特性によっては、サブルーチンでは簡単にインプリメントできないものもあります。たとえば、コンピュータ言語ではほとんどの場合、浮動小数点のタイプは 2 種類程度しかサポートされません。一方、IEEE 標準では、4 種類の精度が用意されています (このうち、推奨されるのは単精度に拡張を加えた拡張単精度フォーマット、単精度、倍精度、拡張倍精度フォーマットです)。これ以外に、無限大 $1.0^0 = 1$ という判断は、 f が非定数であることを規定する制限に依存します。この制限がなければ、 f を一意に 0 とすることにより、関数は、 $\lim_{x \rightarrow 0} f(x)^{g(x)}$ の値として 0 を返します。したがって、 0^0 は NaN と定義されることになります。

2. 0^0 については、さまざまな議論の余地がありますが、もっとも説得力のある説明は "Concrete Mathematics" (Graham, Knuth, Patashnik 共著) で、二項定理が成り立つには $0^0 = 1$ でなければならないと指摘しています。

の扱いという問題もあります。± ∞ を表わす定数は、サブルーチンから供給できません。ただし、一定の変数のイニシャライザとして定数式が要求される状況では、この定数が使用できなくなります。

さらに微妙な状況として、丸めモード、トラップ・イネーブル・ビット、トラップハンドラ、および例外フラグで構成される状態で計算を伴う操作があげられます。1つのアプローチは、状態を読み取りまたは書き込みするためのサブルーチンを用意することです。さらに、新しい値を自動的にセットし、もとの値を返す操作を1回の操作で処理できる呼び出しも便利な場合があります。210 ページの「フラグ」に説明したとおり、IEEE の状態を修正するための一般的なパターンは、ブロック、またはサブルーチンの範囲に限って、その状態を変更することです。これにより、プログラマ側には、ブロックの出口を逐一、探し出す負担と、状態が復元されたことを確認する作業が増えることになります。この場合、状態をブロックの範囲内に正しく設定できる言語は非常に便利です。Modeula-3 は、トラップハンドラにこの概念を適用した言語です (Nelson, 1991)。

言語の中で IEEE 標準をインプリメントするときには、考慮すべき点がいくつかあります。あらゆる x について $x - x = +0$ となるので¹、 $(+0) - (+0) = +0$ となります。ただし、 $-(+0) = -0$ となるので、 $-x$ は $0 - x$ として定義することはできません。NaN は、別の NaN の場合も含め、同じ値には決してならないので、 $x = x$ が必ずしも真にはならないという理由から、NaN を導入することは混乱の原因になります。IEEE で `Isnan` 関数を使用しないように推奨されている場合は、実際に、 $x \neq x$ の式は NaN をテストするもっとも簡単な方法です。また NaN はほかの数との順序に関連性がないので、 $x \leq y$ を非 $x > y$ として定義することもできません。NaN を導入すると、浮動小数点数が部分的に順序付けられるので、 $<$ 、 $=$ 、 $>$ 、または `unordered` のいずれかを返す `compare` 関数により、プログラマは比較処理を簡単に実行できるようになります。

IEEE 標準では、いずれかのオペランドが NaN の場合に、NaN を返す基本的な浮動小数点操作を定義してありますが、複合操作の場合には、この定義が必ずしも有効になるとは限りません。グラフをプロットするときに使用する倍率を求める場合、一連の値の最大値を計算することができます。この場合、最大値の操作で単に NaN を無視すれば、意味があります。

最後に、丸めの操作が問題になることがあります。IEEE 標準では、丸め操作が厳密に定義されており、これは丸めモードの現在の値に依存します。丸めモードは、型変換における暗黙の丸めや、言語中の明示的な `round` 関数と競合することがあります。すなわち IEEE の丸めを使用するプログラムでは、自然言語のプリミティブを使用できず、一方、言語のプリミティブは、しだいに普及の広がる IEEE マシンでインプリメントするには不十分ということになります。

1. 丸めモードが $-\infty$ への丸めになっている場合は除きます (この場合は $x - x = -0$ です)。

最適化プログラム

コンパイラに関する解説書は、浮動小数点に関する問題を無視する傾向があるようです。たとえば、Ahoら (1986) は、 $x/2.0$ を $x*0.5$ に置き換えることにより、読者は $x/10.0$ も $0.1*x$ に置き換えるものと解釈してしまいます。しかし、これらの式は、 0.1 を 2 進数で正確に表現することはできないので、2 進数マシンでは同じセマンティクスにはなりません。本書では、 $x*y - x*z$ を $x*(y-z)$ と置き換えることをお勧めします。これは、 $y \approx z$ のときに、仮に 2 つの式がまったく異なる値を持っている場合でもあてはまります。最適化プログラムが言語の定義に違反してはならないという条件は、コードを最適化するときには、どのような代数単位元でも使用できるという記述を確かに正当化するものですが、その反面、浮動小数点のセマンティクスはそれほど重要ではないという印象も与えます。言語の標準でかつこの位置を尊重すべきかに関する指定の有無を問わず、すでに説明したとおり $(x+y) + z$ と $x + (y+z)$ の解がまったく異なることも考えられます。たとえば、かつこの保持に深くかかわる問題は、次のコードに見ることができます。

```
eps = 1;
do eps = 0.5*eps; while (eps + 1 > 1);
```

このコードは、マシン・イプシロンに関する概算を求めるものです。最適化コンパイラが $eps + 1 > 1 \Leftrightarrow eps > 0$ であることを認識すると、プログラムはまったく異なる動作を示すこととなります。 $1 \oplus x$ が x ($x \approx e \approx \beta \cdot \mathbf{P}$) を超えることがないように、最小の x を計算する代わりに、 $x/2$ を 0 ($x \approx \beta^{e_{\min}}$) 方向に丸めを行う最大の x を計算します。この種の「最適化」を避けることは、最適化によって失われる便利なアルゴリズムを保持するという意味で重要になります。

積分、微分方程式の数値解といった数多くの問題は、多数の項を伴う和の計算を行うものです。加算操作を行うごとに、 0.5 ulp もの誤差が介入する可能性があるため、数千もの項を伴う和の丸め誤差は非常に大きくなる場合があります。この問題を解決する簡単な方法は、部分的な加数を倍精度変数として格納し、各加算処理をそれぞれ倍精度で実行することです。計算を単精度で実行して、和を倍精度で実行することは、ほとんどのコンピュータシステムでサポートされています。ただし、計算がすでに倍精度で行われている場合に、精度を倍加するのはそれほど簡単ではありません。よく提案される 1 つの解決案として、各数値をソートして、小さい順に加算していく方法があります。しかし、和の精度を大幅に改善できる、はるかに効率的な方法があります。すなわち、次の方法です。

定理 8 (Kahan の加法公式)

次のアルゴリズムに従って、 $\sum_{j=1}^N x_j$ を計算するものとする。

```
S = X[1];  
C = 0;  
for j = 2 to N {  
    Y = X[j] - C;  
    T = S + Y;  
    C = (T - S) - Y;  
    S = T;  
}
```

S の計算値は、 $\sum x_j(1 + \delta_j) + O(N\epsilon^2)\sum |x_j|$ (ここで $(|\delta_j| \leq 2\epsilon)$) になる。

固有の公式 $\sum x_j$ を使用して和を計算すると、 $\sum x_j(1 + \delta_j)$ (ただし $|\delta_j| < (n - j)\epsilon$) になります。この結果は、Kahan の加法公式の場合に比べると著しく改善されています。この単純な公式の中で、各加数は、 ne もの摂動の影響は受けず、単に $2e$ で摂動されるだけです。詳細については、236 ページの「加法の誤差」を参照してください。

浮動小数点演算が、代数法則に従っていることを前提とする最適化プログラムでは、結果が $C = [T - S] - Y = [(S + Y) - S] - Y = 0$ となり、アルゴリズムがまったく無意味になります。これらの例により、実数に適用される代数の単位元を浮動小数点変数を伴う式に使用する場合は、最適化プログラムを慎重に使用する必要があるということが言えます。

さらに、最適化プログラムは、定数を扱う場合にも浮動小数点コードのセマンティクスを変える可能性があります。1.0E-40*x の式では、暗黙のうちに 10 進数から 2 進数定数への変換が行われます。この定数は、2 進数で正確に表現することはできないので、不正確な操作の例外が起きます。また、式が単精度で評価された場合は、アンダーフロー・フラグがセットされます。定数が不正確である場合、2 進数への正確な変換は、IEEE 丸めモードの現在の値に依存します。したがって、コンパイル時に 1.0E-40 を 2 進数に変換する最適化プログラムにより、プログラムのセマンティクスが変更されることになります。ただし、27.5 のような定数は、最低の精度でも正確に表現できるので、コンパイル時に正しく変換されます。すなわち、つねに正確に表現されるので、例外も起きず、丸めモードの影響も受けないということです。コンパイル時に変換すべき定数には、たとえば `const pi = 3.14159265` のような定数宣言を使用する必要があります。

共通項の消去の場合も、最適化プログラムによって浮動小数点のセマンティクスを変えられる可能性があります。たとえば、次のようなコードです。

```
C = A*B;  
RndMode = Up  
D = A*B;
```

$A*B$ は一見、共通項のように思われますが、2 か所の評価位置でそれぞれ丸めモードが異なるので共通項としては扱いません。次の 3 つの例が考えられます。まず $x = x$ は、論理定数 `true` で置き換えることはできません。これは、NaN が通常の浮動小数点数に比べて大小で表わすことはできないので、 x が NaN のときに成立しない、 $x = +0$ の場合に $-x = 0 - x$ が成立しない、また $x < y$ は $x \geq y$ の逆ではない、という理由があるからです。

以上の例がある反面、浮動小数点コードに対して効果のある最適化もあります。まず、浮動小数点数に有効となる代数の単位元があります。IEEE 演算の例として $x + y = y + x$ 、 $2 \times x = x + x$ 、 $1 \times x = x$ 、 $0.5 \times x = x/2$ などがあります。ただし、これらの単純な単位元は、CDC や Cray のスーパーコンピュータ上では失敗します。そのほか、命令のスケジューリングやインライン・プロシージャ置換も、最適化によって効果の得られる例となります¹。

3 つ目の例として、 $dx = x*y$ (x と y はともに単精度の変数、 dx は倍精度を表わします) の式を考えてみます。2 つの単精度数を乗算して倍精度数を出力するマシンでは、 $dx = x*y$ はそれぞれ該当する命令にマップされます。つまりオペランドを倍精度に変換してから、倍精度の乗算をするという一連の命令へのコンパイルは行われません。

コンパイラ的设计者の中に、 $(x + y) + z$ から $x + (y + z)$ への変換を禁止する制限は、移植不能なトリックを使用するプログラマにしか関連のないことなので、無意味であるという指摘もあります。おそらく、これは浮動小数点数が実数のモデルに従っているので、実数と同じ法則に従うべきであるという論拠にもとづくものです。実数のセマンティクスに伴う問題は、インプリメントするのにコストがかかるという点です。2 つの n ビット数を乗算すると、積は $2n$ ビットになります。

間隔の広い指数を伴う 2 つの n ビット数を加算するごとに、和のビット数は $n +$ 指数間の間隔で表わされます。和は $(e^{\max} - e^{\min}) + n$ ビットとなり、およそ $2 \cdot e^{\max} + n$ ビットに相当します。何千回もの演算を行うアルゴリズム (たとえば、線形代数の解など) では、多数の有意ビットを含む数を対象として操作を行うので、処理がきわめて低速になります。また、 $\sin$ や $\cos$ などの超越関数の値は有理数にはならないので、これら

1. VAX の VMS 数学ライブラリでは、低速な CALLS や CALLG 命令の代わりに、該当するサブルーチンをコストのかからない形式で呼び出すという意味で、一種のインライン・プロシージャ置換を使用しています。

のライブラリ関数をインプリメントするのはさらに難しくなります。LISP システムでは、正確な整数の演算が提供されていて、問題を解決しやすい場合があります。ただし、正確な浮動小数点演算はほとんど役に立ちません。

実際には、 $(x + y) + z \neq x + (y + z)$ という事実を利用した便利なアルゴリズム (Kahan の加法公式など) が用意されており、次の範囲が有効である限り、正しく動作します。

$$a \oplus b = (a + b)(1 + \delta)$$

この範囲は、市販されている大部分のハードウェアに適用されるので、数値プログラマがこうしたアルゴリズムを無視したり、コンパイラ設計者が浮動小数点変数に実数のセマンティクスが保持されることを前提として、これらのアルゴリズムを利用しないことは、賢明とは言えません。

例外処理

ここまでの説明は主に、システムが持つ精度の意味に関するものでした。トラップハンドラも、システムに関する興味深い問題を提起するものです。IEEE 標準では、ユーザーが 5 つのクラスの各例外に対するトラップハンドラを指定できるような仕様が強く推奨されています。208 ページの「トラップハンドラ」では、ユーザーが定義できるトラップハンドラのアプリケーションをいくつか紹介しました。無効な操作やゼロによる除算が行われた場合、ハンドラには該当するオペランド、あるいは正確に丸めた結果を提供する必要があります。使用するプログラミング言語によってトラップハンドラは、プログラム中のその他の変数にもアクセスできる場合があります。すべての例外について、トラップハンドラは、実行された操作の内容、および意図された精度を識別できなければなりません。

IEEE 標準では、各操作が概念的にシリアル形式で構成され、何らかの例外が起きた時点で操作の内容とオペランドを識別できることを前提としています。パイプライン機能や多重演算ユニットをサポートしたマシンでは、例外が起きた場合に、単にトラップハンドラでプログラムカウンタをチェックさせるだけでは不十分になる場合もあります。すなわち操作の内容を正確に識別できるハードウェアが必要になるということです。

次のプログラム部分は、また別の問題を示したものです。

```
x = y*z;  
z = x*w;  
a = b + c;  
d = a/x;
```

2 つ目の乗算により例外が起きて、トラップハンドラが a の値を使用したいものと仮定します。加算と乗算を並列的に実行できるハードウェアでは、最適化プログラムによって、2 つ目の乗算を始める前に加算操作が移動されるので、この加算処理は最初の乗算と並行して実行することができます。したがって、2 つ目の乗算がトラップされても、 $a = b + c$ の実行が終わっているので、 a の結果が一部変更されている可能性があります。あらゆる浮動小数点演算がトラップされる可能性があり、この結果、最適化をスケジュールする命令がすべて消去されることがあるため、コンパイラで、以上のような最適化を避けることはよい方法とは言えません。この問題は、トラップハンドラがプログラム中の変数に直接アクセスするのをすべて禁止することによって解決できます。代わりに、ハンドラには引数として、オペランドまたは結果を渡すことができます。

これでもなお、別の問題が残ります。次のコード部分の場合、2 つの命令が並列的に実行される可能性があります。

```
x = y*z;  
z = a + b;
```

乗算がトラップされると、その引数 z がすでに次の加算によって上書きされている可能性があります。特に、加算は乗算よりも高速で処理されるので、これがあてはまります。IEEE 標準をサポートするコンピュータシステムでは、 z の値をハードウェアに保存するか、あるいは最初の時点でコンパイラにこのような状況を回避させるか、といった何らかの手段が必要になります。

Kahan は、トラップハンドラの代わりに、「前置換」という方法によってこうした問題を回避できると提案しています。この方法では、ある例外が起きた時点で、その結果として使用する値と例外の内容をユーザー自身が指定します。たとえば、 $(\sin x)/x$ を計算するコードで、ユーザーが $x = 0$ となることはほとんどないと判断した上で、 $x = 0$ のテストを省略し、代わりに $0/0$ トラップが起きた時点で、このケースに対応させることによってパフォーマンスを改善したいとします。IEEE トラップハンドラを使用する場合、ユーザーは、 $\sin x/x$ の計算を始める前に、まず値 1 を返すハンドラを作成し、これをインストールしなければなりません。一方、前置換を利用すれば、無効

な操作が起きた時点で、値 1 を使用するように指定するだけで済みます (例外が実際に起きる前に、使用する値を指定しておく必要があるからです)。トラップハンドラの場合、返される値はトラップが起きた時点で計算することができます。

前置換の利点は、ハードウェアの実装がわかりやすくなるという点です¹。例外のタイプが特定された時点で、これをもとに、必要な操作の結果を保存したテーブルを参照させることができます。前置換にはいくつかの利点がある一方、IEEE の標準がすでに広く普及していることが、ハードウェアメーカーによる受け入れを阻んでいるという問題があります。

詳細

ここまで、浮動小数点演算に関する主な特性について、さまざまな観点から検討してきました。この後の項では、浮動小数点が決して難題ではなく、これまでに指摘されてきた問題はすべて数学的に検証できるわかりやすい問題であるという点を実証していきます。この項は大きく 3 つに分けることができます。第 1 部では、誤差の分析に関する導入説明を行います。また 177 ページの「丸め誤差」に示した「丸め誤差」について詳しく説明します。第 2 部では、2 進数から 10 進数への変換について、191 ページの「IEEE 標準」に関する補足説明を行います。第 3 部では、211 ページの「システムの側面」で例示した「Kahan の加法公式」について詳しく説明します。

丸め誤差

「丸め誤差」の項では、保護桁を 1 桁確保すれば、加算と減算は必ず正確に実行されると説明しました (定理 2)。ここでは、その検証を行います。定理 2 は、加算に関するものと、減算に関するものの 2 つの部分で構成されます。減算に関する定理は次のとおりです。

定理 9

x と y が、パラメータ β と p を伴うフォーマットで正の浮動小数点数を表わす場合、 $p + 1$ 桁 (1 桁は保護桁) で減算を行うと、結果の相対丸め誤差は

$$\left(\frac{\beta}{2} + 1\right)\beta^{-p} = \left(1 + \frac{2}{\beta}\right) e \leq 2e \text{ 以下になる。}$$

1. 前置換の難しさは、ハードウェアによる直接的な実装が必要になる点、またソフトウェア上でインプリメントする場合は、連続的な浮動小数点のトラップを設定するところにあります。-Ed

証明

$x > y$ になるように、必要に応じて x と y を入れ換えます。 x が $x_0.x_1 \dots x_{p-1} \times \beta^0$ になるように x と y を基準化 (スケール) してもかまいません。 y が $y_0.y_1 \dots y_{p-1}$ で表わされる場合、その差は正確です。また y が $0.y_1 \dots y_p$ の場合、保護桁によって計算値の差は、丸め誤差が最大 e 以内になるように、浮動小数点数に丸めた正確な差を表わすことになります。一般に、 $y = 0.0 \dots 0.y_{k+1} \dots y_{k+p}$ 、および $\bar{y}$ は、 $p+1$ 桁に切り捨てた y となります。これにより、次の式が成り立ちます。

$$y - \bar{y} < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \dots + \beta^{-p-k}) \quad (15)$$

保護桁の定義から、 $x - y$ の計算値は $x - \bar{y}$ を浮動小数点数に丸めた値、すなわち $(x - \bar{y}) + \delta$ (丸め誤差 δ は次の式を満たします) になります。

$$|\delta| \leq (\beta/2)\beta^{-p} \quad (16)$$

正確な差は、 $x - y$ なので、誤差は $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y + \delta$ になります。この場合、次の3つのケースが考えられます。 $x - y \geq 1$ の場合、相対誤差は次の範囲になります。

$$\frac{y - \bar{y} + \delta}{1} \leq \beta^{-p} [(\beta - 1)(\beta^{-1} + \dots + \beta^{-k}) + \beta/2] < \beta^{-p}(1 + \beta/2) \quad (17)$$

$x - \bar{y} < 1$ の場合は、 $\delta = 0$ になります。 $x - y$ が取れる最小値は次のとおりです。

$$1.0 - 0.\left(\overbrace{0 \dots 0}^k\right)\left(\overbrace{\rho \dots \rho}^p\right) > (\beta - 1)(\beta^{-1} + \dots + \beta^{-k}), \text{ (ただし } \rho = \beta - 1 \text{)}$$

したがって、この場合の相対誤差は次の範囲になります。

$$\frac{y - \bar{y} + \delta}{(\beta - 1)(\beta^{-1} + \dots + \beta^{-k})} < \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \dots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \dots + \beta^{-k})} = \beta^{-p} \quad (18)$$

最後は、 $x - y < 1$ のときに $x - \bar{y} \geq 1$ となるケースです。これは、 $x - \bar{y} = 1$ の場合 (すなわち $\delta = 0$) に限って起きます。 $\delta = 0$ の場合は、公式 (18) が適用されるので、相対誤差の範囲は $\beta^{-p} < \beta^{-p} (1 + \beta/2)$ となります。■

$\beta = 2$ の場合、範囲は正確に $2e$ となり、この範囲は $p \rightarrow \infty$ とした $x = 1 + 2^{2^{-p}}$ と $y = 2^{1^{-p}} - 2^{1-2^{-p}}$ に対して得られます。同じ符号の数を加算する場合は、次の例に見るように、保護桁がなくても、正確な結果が得られます。

定理 10

$x \geq 0, y \geq 0$ の場合、 $x + y$ を計算するときの相対誤差は、保護桁がない場合でも、最大 2ε に抑えられる。

証明

k の保護桁に対するアルゴリズムは、減算のアルゴリズムに似ています。 $x \geq y$ の場合は、 x と y の基数の位置が合うまで、 y を右にシフトします。 $p + k$ の位置を超えた桁は無視されます。この $p + k$ 桁の 2 つの数の和を求めます。次に p 桁に丸めます。

保護桁を使用しない場合の定理について検証します (一般的なケースは同じです)。 $x \geq y \geq 0$ 、および x を $d.dd... \times \beta^0$ の形式に基準化 (スケール) することを前提とした一般論が失われることはありません。まず、キャリーアウトはないものとします。 y の終わりからシフトオフした桁の値は β^{-p+1} より少なく、和は 1 以上となるので、相対誤差は $\beta^{-p+1}/1 = 2e$ となります。

一方、キャリーアウトを伴う場合、シフトによる誤差は $\frac{1}{2} \beta^{-p+2}$ の丸め誤差に追加する必要があります。

和は β 以上になるので、丸め誤差は

$$\left(\beta^{-p+1} + \frac{1}{2} \beta^{-p+2} \right) / \beta = (1 + \beta/2) \beta^{-p} \leq 2\varepsilon$$

以下に抑えられます。■

これら 2 つの定理を組み合わせることにより、定理 2 を導くことができます。定理 2 は、1 回の操作を実行する場合に伴う相対誤差を求めるものです。 $x^2 - y^2$ と $(x + y)(x - y)$ の丸め誤差を比較する場合、多重操作に伴う相対誤差を考慮しなければなりません。 $x \ominus y$ の相対誤差は、 $\delta_1 = [(x \ominus y) - (x - y)] / (x - y)$ となり、 $|\delta_1| \leq 2e$ の条件を満たします。また、次のように書くこともできます。

$$x \ominus y = (x - y)(1 + \delta_1), \quad |\delta_1| \leq 2e \quad (19)$$

同様に、

$$x \oplus y = (x + y)(1 + \delta_2), \quad |\delta_2| \leq 2e \quad (20)$$

まず正確な積の計算と丸めによって乗算を行うとすると、相対誤差は最大 0.5 ulp となるので、浮動小数点数 u と v には次の式が成り立ちます。

$$u \otimes v = uv(1 + \delta_3), \quad |\delta_3| \leq e \quad (21)$$

上の 3 つの方程式を組み合わせる ($u = x \ominus y$ 、および $v = x \oplus y$) と、次のようになります。

$$(x \ominus y) \otimes (x \oplus y) = (x - y)(1 + \delta_1)(x + y)(1 + \delta_2)(1 + \delta_3) \quad (22)$$

したがって、 $(x - y)(x + y)$ を計算するときに起きる相対誤差は次のようになります。

$$\frac{(x - y) \ominus (x + y) - (x^2 - y^2)}{(x^2 - y^2)} = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) - 1 \quad (23)$$

相対誤差は、 $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3 + \delta_1\delta_2\delta_3$ になり、範囲は $5e + 8e^2$ となります。すなわち最大相対誤差は 5 (丸め誤差) となります (e が最小数で、 e^2 はほぼ無視できます)。

$(x \otimes x) \ominus (y \otimes y)$ を同じように分析しても相対誤差は小さい値にはなりません。これは、 x と y という 2 つの近い値が $x^2 - y^2$ に代入されると、相対誤差は通常、非常に大きくなるためです。この場合、 $(x \ominus y) \otimes (x \oplus y)$ で真になった分析を再び使用して、次の式を得ることができます。

$$\begin{aligned}(x \otimes x) \ominus (y \otimes y) &= [x^2(1 + \delta_1) - y^2(1 + \delta_2)](1 + \delta_3) \\ &= ((x^2 - y^2)(1 + \delta_1) + (\delta_1 - \delta_2)y^2)(1 + \delta_3)\end{aligned}$$

x と y の値が近い場合、誤差の項 $(\delta_1 - \delta_2)y^2$ は、最大 $x^2 - y^2$ と同じくらい大きくなる場合があります。以上の計算によって、 $x^2 - y^2$ より $(x - y)(x + y)$ のほうがより正確であるということが実証されたことになります。

次に、三角形の面積に関する公式を分析します。公式 (7) を計算するときにかかる最大誤差を見積もるには、次の事実が必要になります。

定理 11

保護桁で減算を行うと、 $y/2 \leq x \leq 2y$ の場合に、 $x - y$ は正確に計算される。

証明

x と y に同じ指数がある場合、 $x \ominus y$ は正確に計算されます。これ以外の場合、定理の条件から、指数の差は最大 1 であると言えます。 $0 \leq y \leq x$ になるように、必要に応じて x と y を入れ換えて、 x が $x_0.x_1 \dots x_{p-1}$ 、および y が $0.y_1 \dots y_p$ となるように x と y を基準化 (スケール) してもかまいません。 $x \ominus y$ を計算するアルゴリズムは、まず $x - y$ を正確に計算して、次に浮動小数点に丸めるものです。差が $0.d_1 \dots d_p$ の形式の場合、この差の長さはすでに p 桁になっているので、丸め操作は必要ありません。 $x \leq 2y$ となるので、 $x - y \leq y$ 、また y は $0.d_1 \dots d_p$ となるので、 $x - y$ になります。■

$\beta > 2$ の場合、定理 11 を $y/\beta \leq x \leq \beta y$ に置き換えることはできません。 $y/2 \leq x \leq 2y$ の強い条件が必要になります。定理 10 の証明の直後に示した $(x - y)(x + y)$ の誤差分析は、加算と減算の基本演算中に起きる相対誤差は小さい (定理 (19) と (20)) という事実を前提にしたものです。これは、もっとも一般的な誤差分析の例です。公式 (7) を分析するには、次の証明で示すとおり、定理 11 のような条件が必要になります。

定理 12

保護桁で減算を行う場合、 a 、 b 、 c が三角形の 3 辺 ($a \geq b \geq c$) とすると、 $(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$ を計算するときの相対誤差は最大 16ε (ただし $\varepsilon < 0.005$) になる。

証明

各項を 1 つずつ検証してみます。定理 10 より、 $b \oplus c = (b + c)(1 + \delta_1)$ (ただし δ_1 は相対誤差、 $|\delta_1| \leq 2\varepsilon$) を導くことができます。最初の項の値は $(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1))(1 + \delta_2)$ になり、次のようになります。

$$\begin{aligned} \frac{(a + b + c)(1 - 2\varepsilon)^2}{(1 + 2\varepsilon)} &\leq \frac{[a + (b + c)(1 - 2\varepsilon)](1 - 2\varepsilon)}{(1 + 2\varepsilon)} \leq \frac{[a + (b + c)(1 + 2\varepsilon)]}{(1 + 2\varepsilon)} \\ &\leq (a + b + c)(1 + 2\varepsilon)^2 \end{aligned}$$

すなわち、 η_1 が 1 つなので、次の公式が成り立ちます。

$$(a \oplus (b \oplus c)) = (a + b + c)(1 + \eta_1)^2, \quad |\eta_1| \leq 2\varepsilon \quad (24)$$

次の項には、 $a \ominus b$ に丸め誤差を含むことがあるので、 c と $a \ominus b$ に悪性の減算を伴う可能性があります。 a 、 b 、 c は三角形の 3 辺 ($a \leq b + c$) をなし、この式を $c \leq b \leq a$ の順序と組み合わせると、 $a \leq b + c \leq 2b \leq 2a$ となります。したがって、 $a - b$ は定理 11 の条件を満たします。すなわち $a - b = a \ominus b$ は正しいことになり、 $c \ominus (a - b)$ という減算は無害なので、定理 9 から次のように導くことができます。

$$(c \ominus (a \ominus b)) = (c - (a - b))(1 + \eta_2), \quad |\eta_2| \leq 2\varepsilon \quad (25)$$

3 つ目の項は、2 つの正確な正数の和を計算するものなので、次のようになります。

$$(c \oplus (a \ominus b)) = (c + (a - b))(1 + \eta_3), \quad |\eta_3| \leq 2\varepsilon \quad (26)$$

最後の項は、定理 9 と 10 にもとづいて次のようになります。

$$(a \oplus (b \ominus c)) = (a + (b - c))(1 + \eta_4)^2, \quad |\eta_4| \leq 2\varepsilon \quad (27)$$

$x \otimes y = xy(1 + \xi)$ (ただし $|\xi| \leq \varepsilon$) となるように、乗算を正確に丸めると、公式 (24)、(25)、(26)、(27) の組み合わせは次のようになります。

$$(a \oplus (b \oplus c)) (c \ominus (a \ominus b)) (c \oplus (a \ominus b)) (a \oplus (b \ominus c)) \leq (a + (b + c)) (c - (a - b)) (c + (a - b)) (a + (b - c)) E$$

ただし、

$$E = (1 + \eta_1)^2 (1 + \eta_2) (1 + \eta_3) (1 + \eta_4)^2 (1 + \zeta_1)(1 + \zeta_2) (1 + \zeta_3)$$

E の上限は、 $(1 + 2\varepsilon)^6(1 + \varepsilon)^3$ となり、範囲は $1 + 15\varepsilon + O(\varepsilon^2)$ となります。作成者によっては $O(e^2)$ を単に無視する場合がありますが、これは簡単に説明できます。 $(1 + 2\varepsilon)^6(1 + \varepsilon)^3 = 1 + 15\varepsilon + \varepsilon R(\varepsilon)$ と記述すると、 $R(\varepsilon)$ は正の係数を持つ e の多項式となるので、 ε の漸増関数になります。 $R(0.005) = 0.505$ なので、 $\varepsilon < 0.005$ にはすべて $R(\varepsilon) < 1$ となり、 $E \leq (1 + 2\varepsilon)^6(1 + \varepsilon)^3 < 1 + 16\varepsilon$ が成り立ちます。 E の下限を求める場合、 $1 - 15\varepsilon - \varepsilon R(\varepsilon) < E$ となるので、 $\varepsilon < 0.005$ のとき、 $1 - 16\varepsilon < (1 - 2\varepsilon)^6(1 - \varepsilon)^3$ になります。これら 2 つの範囲を組み合わせると、 $1 - 16\varepsilon < E < 1 + 16\varepsilon$ になります。したがって、相対誤差は最大 16ε です。■

定理 12 は、公式 (7) に悪性の相殺は起きないことを証明するものです。ここで公式 (7) が数値的に安定していることを証明するまでもありませんが、公式全体の有効範囲 (182 ページの「相殺」に示す定理 3 で決まる範囲) を規定しておくのがよい方法です。

定理 3 の証明

次のように仮定します。

$$q = (a + (b + c)) (c - (a - b)) (c + (a - b)) (a + (b - c))$$

および

$$Q = (a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))$$

定理 12 により、 $Q = q(1 + \delta)$ (ただし $\delta \leq 16\varepsilon$) が成り立ちます。 $\delta \leq 0.04 / (0.52)^2 \approx 0.15$ と仮定すれば、次の式は、簡単にチェックできます。

$$1 - 0.52|\delta| \leq \sqrt{1 - |\delta|} \leq \sqrt{1 + |\delta|} \leq 1 + 0.52|\delta| \quad (28)$$

また、 $|\delta| \leq 16\varepsilon \leq 16(0.005) = 0.08$ となるので、 δ が条件を満たすことになります。したがって、 $|\delta_1| \leq 0.52|\delta| \leq 8.5\varepsilon$ の場合、
 $\sqrt{q} = \sqrt{q(1+\delta)} = \sqrt{q(1+\delta_1)}$ となります。

誤差を 0.5 ulp 以内として平方根を計算すると、 $\sqrt{q}$ を計算するときの誤差は、
 $|\delta_2| \leq \varepsilon$ の場合に、 $(1+\delta_1)(1+\delta_2)$ になります。 $\beta=2$ の場合、4 で除算をした場合でもこれ以外の誤差は導入されません。その他の場合、除算にはもう 1 つの因数 $1+\delta_3$ ($|\delta_3| \leq \varepsilon$) が必要で、また、定理 12 の証明に示した方法によって、 $(1+\delta_1)(1+\delta_2)(1+\delta_3)$ の最終的な誤差の範囲は $1+\delta_4$ に依存します (ただし $|\delta_4| \leq 11\varepsilon$)。■

定理 4 の直後に示した帰納的な説明をさらに正確に表現できるように、次の定理は $\mu(x)$ がどれほど定数に接近するかを示したものです。

定理 13

$\mu(x) = \ln(1+x)/x$ とすると、 $0 \leq x \leq \frac{3}{4}$ の場合、 $\frac{1}{2} \leq \mu(x) \leq 1$ になり、この微分は $|\mu'(x)| \leq \frac{1}{2}$ を満たすことになる。

証明

$\mu(x) = 1 - x/2 + x^2/3 - \dots$ は下降項の交代級数なので、 $x \leq 1$ の場合、 $\mu(x) \geq 1 - x/2 \geq \frac{1}{2}$ となります。 μ の級数は入れ代わって $\mu(x) \leq 1$ となることは簡単に確認できます。また $\mu'(x)$ のテイラー級数も入れ代わります。

$x \leq 3/4$ は下降項なので、 $-\frac{1}{2} \leq \mu'(x) \leq -\frac{1}{2} + 2x/3$ 、または $-\frac{1}{2} \leq \mu'(x) \leq 0$ となるので、 $|\mu'(x)| \leq \frac{1}{2}$ となります。■

定理 4 の証明

$\ln$ に対するテイラー級数、

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$$

は交代級数、 $0 < x \cdot \ln(1+x) < x^2/2$ なので、 $\ln(1+x)$ を x で近似させるときの相対誤差は $x/2$ により範囲が決まります。 $1 \oplus x = 1$ の場合、 $|x| < \varepsilon$ となるので、相対誤差の範囲は $\varepsilon/2$ となります。

$1 \oplus x \neq 1$ の場合、 $1 \oplus x = 1 + \tilde{x}$ によって $\tilde{x}$ を定義します。 $0 \leq x < 1$ のとき、 $(1 \oplus x) \ominus 1 = \tilde{x}$ です。除算と対数を $\frac{1}{2} \text{ulp}$ 以内の範囲で計算した場合、 $\ln(1+x)/((1+x) - 1)$ の式の計算値は次のようになります。

$$\frac{\ln(1 \oplus x)}{1 \oplus x - 1} (1 + \delta_1) (1 + \delta_2) = \frac{\ln(1 + \tilde{x})}{\tilde{x}} (1 + \delta_1) (1 + \delta_2) = \mu(\tilde{x}) (1 + \delta_1) (1 + \delta_2) \quad (29)$$

ここで、 $|\delta_1| \leq \varepsilon$ 、 $|\delta_2| \leq \varepsilon$ です。 $\mu(\tilde{x})$ の概算を求めるには、 x から $\tilde{x}$ の間の特定の ξ に対して、次の平均値の定理を使用します。

$$\mu(\tilde{x}) - \mu(x) = (\tilde{x} - x) \mu'(\xi) \quad (30)$$

$\tilde{x}$ の定義から、 $|\tilde{x} - x| \leq \varepsilon$ となり、これを定理 13 と組み合わせると、 $|\mu(\tilde{x}) - \mu(x)| \leq \varepsilon/2$ 、または $|\mu(\tilde{x})/\mu(x) - 1| \leq \varepsilon/(2|\mu(x)|) \leq \varepsilon$ となります。すなわち $|\delta_3| \leq \varepsilon$ の場合に $\mu(\tilde{x}) = \mu(x)(1 + \delta_3)$ となります。最後に x で乗算すると、最終的に δ_4 が導入されるので、 $x \cdot \ln(1 \oplus x)/((1 \oplus x) \ominus 1)$ は次のようになります。

$$\frac{x \cdot \ln(1+x)}{(1+x)-1} (1 + \delta_1) (1 + \delta_2) (1 + \delta_3) (1 + \delta_4), \quad |\delta_1| \leq \varepsilon$$

$\varepsilon < 0.1$ であれば、 $(1 + \delta_1) (1 + \delta_2) (1 + \delta_3) (1 + \delta_4) = 1 + \delta$ (ただし $|\delta| \leq 5\varepsilon$ の場合) となります。■

公式 (19)、(20)、(21) を使用した誤差分析の興味深い例は、二次方程式の根の公式 $(-b \pm \sqrt{b^2 - 4ac})/2a$ に見ることができます。182 ページの「相殺」で、方程式を書き換えることにより、土の操作が原因で起きる可能性のある相殺をいかに除去できるかを示しました。しかし、 $d = b^2 - 4ac$ の計算時に起きる可能性のある相殺がもう 1 つあります。この相殺は、単に公式を入れ換えるだけでは取り除くことはできません。簡単に言うと、 $b^2 \approx 4ac$ のとき、丸め誤差により、二次方程式の根の公式で計算する根の半分の桁まで汚染される可能性があります。ここで、略式の証明を示します (二次方程式の根の公式の誤差を除去するための別のアプローチは Kahan (1972) の資料に紹介されています)。

$b^2 \approx 4ac$ の場合、丸め誤差により、二次方程式の根の公式 $(-b \pm \sqrt{b^2 - 4ac})/2a$ で計算する根の桁の半分まで汚染される可能性がある。

証明：

$(b \otimes b) \ominus (4a \otimes c) = (b^2(1 + \delta_1) - 4ac(1 + \delta_2))(1 + \delta_3)$ (ただし $|\delta_i| \leq \varepsilon$) となります。¹

$d = b^2 - 4ac$ を使用すると、 $(d(1 + \delta_1) - 4ac(\delta_2 - \delta_1))(1 + \delta_3)$ のように再書き込みされます。

この誤差のサイズを見積もるには、 δ_i の第 2 項を無視します。この場合の絶対誤差は $d(\delta_1 + \delta_3) - 4ac\delta_4$ となります (ただし $|\delta_4| = |\delta_1 - \delta_2| \leq 2\varepsilon$ です)。 $d \ll 4ac$ であるため、第 1 項 $d(\delta_1 + \delta_3)$ は無視できます。第 2 項を見積もる場合は、 $ax^2 + bx + c = a(x - r_1)(x - r_2)$ という事実にもとづき、 $ar_1r_2 = c$ となります。 $b^2 \approx 4ac$ の場合、 $r_1 \approx r_2$ となるので、第 2 の項は $4ac\delta_4 \approx 4a^2r_1\delta_4^2$ となります。

したがって、 $\sqrt{d}$ の計算値は $\sqrt{d + 4a^2r_1^2\delta_4}$ となります。

次の不等式

$$p - q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q, p \geq q > 0$$

は、 $\sqrt{d + 4a^2r_1^2\delta_4} = \sqrt{d} + E$ (ただし $|E| \leq \sqrt{4a^2r_1^2|\delta_4|}$) であることを示すので、 $\sqrt{d}/2a$ の絶対誤差は約 $r_1\sqrt{\delta_4}$ となり、根 $r_1 \approx r_2$ の下位半分を破壊します。 $\delta_4 \approx \beta^{-p}$ なので、 $\sqrt{\delta_4} \approx \beta^{-p/2}$ となり、絶対誤差は $r_1\sqrt{\delta_4}$ になります。すなわち、平方根の計算では $(\sqrt{d})/(2a)$ の計算を伴い、この式では r_i の下位半分に対応する位置に意味のあるビットが格納されないため、 r_i の下位ビットは有意になりません。■

最後に、定理 6 を検証してみます。これは、241 ページの「定理 14 と定理 8」の項で証明する事実にもとづく説明です。

定理 14

$0 < k < p$ として、 $m = \beta^k + 1$ をセットし、浮動小数点演算が正確に計算されるものと仮定する。 $(m \otimes x) \ominus (m \otimes x \ominus x)$ は有意桁 $p - k$ に丸めた正確な x として計算される。さらに正確に言うと、最下位桁 k の左の位置に基数ポイントがあることを前提に、整数に丸めることにより、 x が有意桁 x を使用して丸められる。

1. この略式の証明では、 $\beta = 2$ を前提とするので、4 による乗算は正確に行われるため、 δ_1 は必要ありません。

定理 6 の証明

定理 14 により、 x_h は $p-k=[p/2]$ に丸められます。キャリーアウトがなければ、 x_h は $[p/2]$ の有意桁で表わすことができます。キャリーアウトがある場合、 $x = x_0.x_1 \dots x_{p-1} \times \beta^e$ であれば、 x_{p-k-1} に 1 が加算されます。キャリーアウトが存在できるのは $x_{p-k-1} = \beta - 1$ の場合だけに限られ、 x_h の下位桁は $1 + x_{p-k-1} = 0$ となるので、 x_h は $[p/2]$ 桁で表現することができます。

x_l を扱えるように、 $\beta^{p-1} \leq x \leq \beta^p - 1$ を満たす整数に x を基準化 (スケール) します。 $x = \bar{x}_h + \bar{x}_l$ (ただし $\bar{x}_h$ は x の上位桁 $p-k$ で、 $\bar{x}_l$ は下位桁 k) とします。

考慮すべきケースが 3 つあります。 $\bar{x}_l < (\beta/2)\beta^{k-1}$ の場合、 $p-k$ 桁に x を丸める操作はチョッピング、および $x_h = \bar{x}_h$ 、 $x_l = \bar{x}_l$ と同じになります。 $\bar{x}_l$ の最大桁は k なので、 p が偶数の場合、 $\bar{x}_l$ の最大桁は $k=[p/2]=[p/2]$ となります。これ以外の場合は、 $\beta=2$ 、および $\bar{x}_l < 2^{k-1}$ は、有意桁 $-1 \leq [p/2]$ で表わすことができます。2 つ目のケースは、 $\bar{x} > (\beta/2) \beta^{k-1}$ の場合で、 x_h を計算すると切り上げが行われるので、 $x_h = \bar{x}_h + \beta^k$ になり、 $x_l = x - x_h = x - \bar{x}_h - \beta^k = \bar{x}_l - \beta^k$ になります。 $\bar{x}_l$ の最大桁は k になるので、 $[p/2]$ で表現できます。最後の $\bar{x}_l = (\beta/2) \beta^{k-1}$ の場合、切り上げの有無によって、 $x_h = \bar{x}_h$ 、または $\bar{x}_h + \beta^k$ となります。したがって、 x_l は $(\beta/2) \beta^{k-1}$ 、または $(\beta/2) \beta^{k-1} - \beta^k = -\beta^k/2$ になり、いずれも 1 桁で表わすことができます。 ■

定理 6 により、2 つの精度の数を正確な和として表現することができます。和を正確に表わすための公式があります。 $|x| \geq |y|$ のとき $x+y = (x \oplus y) + (x \ominus (x \oplus y)) \oplus y$ (Dekker 1971, Knuth 1981, Theorem C 第 4.2.2 節)。ただし正確な丸め操作を使用した場合、この公式は、 $x=0.99998$ 、 $y=0.99997$ の例に示すとおり $\beta=2$ の場合に限り真になり、 $\beta=10$ には適用されません。

2 進数から 10 進数への変換

単精度では、 $p=24$ 、 $2^{24} < 10^8$ になるので、2 進数から 8 桁の 10 進数に変換しても、もとの 2 進数を復元できるものと期待しがちですが、実際にはそうではありません。

定理 15

2 進の IEEE 単精度数をもっとも近い 8 桁の 10 進数に変換した場合、必ずしも 10 進数からもとの 2 進数を一意に復元できるわけではない。ただし、9 桁の 10 進数を使用すれば、10 進数からもっとも近い 2 進数への変換により、もとの浮動小数点数が復元される。

証明

2 進の単精度数は半分開いた間隔 $[10^3, 2^{10}) = [1000, 1024)$ に存在するので、2 進小数点の右側には 10 ビット、左側には 14 ビットがそれぞれ置かれます。したがって、この間隔の中に計 $(2^{10} - 10^3)2^{14} = 393,216$ 個の 2 進数が存在することになります。10 進数を 8 桁で表わすと、同じ間隔に $(2^{10} - 10^3) 10^4 = 240,000$ 個の 10 進数が存在します。240,000 個の 10 進数で 393,216 個の 2 進数を表現することができないのは明らかです。したがって、単精度の 2 進数を一意に表現するには、8 桁の 10 進数では不十分ということになります。

9 桁あれば十分であるという点は、各 2 進数間の間隔がつねに 10 進数どうしの間隔より大きいことを明らかにすれば、実証されます。これにより、各 10 進数 N ごとに、

$[N - \frac{1}{2} \text{ulp}, N + \frac{1}{2} \text{ulp}]$ の間隔に最大 1 つの 2 進数が含まれることが保証されます。したがって、各 2 進数はそれぞれ一意の 10 進数に丸められる一方、10 進数は一意の 2 進数に変換されます。

2 進数どうしの間隔が 10 進数の間隔よりつねに大きいことを証明するために、 $[10^n, 10^{n+1}]$ という間隔の場合を考えてみます。この間隔の場合、連続した 10 進数の間隔は、 $10^{(n+1)-9}$ になります。 $[10^n, 2^m]$ で m が最小整数の場合、 $10^n < 2^m$ 、2 進数どうしの間隔は 2^{m-24} となり、この間隔はしだいに広がります。したがって、 $10^{(n+1)-9} < 2^{m-24}$ であることを確認すれば十分ということになります。しかし実際には、 $10^n < 2^m$ になるので、 $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^{m-24}$ となります。■

同じ説明を倍精度の場合にも適用して、17 桁の 10 進数を使用すれば、倍精度数を一意に復元できることを証明できます。

2 進数と 10 進数の変換は、フラグの使用に関する例にも適用されます。193 ページの「精度」で、10 進数の拡張から 2 進数を復元するには、10 進数から 2 進数への変換を正確に実行しなければならないと説明しました。この変換は、拡張単精度フォーマットの N と $10^{|p|}$ (いずれも $p < 13$ の場合は正確な数) を乗じた後、これを単精度に丸めることによって行います($p < 0$ の場合は除算で、いずれの場合も同じ操作)。 $N \cdot 10^{|p|}$ の計算が正確でないことは言うまでもありません。拡張単精度から単精度に変換する正確な丸め操作を組み合わせたもの $(N \cdot 10^{|p|})$ であるからです。正確さを欠く可能性のある理由を見るために、 $\beta = 10$ 、 $p = 2$ (単精度の場合)、また $p = 3$ (拡張単精度の場合) の単純なケースを考えてみます。積が 12.51 の場合、これは拡張単精度の乗算の 1 部として 12.5 に丸められます。単精度に丸めると、12 になります。しかし、積 12.51 を単精度に丸めると 13 になるので、解は正しくありません。この誤差は倍精度による丸めによって起きたものです。

IEEE フラグを使用することにより、この倍精度の丸め誤差を避けることができます。まず、不正確なフラグの現在の値を保存して、これをリセットします。丸めモードを「ゼロへの丸め」にセットします。この後、 $N \cdot 10^{|P|}$ を計算します。不正確フラグの新しい値を `ixflag` に格納して、丸めモードと不正確フラグをリストアします。`ixflag` が 0 であれば、 $N \cdot 10^{|P|}$ は正確になるので、 $(N \cdot 10^{|P|})$ を最終ビットまで正確に丸めます。また `ixflag` が 1 であれば、「ゼロへの丸め」では必ず切り捨てが行われるので、桁が落ちている可能性があります。積の有意桁は、 $1.b_1...b_{22}b_{23}...b_{31}$ のようになります。倍精度の丸め誤差は、 $b_{23}...b_{31} = 10...0$ の場合に起きる可能性があります。これらのケースを検証するには、単に b_{31} に対し `ixflag` の論理 OR を実行します。これにより $(N \cdot 10^{|P|})$ の丸め操作は、すべてのケースについて正しく実行されるようになります。

加法の誤差

219 ページの「最適化プログラム」では、長大な和を正確に計算するときの問題を指摘しました。この精度を改善するためのもっとも簡単なアプローチは、精度を倍加することです。精度を倍加した結果、和の精度がどの程度改善されるかを概算する場合に、 $s_1 = x_1$ 、 $s_2 = s_1 \oplus x_2, \dots, s_i = s_{i-1} \oplus x_i$ とします。次に、 $s_i = (1 + \delta_i)(s_{i-1} + x_i)$ (ただし $|\delta_i| \leq \varepsilon$) として、 δ_i の第 2 項を無視すると、次の式が得られます。

$$s_n = \sum_{j=1}^n x_j \left(1 + \sum_{k=j}^n \delta_k \right) = \sum_{j=1}^n x_j + \sum_{j=1}^n x_j \left(\sum_{k=j}^n \delta_k \right) \quad (31)$$

公式 (31) の最初の等式により、 $\sum x_j$ の計算値は、 x_j の摂動値に正確な加法を実行した場合と同じになります。最初の項 x_1 は $n\varepsilon$ によって摂動され、最後の項 x_n は ε によって摂動されます。公式 (31) の 2 つ目の等式は、誤差の範囲が $n\varepsilon \sum |x_j|$ によって規定されることを示すものです。精度を倍加すると、 ε が二乗されます。IEEE 倍精度フォーマットで和を求めると、 $1/\varepsilon \approx 10^{16}$ になるので、 n の妥当な値について $n\varepsilon \ll 1$ となります。したがって、精度を倍加すると、 $n\varepsilon$ の最大摂動が適用され、 $n\varepsilon^2 \ll \varepsilon$ に変更されます。これにより、Kahan の加法公式に関する 2ε という誤差範囲 (定理 8) は、単精度よりもはるかに改善されるものの、倍精度の場合ほどの精度は得られないことになります。

Kahan の加法公式が有効になる理由をわかりやすく説明するために、次の図に従って考えてみます。

$$\begin{array}{r}
 \boxed{S} \\
 + \quad \boxed{Y_h} \boxed{Y_l} \\
 \hline
 \boxed{T} \\
 \\
 \boxed{T} \\
 - \quad \boxed{S} \\
 \hline
 \boxed{Y_h} \\
 - \quad \boxed{Y_h} \boxed{Y_l} \\
 \hline
 \boxed{-Y_l} = C
 \end{array}$$

加法を実行するたびに、次のループに適用される係数 C が修正されます。まず、前のループで計算した修正値 C を X_j から減算して、正しく加数 Y を求めます。次にこの加数を現在の和に加算します。 Y の下位ビット (すなわち Y_l) は和には入れられません。次に $T - S$ を計算して、 Y の上位ビットを計算します。ここから Y を減じると、 Y の下位ビットが復元されます。これは、図の最初の和で失われたビットに相当します。これらのビットは、次のループの修正要素となります。**Knuth 資料 (1981) の 572 ページにある定理 8 の正式な証明は、241 ページの「定理 14 と定理 8」に紹介されています。**

まとめ

コンピュータシステムの設計者が、浮動小数点に関する部分を無視するのは珍しいことではありません。おそらく、これはコンピュータサイエンスの授業で浮動小数点について扱う機会がほとんど、あるいはまったくないことに起因しているものと思われます。このことは、浮動小数点が定量化できる問題ではなく、この問題を扱ったハードウェアやソフトウェアについてあれこれ騒ぎ立てる意味はないといった認識を広める原因にもなっています。

本資料では、浮動小数点について積極的に論理的な検討を加えることが可能であることを実証してきました。たとえば、相殺を伴う浮動小数点アルゴリズムは、使用するハードウェアで保護桁を確保し、拡張精度がサポートされた逆操作の可能な 2 進数 /10 進数変換に関する効果的なアルゴリズムを用意すれば、相対誤差を最小限に抑えることができます。使用するコンピュータシステムで浮動小数点をサポートされていれば、信頼性のある浮動小数点ソフトウェアの構築作業が大幅に簡素化されます。本資料で紹介した 2 つの例 (保護桁と拡張精度) 以外に、211 ページの「システムの側面」では、命令セットの設計からコンパイラの最適化に至るまで、浮動小数点を効率的にサポートするためのさまざまな例を紹介しました。

IEEE 浮動小数点標準が広く普及していることは、標準に準拠したコードの移植性が広がることを意味します。191 ページの「IEEE 標準」では、IEEE 標準の機能を利用して、実践的な浮動小数点コードをどのように記述できるかを具体的に説明しました。

謝辞

本資料は、1988 年 5 月から 7 月、Sun Microsystems 社の David Hough 氏の協力のもとに Sun Microsystems 社で開催された W. Kahan 氏の講義を参考に作成したものです。本資料の草稿を見直しに並々ならぬご協力と貴重なコメントをいただいた Kahan 氏と Xerox PARC の皆様、特に John Gilbert 氏には、深く感謝の意を表わすものです。また Paul Hilfinger 氏をはじめ、関係各者にも多大なご協力をいただいたことに感謝いたします。

参考資料

Aho, Alfred V., Sethi, R., and Ullman J. D. 1986. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

ANSI 1978. *American National Standard Programming Language FORTRAN*, ANSI Standard X3.9-1978, American National Standards Institute, New York, NY.

Barnett, David 1987. *A Portable Floating-Point Environment*, unpublished manuscript.

Brown, W. S. 1981. *A Simple but Realistic Model of Floating-Point Computation*, ACM Trans. on Math. Software 7(4), pp. 445-480.

- Cody, W. J et. al. 1984. *A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic*, IEEE Micro 4(4), pp. 86-100.
- Cody, W. J. 1988. *Floating-Point Standards — Theory and Practice*, in “Reliability in Computing: the role of interval methods in scientific computing”, ed. by Ramon E. Moore, pp. 99-107, Academic Press, Boston, MA.
- Coonen, Jerome 1984. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, PhD Thesis, Univ. of California, Berkeley.
- Dekker, T. J. 1971. *A Floating-Point Technique for Extending the Available Precision*, Numer. Math. 18(3), pp. 224-242.
- Demmel, James 1984. *Underflow and the Reliability of Numerical Software*, SIAM J. Sci. Stat. Comput. 5(4), pp. 887-919.
- Farnum, Charles 1988. *Compiler Support for Floating-point Computation*, Software-Practice and Experience, 18(7), pp. 701-709.
- Forsythe, G. E. and Moler, C. B. 1967. *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.
- Goldberg, I. Bennett 1967. *27 Bits Are Not Enough for 8-Digit Accuracy*, Comm. of the ACM. 10(2), pp 105-106.
- Goldberg, David 1990. *Computer Arithmetic*, in “Computer Architecture: A Quantitative Approach”, by David Patterson and John L. Hennessy, Appendix A, Morgan Kaufmann, Los Altos, CA.
- Golub, Gene H. and Van Loan, Charles F. 1989. *Matrix Computations*, 2nd edition, The Johns Hopkins University Press, Baltimore Maryland.
- Graham, Ronald L. , Knuth, Donald E. and Patashnik, Oren. 1989. *Concrete Mathematics*, Addison-Wesley, Reading, MA, p.162.
- Hewlett Packard 1982. HP-15C *Advanced Functions Handbook*.
- IEEE 1987. *IEEE Standard 754-1985 for Binary Floating-point Arithmetic*, IEEE, (1985). Reprinted in SIGPLAN 22(2) pp. 9-25.
- Kahan, W. 1972. *A Survey Of Error Analysis*, in Information Processing 71, Vol 2, pp. 1214 - 1239 (Ljubljana, Yugoslavia), North Holland, Amsterdam.
- Kahan, W. 1986. *Calculating Area and Angle of a Needle-like Triangle*, unpublished manuscript.

- Kahan, W. 1987. *Branch Cuts for Complex Elementary Functions*, in "The State of the Art in Numerical Analysis", ed. by M.J.D. Powell and A. Iserles (Univ of Birmingham, England), Chapter 7, Oxford University Press, New York.
- Kahan, W. 1988. Unpublished lectures given at Sun Microsystems, Mountain View, CA.
- Kahan, W. and Coonen, Jerome T. 1982. *The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments*, in "The Relationship Between Numerical Computation And Programming Languages", ed. by J. K. Reid, pp. 103-115, North-Holland, Amsterdam.
- Kahan, W. and LeBlanc, E. 1985. *Anomalies in the IBM Acrith Package*, Proc. 7th IEEE Symposium on Computer Arithmetic (Urbana, Illinois), pp. 322-331.
- Kernighan, Brian W. and Ritchie, Dennis M. 1978. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.
- Kirchner, R. and Kulisch, U. 1987. *Arithmetic for Vector Processors*, Proc. 8th IEEE Symposium on Computer Arithmetic (Como, Italy), pp. 256-269.
- Knuth, Donald E., 1981. *The Art of Computer Programming, Volume II, Second Edition*, Addison-Wesley, Reading, MA.
- Kulisch, U. W., and Miranker, W. L. 1986. *The Arithmetic of the Digital Computer: A New Approach*, SIAM Review 28(1), pp 1-36.
- Matula, D. W. and Kornerup, P. 1985. *Finite Precision Rational Arithmetic: Slash Number Systems*, IEEE Trans. on Comput. C-34(1), pp 3-18.
- Nelson, G. 1991. *Systems Programming With Modula-3*, Prentice-Hall, Englewood Cliffs, NJ.
- Reiser, John F. and Knuth, Donald E. 1975. *Evading the Drift in Floating-point Addition*, Information Processing Letters 3(3), pp 84-87.
- Sterbenz, Pat H. 1974. *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ.
- Swartzlander, Earl E. and Alexopoulos, Aristides G. 1975. *The Sign/Logarithm Number System*, IEEE Trans. Comput. C-24(12), pp. 1238-1242.
- Walther, J. S., 1971. *A unified algorithm for elementary functions*, Proceedings of the AFIP Spring Joint Computer Conf. 38, pp. 379-385.

定理 14 と定理 8

この項では、本書で扱っていない技術的な証明を 2 つ紹介します。

定理 14

$0 < k < p$ として、 $m = \beta^k + 1$ をセットし、浮動小数点演算が正確に計算されるものと仮定する。 $(m \otimes x) \ominus (m \otimes x \ominus x)$ は有意桁 $p - k$ に丸めた正確な x として計算される。さらに正確に言うと、最下位桁 k の左の位置に基数ポイントがあることを前提に、整数に丸めることにより、 x が有意桁 x を使用して丸められる。

証明

証明の手順は、 $mx = \beta^k x + x$ の計算結果におけるキャリーアウトの有無によって 2 つのケースに分けられます。

キャリーアウトがない場合は、 x が整数になるように基準化 (スケール) しても問題はありせん。これにより $mx = x + \beta^k x$ は次のようになります。

$$\begin{array}{r} aa \dots aabb \dots bb \\ + aa \dots aabb \dots bb \\ \hline zz \dots zzbb \dots bb \end{array}$$

ここで x は、2 つの部分に区分されています。下位の k 桁は b 、上位の $p - k$ 桁は a と表わします。 mx から $m \otimes x$ を計算すると、下位の k 桁 (b で表わした桁) が丸められます。

$$m \otimes x = mx - x \bmod (\beta^k) + r \beta^k \quad (32)$$

r の値は、 $.bb \dots b$ が $\frac{1}{2}$ より大きければ 1、これ以外の場合は 0 になります。厳密に言うと次のようになります。

$$a.bb \dots b \text{ を } a + 1 \text{ に丸める場合は } r = 1、\text{これ以外の場合は } r = 0 \quad (33)$$

次に $m \otimes x - x = mx - x \bmod (\beta^k) + r \beta^k - x = \beta^k(x + r) - x \bmod (\beta^k)$ を計算します。次の図に、 $m \otimes x - x$ の計算が丸められる、つまり $(m \otimes x) \ominus x$ であることを示します。上の段は $\beta^k(x + r)$ を表わします (ただし B は、最下位桁 b に r を加え

た場合の桁を表わします)。 $.bb\dots b < \frac{1}{2}$ の場合は $r=0$ となり、減算により B の

$$\begin{array}{r} aa\dots aabb\dots bB00\dots 00 \\ - bb\dots bb \\ \hline zz\dots zzZ00\dots 00 \end{array}$$

桁から借りが生じますが、差が丸められるため最終的には丸めの差が上段と同じ、すなわち $\beta^k x$ となります。また $.bb\dots b > \frac{1}{2}$ の場合は、 $r=1$ となり、借りのために B から 1 が減算され結果は同じく $\beta^k x$ となります。最後に $.bb\dots b = \frac{1}{2}$ の場合を考えます。 $r=0$ の場合、B は偶数、Z は奇数となり、差が切り上げられるので、結果は $\beta^k x$ となります。同様に $r=1$ であれば、B は奇数、Z は偶数になり、差は切り捨てられます。結果は同じく $\beta^k x$ となります。以上をまとめると次のようになります。

$$(m \otimes x) \ominus x = \beta^k x \quad (34)$$

(32) と (34) の方程式を組み合わせると、 $(m \otimes x) - (m \otimes x \ominus x) = x - x \bmod (\beta^k) + \rho \cdot \beta^k$ となります。これを計算すると、次の結果が得られます。

$$\begin{array}{r} r00\dots 00 \\ + aa\dots aabb\dots bb \\ - bb\dots bb \\ \hline aa\dots aA00\dots 00 \end{array}$$

方程式 (33) で r を計算するルールは、 $a\dots ab\dots b$ を $p-k$ 桁に丸める場合のルールと同じです。したがって、浮動小数点演算の精度で $mx - (mx - x)$ を計算することは、 $x + \beta^k x$ でキャリーアウトなしのときに x を $p-k$ 桁に丸める場合とまったく等しくなります。

$x + \beta^k x$ でキャリーアウトが伴う場合、 $mx = \beta^k x + x$ は次のようになります。

$$\begin{array}{r} aa\dots aabb\dots bb \\ + aa\dots aabb\dots bb \\ \hline zz\dots zZbb\dots bb \end{array}$$

したがって、 $m \otimes x = mx - x \bmod(\beta^k) + w\beta^k$ (ただし $Z < \beta/2$ の場合に $w = -Z$) となります (w の正確な値は重要ではありません)。次に $m \otimes x - x = \beta^k x - x \bmod(\beta^k) + w\beta^k$ になります。図解すると次のようになります。

$$\begin{array}{r} \text{aa...aabb...bb00...00} \\ - \text{bb...bb} \\ + \text{w} \\ \hline \text{zz} \quad \dots \text{zZbb} \quad \dots \text{bb}^1 \end{array}$$

これを丸めると $(m \otimes x) \ominus x = \beta^k x + w\beta^k - r\beta^k$ (ただし、 $\text{.bb...b} > \frac{1}{2}$ の場合、または $\text{.bb...b} = \frac{1}{2}$ で $b_0 = 1$ の場合に $r = 1$) となります¹。また $(m \otimes x) - (m \otimes x \ominus x) = mx - x \bmod(\beta^k) + w\beta^k - (\beta^k x + w\beta^k - r\beta^k) = x - x \bmod(\beta^k) + r\beta^k$ となります。この場合も、 a...ab...b を $p-k$ 桁に丸めるときに切り上げを伴うと、 $r = 1$ になります。したがって、あらゆるケースについて定理 14 が証明されたことになります。■

定理 8 (Kahan の加法公式)

$\sum_{j=1}^N x_j$ を次のアルゴリズムに従って計算する。

```
S = X [1];
C = 0;
for j = 2 to N {
  Y = X [j] - C;
  T = S + Y;
  C = (T - S) - Y;
  S = T;
}
```

このとき S の計算値は、 $S = \sum x_j (1 + \delta_j) + O(N \varepsilon^2) \sum |x_j|$ (ただし $|\delta_j| \leq 2\varepsilon$) に等しくなる。

証明

まず、単純な公式 $\sum x_i$ の誤差をどのように概算するかを考えてみます。 $s_1 = x_1$ 、 $s_i = (1 + \delta_i)(s_{i-1} + x_i)$ を導入します。和の計算値は、 s_n となり、これは各項 x を δ_j のある式で乗じた値 x_i の和に相当します。 x_1 の正確な係数は $(1 + \delta_2)(1 + \delta_3) \dots$

1. $(\beta^k x + w\beta^k)$ で $\beta^k x$ の形式が保持される場合に限り、丸めにより、 $\beta^k x + w\beta^k - r\beta^k$ となります。-Ed

$(1+\delta_n)$ なので、番号の変更により x_2 の係数は、 $(1+\delta_3)(1+\delta_4) \dots (1+\delta_n)$ にならなければなりません。定理 8 は、 x_1 の係数が多少複雑になる点を除き、まったく同じように証明することができます。 $s_0=c_0=0$ で、次のように仮定すると、

$$\begin{aligned}y_k &= x_k \ominus c_{k-1} = (x_k - c_{k-1}) (1 + \eta_k) \\s_k &= s_{k-1} \oplus y_k = (s_{k-1} + y_k) (1 + \sigma_k) \\c_k &= (s_k \ominus s_{k-1}) \ominus y_k = [(s_k - s_{k-1}) (1 + \gamma_k) - y_k] (1 + \delta_k)\end{aligned}$$

ただし、上記のギリシア文字はすべて ε で範囲が決まります。 s_k の x_1 の係数は極端な式ですが、 $s_k - c_k$ と c_k の x_1 の係数が計算しやすくなります。 $k=1$ のとき、次のようになります。

$$\begin{aligned}c_1 &= (s_1(1 + \gamma_1) - y_1) (1 + d_1) \\&= y_1((1 + s_1) (1 + \gamma_1) - 1) (1 + d_1) \\&= x_1(s_1 + \gamma_1 + s_1 g_1) (1 + d_1) (1 + h_1) \\s_1 - c_1 &= x_1[(1 + s_1) - (s_1 + g_1 + s_1 g_1) (1 + d_1)](1 + h_1) \\&= x_1[1 - g_1 - s_1 d_1 - s_1 g_1 - d_1 g_1 - s_1 g_1 d_1](1 + h_1)\end{aligned}$$

以上の式で仮に、 x_1 の係数をそれぞれ C_k と S_k とすると、次のようになります。

$$\begin{aligned}C_1 &= 2\varepsilon + O(\varepsilon^2) \\S_1 &= +\eta_1 - \gamma_1 + 4\varepsilon^2 + O(\varepsilon^3)\end{aligned}$$

S_k と C_k の公式が得られるように、 $i > 1$ の場合の x_i を伴う項をすべて無視して、 S_k と C_k の定義を拡張します。結果は次のとおりです。

$$\begin{aligned}
s_k &= (s_{k-1} + y_k)(1 + \sigma_k) \\
&= [s_{k-1} + (x_k - c_{k-1})(1 + \eta_k)](1 + \sigma_k) \\
&= [(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}](1 + \sigma_k) \\
c_k &= [(s_k - s_{k-1})(1 + \gamma_k) - y_k](1 + \delta_k) \\
&= [(((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1 + \sigma_k) - s_{k-1})(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k) \\
&= [((s_{k-1} - c_{k-1})\sigma_k - \eta_k c_{k-1}(1 + \sigma_k) - c_{k-1})(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k) \\
&= [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))](1 + \delta_k), \\
s_k - c_k &= ((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1 + \sigma_k) \\
&\quad - [((s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))](1 + \delta_k) \\
&= (s_{k-1} - c_{k-1})((1 + \sigma_k) - \sigma_k(1 + \gamma_k)(1 + \delta_k)) \\
&\quad + c_{k-1}(-\eta_k(1 + \sigma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))(1 + \delta_k)) \\
&= (s_{k-1} - c_{k-1})(1 - \sigma_k(\gamma_k + \delta_k + \gamma_k \delta_k)) \\
&\quad + c_{k-1}[-\eta_k + \gamma_k + \eta_k(\gamma_k + \sigma_k \gamma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))\delta_k]
\end{aligned}$$

S_k と C_k は ε^2 の位までしか計算されないので、これらの公式は次のように簡略化できます。

$$\begin{aligned}
C_k &= (\sigma_k + O(\varepsilon^2))S_{k-1} + (-\gamma_k + O(\varepsilon^2))C_{k-1} \\
S_k &= ((1 + 2\varepsilon^2 + O(\varepsilon^3))S_{k-1} + (2\varepsilon + O(\varepsilon^2))C_{k-1}
\end{aligned}$$

これらの式を使用すると、次のようになります。

$$\begin{aligned}
C_2 &= \sigma_2 + O(\varepsilon^2) \\
S_2 &= 1 + \eta_1 - \gamma_1 + 10\varepsilon^2 + O(\varepsilon^3)
\end{aligned}$$

一般に次のように帰納法でチェックすると簡単です。

$$C_k = \sigma_k + O(\varepsilon^2)$$

$$S_k = 1 + \eta_1 - \gamma_1 + (4_k+2)\varepsilon^2 + O(\varepsilon^3)$$

最終的に s_k の x_1 の係数が必要になります。この値を得るには、 $x_{n+1} = 0$ とし、 $n+1$ という添え字の付いたギリシア文字をすべて 0 として s_{n+1} を計算します。この結果、 $s_{n+1} = s_n - c_n$ となり、 s_n の x_1 の係数は s_{n+1} の係数より小さくなります。この係数は、 $s_n = 1 + \eta_1 - \gamma_1 + (4n+2)\varepsilon^2 = (1 + 2\varepsilon + O(n\varepsilon^2))$ となります。■

IEEE 754 実装間の違い

注 - この節は、発行された論文の一部ではありません。この節は、IEEE 規格の一部を説明し、読者が論文から推量しやすい IEEE についての思い違いを正すために追加されたものです。この資料は David Goldberg によって記述されたものではありませんが、Goldberg 氏の許可を得てここに含められています。

前記の論文は、プログラマはプログラムの正確度を浮動小数点演算の特性に頼ることがあるため、浮動小数点演算は十分な注意を払って実装しなければならないことを示しています。IEEE 規格は注意深い実装を規定しており、正しく動作する便利なプログラムを作成してこの規格に準拠したシステムにのみ正確な結果を配布することは可能です。そのようなプログラムはすべての IEEE システムに移植できなければならないと読者は結論付けるかもしれません。実際、196 ページの「あるマシンから別のマシンにプログラムを移植するときに、両方のマシンで IEEE 標準がサポートされていれば、その間に生成される中間結果はすべてソフトウェアバグに原因があり、演算上の差によるものでない」という所見が真実であるなら、移植性のあるソフトウェアの記述は比較的簡単でしょう。

残念ながら IEEE 規格は、同一のプログラムが IEEE 準拠のすべてのシステムでまったく同じ結果を出すとは保証していません。実際、さまざまな理由から、ほとんどのプログラムはシステムによって異なった結果を出します。一例を挙げると、ほとんどのプログラムは 10 進形式と 2 進形式の間で数値を変換させる必要がありますが、IEEE 規格はこの変換を行うための正確度を完全には指定していません。また、多くのプロ

グラムはシステムライブラリによって供給される基本関数を使用しますが、IEEE 規格はこれらの関数についてはまったく規定していません。当然、ほとんどのプログラマはこれらの機能が IEEE 規格の範囲外であることを認識しています。

IEEE 規格に規定された数値形式と演算だけを使用するプログラムですら、システムによって異なる結果を出す場合があることに気づいていないプログラマも多いことでしょう。実際、IEEE 規格の執筆者は、異なる実装が異なる結果を生むことを認めました。それは、次に示す IEEE 754 規格の用語「宛先」の定義で明らかです。「宛先は、ユーザーによって明示的に指定されるか、システムにより暗黙に供給される (たとえば、プロシージャのサブエクスプレッションや引数における中間結果)。言語によっては、中間計算の結果をユーザー制御の及ばない宛先に置くものもあるが、当規格は演算結果を宛先の形式とオペランドの値によって定義する。」(IEEE 754-1985、7 ページ) つまり IEEE 規格は各結果をその宛先の精度に正しく丸めることを規定していますが、その宛先の精度をユーザーのプログラムで決定するとは規定していません。したがって、各システムは独自の精度で結果を宛先に配布できるため、システムがすべて IEEE 規格に準拠していても、同一のプログラムで異なる結果が出てしまいます (ときどきこの程度が著しくなります)。

前記の論文に示されたいくつかの例は、浮動小数点演算の丸めに関する知識を前提にしています。このような例を頼みにするには、プログラマはプログラムがどのように解釈されるか (具体的に IEEE システムでは各算術演算の宛先の精度) を予測する必要があります。何と、IEEE 規格の「宛先」定義は、その狭間でプログラムがどのように解釈されるかを認識するプログラマの能力を揺るがしています。その結果、前記の例のいくつかは、高級言語内で一見移植可能なプログラムとして実装される場合、通常プログラマが予期するものとは異なる精度で宛先に結果を配布する IEEE システムでは正しく動作しない場合があります。正しく動作するものもあるかもしれませんが、それらの動作が平均的なプログラマの能力を超える場合があります。

この節では、IEEE 754 演算が通常使用する宛先形式の精度に基づいた IEEE 754 演算の既存の実装について分類します。そのあとで、論文の一部の例を取り上げ、プログラムが予期する精度よりも広い精度で結果を配布すると、予期された精度が使用されるときには正しいはずでも、間違った結果が生成される場合があることを示します。また、論文内の証明の 1 つを再考して、予期しない精度に取り組む (その精度がプログラムを無効にしないにせよ) ために必要な努力を示します。これらの例は、IEEE 規格のあらゆる規定にもかかわらず、IEEE 規格が認める異なる実装間の相違のために、動作が正確に予測でき、移植可能で効率的であるという数値ソフトウェアを記述できない場合があることを示しています。このようなソフトウェアを開発するには、IEEE

規格が認めている可変性を制限するとともに、プログラムが依存する浮動小数点意味論をプログラマが表現できるプログラミング言語と環境をまず作成する必要があります。

現在の IEEE 754 実装

IEEE 754 演算の現在の実装は、ハードウェア内でそれらがサポートする各種の浮動小数点形式の程度により 2 つのグループに分けることができます。Intel x86 ファミリのプロセッサで代表される「拡張ベース」システムは拡張倍精度形式をフルサポートしますが、単精度と倍精度については部分的にしかサポートしません。拡張ベースシステムは、単精度と倍精度でデータの読み込みと格納を行い、データを拡張倍精度形式との間ですばやく変換する命令を提供します。このシステムはまた、拡張倍精度形式でレジスタに保持される場合でも算術演算の結果が単精度または倍精度に丸められる、デフォルトとは別の特殊なモードを提供します。Motorola 68000 シリーズプロセッサは、結果をこれらのモードで単精度形式または倍精度形式の精度と範囲の両方に丸めます。Intel x86 および互換プロセッサは、単精度または倍精度形式の精度に結果を丸めますが、同じ範囲を拡張倍精度形式として保持します。ほとんどの RISC プロセッサを含む単精度および倍精度システムは、単精度形式と倍精度形式をフルサポートしますが、IEEE 準拠の拡張倍精度形式はサポートしません。IBM POWER アーキテクチャは単精度については部分的なサポートしか提供していませんが、この節では単精度 / 倍精度システムとして分類します。

拡張ベースシステムにおける演算動作が単精度 / 倍精度システムにおける動作とどう異なるかを確認するために、211 ページの「システムの側面」の C による例を考えてみます。

```
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;
}
```

この例では、定数 3.0 と 7.0 が倍精度の浮動小数点数として解釈されており、式 3.0/7.0 は、データ型 double を継承します。単精度 / 倍精度システムでは、使用する上で倍精度がもっとも効率的な形式であるため、この式は倍精度で評価されます。

そのため、倍精度に正しく丸められた値 3.0/7.0 が q に代入されます。次の行で、式 3.0/7.0 は再び倍精度で評価されます。その結果は当然 q に代入したばかりの値と等しいため、プログラムは予測どおり「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 の場合でも、商はレジスタ内で拡張倍精度形式で (つまりデフォルトモードで) 計算され、拡張倍精度に丸められます。しかし、結果の値が変数 q に代入される際にこの値はメモリーに格納され、 q は 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 は再び拡張精度で評価されることがあり、その場合には q に格納されている倍精度の値とは異なる結果を生成し、プログラムは「Not equal」と出力します。当然、ほかの出力も考えられます。コンパイラは、2 番目の行で式 3.0/7.0 の値の格納と丸めを行なった後でそれを q と比較することも、値を格納せずにレジスタに拡張精度で q を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 を倍精度で評価することも拡張倍精度で評価することも考えられます (1 つの x86 コンパイラにおいて、最適化によりコンパイルするときにプログラムは「Equal」と出力し、デバッグのためにコンパイルするときは「Not Equal」と出力します)。拡張ベースシステム用のコンパイラの中には、丸め精度モードを自動的に変更し、レジスタ内に結果を生成する演算によって結果を (より広い範囲を使用して) 単精度または倍精度に丸めるものもあります。そのため、このようなシステムでは、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。私たちは、ハードウェアやコンパイラが IEEE 754 準拠の環境を提供しないからといって責めることはできません。ハードウェアは、求められるとおり、正確に丸められた結果をそれぞれの宛先に配布しています。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宛先に中間結果を代入しています。

拡張ベースシステムにおける演算の落とし穴

世間一般の通念では、拡張ベースシステムは、単精度 / 倍精度システムで配布される結果より正確とは言えなくても、最低限正確な結果を生成しなければならないとされています。これは、拡張ベースシステムが常に少なくとも単精度 / 倍精度システムと同じ正確度を提供しており、単精度 / 倍精度システムを超える正確度を提供する場合も多いからです。前記の C プログラムのような単純な例や以下で説明する例に基づいた比較的微妙なプログラムは、この通念がせいぜい理想論であることを示しています。単精度 / 倍精度システムには確かに移植できるような、見かけ上移植性のあるプログラムの中には、拡張ベースシステムでは明らかに不正確な結果を出すものがあります。これは、コンパイラとハードウェアが、時折プログラムが期待する以上の精度を提供しようとするためです。

現在の各種のプログラミング言語では、プログラムが期待する精度を指定することが困難です。214 ページの「言語とコンパイラ」の節で触れているように、多くのプログラミング言語は、同じコンテキスト内で $10.0*x$ のような式が発生するごとにそれらが同じ値に評価されなければならないとは指定していません。Ada など一部の言語は、この点について IEEE 規格以前のさまざまな演算におけるバリエーションの影響を受けています。さらに新しい ANSI C のような言語は、標準準拠の拡張ベースシステムに影響を受けています。実際 ANSI C 標準は、コンパイラが浮動小数点式を通常その種類に対応づけられている精度よりも広い精度に評価することを明確に許可しています。その結果、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ります。-- その式が変数にただちに代入されるのか、それともより大きな式の一部として表示されるのか、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、また渡される場合、引数は値と参照のどちらとして渡されるか、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。

予測のつかない式の評価は、言語規格にすべての原因が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場合にもっとも効率よく動作します。しかし、格納が必要な値は、要求されるもっとも狭い精度で格納されます。 $10.0*x$ があらゆる位置で同じ値に評価されるように言語で要求させると、これらのシステムでパフォーマンス低下というペナルティが避けられなくなります。残念ながら、構文上同じコンテキストで $10.0*x$ を別の値に評価することをこれらのシステムに許すと、それ自体のペナルティが正確な数値ソフトウェアを開発するプログラマにかかり、意図する意味論を表現するためにプログラムの構文に頼ることができなくなります。

実際のプログラムは、指定された式は常に同じ値に評価されるという仮定に頼っているのでしょうか。 $\ln(1+x)$ を計算する定理 4 で示されたアルゴリズムを思い出してください。次に、FORTRAN による記述を示します。

```
real function loglp(x)
real x
if (1.0 + x .eq. 1.0) then
    loglp = x
else
    loglp = 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 と比較できます。しかし、この同じ式が 6 行目で `log` 関数に渡される時、コンパイラはその値をメモリーに格納して、値を単精度に丸めることができます。そのため x のサイズが、拡張精度で $1.0 + x$ が 1.0 に丸められるほど小さくないが、単精度では $1.0 + x$ が 1.0 に丸められるくらい小さいという場合、`log1p(x)` が返す値は x ではなくゼロになり、相対誤差は 1 になります (5e よりもいくぶん大きい)。同様に、サブエクスプレッション $1.0 + x$ が再指定されている 6 行目の式の残り部分が拡張精度で評価されることを想定します。この場合、 x が、小さいが $1.0 + x$ が単精度で 1.0 に丸められるほどには小さくないというとき、`log1p(x)` が返す値は正しい値よりも大きくなります。この差は x とほぼ同じで、相対誤差はこの場合も 1 に近づきます。具体的な例として、 x を $2^{-24} + 2^{-47}$ と想定します。つまり x は、 $1.0 + x$ が次に大きな数値 $1 + 2^{-23}$ に丸められるような最小の単精度数です。この場合、`log(1.0 + x)` はおよそ 2^{-23} です。6 行目の式の分母は拡張精度で評価されるため、正確に計算されて x を配布します。そのため `log1p(x)` は正確な値のほぼ 2 倍にあたるおよそ 2^{-23} を返します。このような状況は、少なくとも 1 つのコンパイラで発生します。前記のコードが x86 システム用の Sun WorkShop Compiler 4.2.1 FORTRAN 77 コンパイラで `-o` 最適化フラグを使用してコンパイルされる場合、生成されたコードは $1.0 + x$ を上記のとおり計算します。その結果、この関数は `log1p(1.0e-10)` にゼロを配布し、`log1p(5.97e-8)` に `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ためには、式 $1.0 + x$ は出現するたびに同じ方法で評価されなければなりません。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、あるときは単精度や倍精度に評価されるというように一定でない場合だけ、拡張ベースシステムで正しい結果を出しません。`log` は FORTRAN の総称組み込み関数であるため、当然コンパイラは式 $1.0 + x$ を終始拡張精度で評価してその対数を同じ精度で計算できますが、コンパイラがそのように実行すると決めてかかることはできません (ユーザー定義の関数を使用した類似例の場合、関数が単精度の結果を返す場合でもコンパイラは拡張精度で引数を保持できます。しかし、もし存在したとしても、これを実行する FORTRAN コンパイラはほとんどありません)。そのため、 $1.0 + x$ を変数に代入することにより、この式を常に同じように評価させようとプログラマは試みるかもしれません。残念ながら、変数 `real` を宣言すると、ある変数を拡張精度でレジスタに格納されている値で置換し、別の変数を単精度でメモリーに格納されている値で置換するようなコンパイラで裏をかかれる場合があります。そこで、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 FORTRAN 77 は、このような方法を提供していません。FORTRAN 95 は各種の形式を表現する `SELECTED_REAL_KIND` メカニズムを提供していますが、変数が拡張

精度で宣言されるように拡張精度で式を評価する実装をはっきりとは規定していません。つまり、標準の FORTRAN には、式 $1.0 + x$ が私たちの証明を無効にしてしまう方法で評価されることを確実に防ぐような可搬性のある方法はありません。

拡張ベースシステムでは、各サブエクスプレッションが格納され、したがって同じ精度に丸められている場合でも、誤動作する例がほかにもあります。原因は二重の丸めです。デフォルトの精度モードでは、拡張ベースシステムは初めにそれぞれの結果を拡張倍精度に丸めます。その後この結果が倍精度に格納される場合には、再び丸めが行われます。この 2 度の丸めの組み合わせにより、最初の結果を正しく倍精度に丸めるときに得られる値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結果が「中間値」(2 つの倍精度数のちょうど中間)であるため、2 度めの丸めが結合を偶数に丸める規則によって決定される場合に起きます。この 2 番目の丸めが最初の丸めと同じ方向で行われる場合、最終的な丸め誤差は最終桁のユニットの半分を超えます。しかし、二重の丸めは倍精度演算にしか影響を与えません。初めに 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、2 つの p ビット数値の合計、差、積、または商、あるいは p ビット数値の平方根は、 $q \geq 2p + 2$ の場合、結果があたかも 1 度だけで p ビットに丸められたかのように同じ値を生成することがわかります。拡張倍精度は十分広いいため、単精度演算は二重の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、二重の丸めに対応できないものもあります。実際、正確な丸めを必要とせず IEEE 754 に準拠しないさまざまなマシンで正しく動作するようなアルゴリズムでも二重の丸めで動作しなくなる場合があります。これらの中でもっとも便利なのは、188 ページの「定理 5」で説明しているシミュレートされた多重精度演算を実行するための移植性のあるアルゴリズムです。たとえば、浮動小数点数を上位と下位の部分に分割するための定理 6 で述べられた方法は、二重の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、それぞれ最大 26 ビットで 2 つの部分に分割してみてください。それぞれの演算が倍精度に正しく丸めれるとき、上位の部分は $2^{52} + 2^{27}$ となり下位の部分は $2^{26} - 1$ となりますが、それぞれの演算が初めに拡張倍精度に丸められ続いて倍精度に丸められるとき、上位部分として $2^{52} + 2^{28}$ 、下位部分として $-2^{26} - 1$ が生成されます。 $-2^{26} - 1$ は 27 ビットを必要とするため、その二乗は倍精度では正確に計算できません。もちろん、この数値の二乗を拡張倍精度で計算することは可能ですが、その結果生まれるアルゴリズムは単精度 / 倍精度システムには移植できません。また、多重精度の乗算アルゴリズムによるその後のステップは、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ます。倍精度変数と拡張倍精度変数の組み合わせを正確に処理するには、実装の手間が大幅に増えます。

同様に、倍精度値の配列として表現される多重精度値を加える移植性のあるアルゴリズムは、二重に丸めを行う演算では失敗することがあります。このようなアルゴリズムは、通常Kahanの加法公式に類似した手法に頼っています。236 ページに挙げられた、加法公式を簡単に示した図からわかるように、 s と y が $|s| \geq |y|$ である浮動小数点変数であり、次の計算を実行する場合、ほとんどの演算において e は t の計算で発生した丸め誤差を正確に回復させます。

$$\begin{aligned} t &= s + y; \\ e &= (s - t) + y; \end{aligned}$$

しかしこの手法は、二重に丸められる演算では無効です。 $s = 2^{52} + 1$ 、 $y = 1/2 - 2^{-54}$ の場合、 $s + y$ は初めに拡張倍精度で $2^{52} + 3/2$ に丸められ、さらに結合を偶数に丸める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 の計算における最終的な丸め誤差は $1/2 + 2^{-54}$ です。これは、倍精度で正確に表現することは不可能であり、そのため上記の式では正確に計算できません。この場合も、拡張倍精度のまま合計を計算することにより丸め誤差を回復できますが、そうするとプログラムは最終出力を減らして倍精度に戻す作業を行う必要があり、二重の丸めがこのプロセスも悩ます可能性があります。このような理由から、これらの方法による多重精度演算のシミュレートを行う移植性のあるプログラムは多様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どおりには動作しません。

一見正確な丸めに依存してい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。このようなケースでは、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検証するために、手間のかかる二重の丸めが行われます。このことを説明するため、定理 7 の変更例を次に証明します。

定理 7'

m と n が IEEE 754 倍精度で表現可能な $|m| < 2^{52}$ の整数で、 n が $n = 2^i + 2^j$ という特殊な式で表される場合、両方の浮動小数点演算が倍精度に正しく丸められるか、あるいは初めに拡張倍精度に丸められて続いて倍精度に丸められるとき、 $(m \oslash n) \otimes n = m$ である。

証明

損失なしで $m > 0$ と仮定します。 $q = m \oslash n$ とします。2 の累乗でスケールし、 $2^{52} \leq m < 2^{53}$ 、 $2^{52} \leq q < 2^{53}$ となる同等の設定を考えることができます。この場合、 m と q の両方とも、最下位ビットがユニットの桁を占める整数 (つまり $\text{ulp}(m) = \text{ulp}(q) = 1$)

です。スケール前に $m < 2^{52}$ と仮定したため、スケール後 m は偶数の整数です。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 と q のどちらが大きいかに基づいて次の 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 の累乗値の合計であるため、特定の k については $n = 1 + 2^k$ です。同様に、 $q > m$ の場合 $1/2 < n < 1$ であるため、 $n = 1/2 + 2^{-(k+1)}$ です (n は 2 つの 2 の累乗値の合計であるため、1 にもっとも近い値である n は $n = 1 + 2^{-52}$ 。 $m/(1 + 2^{-52})$ は、 m より小さい 2 番目に小さな倍精度値よりも大きくないため、 $q = m$ とはなりません)。

q を求めるための丸め誤差を e と仮定し、 $q = m/n + e$ 、計算された値 $q \otimes n$ は $m + ne$ を 1 度または 2 度丸めた値であるとしてます。最初に、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、 $|e| < 1/2$ です。 n の式が $1/2 + 2^{-(k+1)}$ の場合、 $ne = nq - m$ は $2^{-(k+1)}$ の整数倍で、 $|ne| < 1/4 + 2^{-(k+2)}$ です。これは、 $|ne| \leq 1/4$ を意味します。 m と次に大きな表現可能な数値の差は 1 であり、 m と次に小さな表現可能な数値の差は $m > 2^{52}$ の場合 1、 $m = 2^{52}$ の場合 $1/2$ であることを思い出してください。このため、 $|ne| \leq 1/4$ 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、積は結合を偶数に丸める規則によって m に丸められます)。同様に、 n の式が $1 + 2^k$ の場合、 ne は 2^k の整数倍で、 $|ne| < 1/2 + 2^{-(k+1)}$ です。これは、 $|ne| \leq 1/2$ を意味します。 m は q よりも確実に大きいので、この場合 $m = 2^{52}$ にはなりません。このため、 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、 $|ne| \leq 1/2$ 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 結合を偶数に丸める規則によって積は m に丸められます)。正確に丸められる演算の証明は以上です。

二重の丸めを行う演算では (実際に 2 度丸められた場合でも) q が正確に丸められた商と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 度丸められるという事実を考慮するならば、上記段落の論証を利用できます。このことを説明するため、拡張倍精度形式は少なくとも 64 個の有効ビットを持つことを IEEE 規格が規定しており、そのため $m \pm 1/2$ と $m \pm 1/4$ は拡張倍精度で正確に表現可能であ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、そのため $|ne| \leq 1/4$ である場合に $m + ne$ を拡張倍精度に丸めると、 m と最大 $1/4$ の差がある結果が生成されます。上記に述べたように、この値は倍精度では m に丸められます。同様に、 n の式が $1 + 2^k$ であり、そのため $|ne| \leq 1/2$ である場合に $m + ne$ を拡張倍精度に丸めると、 m と最大 $1/2$ の差がある結果が生成され、この値は倍精度では 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、二重の丸めのために q が正しく丸められた商ではない場合を考えてみましょう。この場合、最悪の場合には $|e| < 1/2 + 2^{-(d+1)}$ となります。この場合、 d は拡張倍精度形式における余分なビットの数です (既存の拡張ベースシステムはすべて、ちょ

うど 64 個の有効ビットによる拡張倍精度形式をサポートしています。この形式では、 $d = 64 - 53 = 11$ 。二重の丸めは、結合を偶数に丸める規則によって 2 度目の丸めが決定される場合に限り不正確な丸め結果を出すため、 q は偶数の整数でなければなりません。したがって、 n の式が $1/2 + 2^{-(k+1)}$ の場合、 $ne = nq - m$ は 2^{-k} の整数倍で、 $|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}$ です。 $k \leq d$ の場合、 $|ne| \leq 1/4$ となります。 $k > d$ の場合、 $|ne| \leq 1/4 + 2^{-(d+2)}$ です。どちらの場合も、積の最初の丸めにより m と最大 $1/4$ の差がある結果が配付され、前記の論証に従って 2 度目の丸めにより m に丸められます。同様に、 n の式が $1 + 2^{-k}$ の場合、 ne は $2^{-(k-1)}$ の整数倍で、 $|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}$ です。 $k \leq d$ の場合、 $|ne| \leq 1/2$ です。 $k > d$ の場合、 $|ne| \leq 1/2 + 2^{-(d+1)}$ です。どちらの場合も、積の最初の丸めにより m と最大 $1/2$ の差がある結果が配付され、前記の証明に従って 2 度目の丸めにより m に丸められます。■

前記の証明は、商が二重の丸めを引き起こす場合にのみ積は二重の丸めを引き起こし、その場合でも正しい結果に丸めることを示しています。前記の証明はまた、推論を二重の丸めの可能性を含めるところまで拡張することは、たった 2 つの浮動小数点演算しか含まないプログラムでも困難であることも示しています。もっと複雑なプログラムでは、二重の丸めの効果を体系的に説明することは通常不可能です。倍精度演算と拡張倍精度演算の一般的な組み合わせは言うまでもありません。

拡張精度に対するプログラミング言語サポート

前記の例は、拡張精度 *per se* が有害であると述べているものではありません。プログラマが拡張精度を選択して使用できる場合には、多くのプログラムが拡張精度の恩恵を得ることができます。残念ながら、現在のプログラミング言語は拡張精度をいつどのような方法で使用するべきかプログラマが指定するための十分な手段を提供していません。どのようなサポートが必要かを示すため、ここでは拡張精度の使用を制御する方法を検討します。

名目上の作業精度として倍精度を使用する移植性のあるプログラムでは、より広い精度の使用を制御する方法として次の 5 つを利用できます。

1. 拡張ベースのシステムでは、可能なかぎり拡張精度を使用してもっとも速いコードを生成するようにコンパイルする。ほとんどの数値ソフトウェアでは、「マシニプシロン」が範囲設定する各演算の相対誤差以上の演算は必要としません。メモリ内のデータが倍精度で格納される場合、その精度における最大の相対丸め誤差として通常マシニプシロンが使用されます。そのため、入力データは入力時に丸めがなされたと (正しくまたは誤って) 想定され、結果はそれらが格納されるときに同様に丸められます。したがって、拡張精度での中間結果の計算はより正確な結果

を生成する場合もありますが、拡張精度は必須ではありません。この場合、感知されるほどプログラムの速度が低下しないときだけコンパイラで拡張精度を使用し、それ以外では倍精度を使用することをお勧めします。

2. 適度に高速で十分広い場合は倍精度よりも広い形式を使用し、それ以外ではほかの形式を使用する。計算の中には、拡張精度が使用できればもっと簡単に実行できるものがあります。しかし、そのような計算もいくらか余分な手間をかけるだけで、倍精度でも実行できます。倍精度値のベクトルのユークリッド正規形を計算することを想定します。要素の二乗を計算し、広い指数範囲を使用して IEEE 754 拡張倍精度形式でそれらの合計を累積することにより、実際の長さのベクトルに対する早まったアンダーフローまたはオーバーフローを自明な方法で防ぐことができます。拡張ベースシステムでは、これが正規形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、拡張倍精度形式はソフトウェア内でエミュレートする必要があり (拡張倍精度形式がサポートされている場合)、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するために例外フラグをテストし、もし発生している場合には明示的なスケールにより計算を繰り返すために、このようなエミュレーションは単純に倍精度を使用するよりも大幅に速度が低下します。拡張精度のこのような使用をサポートするために、言語は、使用するメソッドをプログラムが選択できるように、適度に高速で利用できる中で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。ユークリッド正規形の例よりも複雑なプログラムでは、プログラマは 2 種類のプログラムを記述する必要を避け、代わりに速度が遅くても拡張精度に頼ることを望む場合があります。この場合も言語は、利用できる中でもっとも範囲の広い形式の範囲と精度をプログラムが確認できるように、環境パラメータを提供する必要があります。
4. 倍精度より広い精度を使用しない (範囲が拡張される場合があるが、倍精度形式の精度に正しく丸められる)。上記で説明している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いたって簡単に記述されるプログラムの場合、中間結果が倍精度よりも広い指数範囲のレジスタで計算できるとしても、言語は拡張精度が使用され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る中間結果でも、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の丸めを引き起こします。算術演算の結果が初めに 53 個の有効ビットに丸められ、非正規化される必要があるときに引き続いてより少ない有

効ビットに再び丸められるような場合、最終結果は非正規化数に 1 度だけ丸める場合に取得された結果とは異なる場合があります。もちろん、このような二重の丸めが実際のプログラムに不利な影響を与える可能性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丸められる。倍精度のこのきびしい強制は、倍精度形式の範囲と精度の両方の限度近くで、数値ソフトウェアまたは演算そのものをテストするプログラムにもっとも便利です。そのような注意深いテストプログラムは、移植可能な方法で記述するのは通常困難であり、特定の形式に結果を強制的に丸めるためにダミーのサブルーチンやほかのトリックを使用しなければならない場合にはさらに難しく、エラーも生じやすくなります。したがって、拡張ベースシステムを使用してあらゆる IEEE 754 実装に移植可能な強固なソフトウェアを開発するプログラマは、並外れた労力をかけることなく単精度 / 倍精度システムの演算をエミュレートできることの真価をすぐに認めるようになるでしょう。

これらの 5 つのオプションをすべてサポートするような言語は現在ありません。実際、拡張精度の使用を制御できる機能をプログラマに提供した言語はほとんどありません。顕著な例外の 1 つは、現在標準化の最終段階にある C 言語の最新リビジョン、ISO/IEC 98 99; 1999 プログラミング言語である C 標準です。

現在の C 標準と同様、C99 標準は、通常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を許可しています。しかし C99 標準は、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。推奨されているこの 3 つの方法の区別は、式がより広い形式に「拡張」されるその程度にあります。C99 標準は、プリプロセッサマクロ FLT_EVAL_METHOD を定義することにより使用するメソッドを実装が識別することを推奨しています。FLT_EVAL_METHOD が 0 の場合、それぞれの式はその種類に対応する形式で評価されます。FLT_EVAL_METHOD が 1 の場合は、float の式は double に対応する形式に拡張されます。FLT_EVAL_METHOD が 2 の場合は、float と double は long double に対応する形式に拡張されます (式の評価方法が決定不可能であることを示すため、FLT_EVAL_METHOD を -1 に設定することが実装に許可されています)。C99 標準は、`<math.h>` ヘッダーファイルが型 `float_t` と `double_t` を定義することも規定しています。`float_t` は少なくとも float と同じ広さであり、`double_t` は少なくとも double と同じ広さです。これらは、float および double の式の評価に使用する型に通常一致します。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。C99 標準は、`<float.h>` ヘッダーファイルが、それぞれの浮動小数点の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を規定しています。

C99 標準が規定または推奨する機能を組み合わせると、上記に示した 5 つのオプションの一部がサポートされます。たとえば、実装が `long double` 型を拡張倍精度形式に割り当て、`FLT_EVAL_METHOD` を 2 と定義する場合、プログラムは拡張精度が比較的高速であり、そのためユークリッド正規形例のようなプログラムは型 `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、無名の式を拡張精度で保持する必要があります。これは、それらがメモリーに格納される場合 (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場合など) でも同様です。また、`double` と宣言された変数に割り当てられた式の結果を格納し、それらを倍精度に変換する必要があります。これは、式の結果をレジスタに保持できた場合でも同様です。したがって、`double` 型、`double_t` 型とも、現在の拡張ベースハードウェアではもっとも速いコードを生成するようにはコンパイルできません。

この節の例が示している一部 (すべてではない) の問題は、C99 標準が提供する手段で解決できます。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、`log1p` 関数の C99 標準バージョンは正しく動作することが保証されています。しかし、倍精度数を上位と下位の部分に分割するための移植性のある効率的な C99 標準プログラムは、比較的困難です。`double` の式が倍精度に正しく丸められることが保証できない場合、どうすれば正確な位置で分割し、二重の丸めを避けることができるでしょうか。1 つの解決法として、`double_t` 型を使用すれば、単精度 / 倍精度システムでは倍精度に分割し、拡張ベースシステムでは拡張精度に分割できます。これで、どちらの場合も演算は正しく丸められます。定理 14 は、基礎となる演算の精度がわかればどのビット位置においても分割が可能であるとしています。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;

```

当然このソリューションを見つけるには、プログラマは、double の式は拡張精度で評価されることがあること、二重の丸めの問題が継続するとアルゴリズムの誤動作を引き起こすことがあること、および定理 14 に従って拡張精度を代用できることを認識する必要があります。もっと明瞭なソリューションとして、それぞれの式が倍精度に正しく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丸め精度モードを変更すればすみますが、残念ながら C99 標準 はこれを実施する可搬性のある方法を提供していません。浮動小数点をサポートするために C 標準に加える変更を指定した作業文書、Floating-Point C Edit の初期の草案は、丸め精度モードを備えたシステムにおける実装で、丸め精度の取得と設定を行う fegetprec および fesetprec 関数 (丸め方向の取得と設定を行う fegetround および fesetround 関数に類似したもの) を提供することを推奨していました。この推奨は、C99 標準草案に変更が加えられる前に除かれました。

また、異なる整数演算機能を持つ各種のシステム間の移植性をサポートする 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をサポートするより優れた方法を提示しています。それぞれの C99 標準実装は、その実装がサポートする整数型を定義する <inttypes.h> ヘッダーファイルを供給しています。この整数型の名前は、それらのサイズと効率に従って付けられています。たとえば、int32_t はちょうど 32 ビットの幅の整数型、int_fast16_t はその実装でもっとも速い最低 16 ビット幅の整数型、intmax_t はサポートされている中でもっとも広い整数型です。浮動小数点の型についても、類似した体系を想像できます。たとえば、float53_t はちょうど

53 ビットの精度であるが範囲がこれを超えることもある浮動小数点、`float_fast24_t` は精度が最低 24 ビットであるその実装のもっとも高速な型、`floatmax_t` はサポートされている中でもっとも広い適度に高速な型の名前として使用できます。高速な型を使用すると、レジスタがあふれた結果として名前付き変数の値が変更されてはならないという制限がありますが、拡張ベースシステム上のコンパイラは可能なかぎりもっとも速いコードを生成できます。幅がちょうどどの型が使用されると拡張ベースシステムのコンパイラは、上記と同じ制限を条件として、丸め精度モードが指定された精度に丸めるように設定し、より広い範囲を許可します。`double_t` は、正確な倍精度評価を条件に、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使用できます。このような体系をしかるべき名前の付いた環境パラメータマクロとともに使用すると、前述した 5 つのオプションを簡単にサポートでき、プログラマはプログラムが必要とする浮動小数点意味論を簡単にかつ明白な形で指定できます。

拡張精度の言語サポートはそれほど複雑なのでしょうか。単精度 / 倍精度システムでは、上記の 5 つのオプション中 4 つが当てはまり、高速な型と正確な範囲の型を区別する必要はありません。しかし拡張ベースシステムでは、難しい選択にせまられます。拡張ベースシステムは、純粋な倍精度演算、純粋な拡張精度演算のどちらも、これらの組み合わせの場合と同じ効率ではサポートせず、プログラムによって異なる組み合わせを要求します。また、いつ拡張精度を使用するかという選択は、浮動小数点演算は「本来不正確」であり、したがって整数演算にふさわしくなく、その予測力もないとベンチマークから見なす傾向がある (時には数値アナリストによってこのように公然と語られることもあります) コンパイラライターに任せてはなりません。この選択はプログラマにゆだねるべきです。プログラマは、彼らの選択を十分表現できる言語を必要とします。

結論

前述の論評は、拡張ベースシステムを非難することがねらいではなく、その意図はいくつかの誤った議論 (その最たるものは、すべての IEEE 754 システムは同一プログラムに対しまったく同じ結果を出さなければならないというもの) を明らかにすることにあります。これまで拡張ベースシステムと単精度 / 倍精度システム間の違いに焦点を当ててきましたが、それぞれのグループにおける各種のシステム間にはさらに相違があります。たとえば、単精度 / 倍精度システムの中には、2 つの数を掛けて 3 つめの数を足す単一の命令で、最後の 1 つの丸めしか行わないものがあります。「乗算/加算の混合演算」と呼ばれるこの演算では、プログラムは単精度 / 倍精度システムごとに異なる結果を生成します。この演算では、拡張精度のように、プログラムが使用されるかどうか、そしていつ使用されるかによって、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があります (「乗算/加算の混合演算」は、分割せずに多重精度の乗算を実行するために非可搬的な方法で使用できますが、定理 6 の分割プロセスどおりにならない場合があります)。IEEE 規格が仮にこのような演算を予想しなかったとしても、中間の積はユーザー制御が及ばない正確に保持するだけの広さを持った「宛先」に配布され、最終の合計はその単精度または倍精度の宛先のサイズに合うように正しく丸められます。

IEEE 754 は一定のプログラムが配布すべき結果を正確に規定しているという考えは、やはり魅力的です。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プログラムを実行するコンピュータにかかわりなくプログラムの動作を理解でき、プログラムが正しく動作することを証明できると信じたがります。この確信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大いに価値のある目標です。残念ながら、浮動小数点演算に関しては、この目標は実際、達成が不可能です。IEEE 規格の執筆者はこれを認識しており、この達成を試みてはいません。その結果、コンピュータ業界のほぼ全体がほとんどの IEEE 754 標準に準拠しているにもかかわらず、移植性のあるソフトウェアの開発を手掛けるプログラマは、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 の特徴を利用するのであれば、浮動小数点演算を予想可能なものにできるプログラミング言語が必要です。C99 標準では、FLT_EVAL_METHOD ごとにプログラムを複数作成しなければなりません。幾分予測性が向上しています。将来の言語では IEEE 754 意味論に依存する範囲を明白に示す構文により、単一のプログラムを記述すればすむ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、演算が一定のシステムでどのように行われるべきかはプログラマよりもコンパイラとハードウェアの方がよく認識できると私たちに決め込ませることにより、この見通しを脅かしています。この決め込みは 2 つ目の誤信です。計算結果に求められる正確度は、結果を生成するマシンではなく、結果から引き出される結論によってのみ決まります。それらの結論が何かを把握できるのは、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です。

規格への準拠

Solaris 環境に対応する Compiler Collection のコンパイラ言語製品の中のコンパイラ、ヘッダーファイル、ライブラリは、複数の規格、すなわち System V Interface Definition (SVID) 第 3 版、X/Open、および ANSI C をサポートしています。その結果、数学ライブラリ libm とそれに関連したファイルも、C プログラムが各規格に準拠するように修正されました。この修正の変更点は、主として例外処理関係であるため、ユーザーのプログラムは通常は影響を受けません。

SVID の歴史

SVID に従った例外処理と IEEE 規格が表わしている立場との相違点を理解するためには、両者が発展してきた状況を検討してみる必要があります。SVID にある考え方の源は、その多くが UNIX の誕生間もない時期、つまりコンピュータ本体上に初めて実装された時期に発しています。このような初期の環境に共通しているのは、有理浮動小数点演算 $+$ 、 $-$ 、 $*$ 、 $/$ が不可分 (atomic) な機械命令であること、また一方では `sqrt`、浮動小数点形式での整数値への変換、ならびに基本超越整関数が多数の不可分な機械命令から成るサブルーチンであることです。

これらの環境では浮動小数点例外を多様な方法で処理しますが、一様性を持たせようとすると、不可分 (atomic) な各浮動小数点命令の前後にソフトウェア内で引数と結果をチェックしなければ実現できないと考えられます。しかし、これを行うと性能に及ぼす影響が大きくなりすぎると考えられるので、SVID ではゼロによる除算やオーバーフローなど浮動小数点例外の影響は明示していません。

サブルーチンによって実現される演算は、単一の不可分 (atomic) な浮動小数点命令に比べると速度が劣ります。引数と結果について特別なエラーチェックを行なっても性能にはほとんど影響がないので、SVID ではそのようなチェックを必須にしていま

す。例外が検出されると、デフォルト結果が指定され、不適格なオペランドの場合には `errno` が `EDOM` に設定されます。またオーバーフロー、あるいはアンダーフローする結果が出る場合には、`errno` が `ERANGE` に設定されます。さらに、例外の詳細を含むレコード付きで関数 `matherr()` が呼び出されます。このことは、UNIX が開発された当初に対象としたマシンにはほとんど負担をかけませんが、基本的な演算 `+`、`-`、`*`、`/` における一般的な例外がまったく未指定であるため、価値はそれ相応に小さいものとなります。

IEEE 754 の歴史

IEEE 規格は、以前の実装との互換性は目標でなかったと明白に述べています。代わりに、効率とユーザーの要求内容とを念頭に置いて例外処理の機構が開発されました。この機構は、単純な有理演算 (`+`、`-`、`*`、`/`) と、さらに複雑な剰余、平方根、フォーマット間変換などの演算との両方を通じて一様です。規格では超越関数については規定していませんが、規格の創設者は、準拠システム内の基本超越整関数にも同じ例外処理機構が適用されることを期待していました。

IEEE 例外処理の要素には、あらかじめ要求された場合にのみ、適当なデフォルト結果と計算の中断が含まれます。

SVID の将来の方向

現在の SVID (第 3 版または SVR4) では、将来の発展について一定の方向が明らかにされています。方向の 1 つは IEEE 規格との互換性です。特に、SVID の将来のバージョンにより、大きな有限数用に用意された `HUGE` への参照が、IEEE システム上での無限大である `HUGE_VAL` で置換されます。たとえば `HUGE_VAL` は、浮動小数点オーバーフローの結果として返されます。例外を発生させる入力引数について `libm` 関数が返す値は、後出の表 E-1 の IEEE 欄に示すものとなります。`errno` は今後設定される必要がなくなります。

SVID の実装

以下の表に示す libm 関数により、SVID に対応するオペランドチェックまたは結果チェックが行われます。`-xlibmil` 経由で libm のインライン展開テンプレートを使用する C プログラムから呼び出されたとき、平方根用のハードウェア命令 `fsqrt[sd]` が関数コールの代わりに使用されるため、`sqrt` 関数は SVID に準拠しない唯一の関数です。

表 E-1 例外のケースと libm 関数

関数	errno	エラーメッセージ	SVID	X/Open	IEEE*
<code>acos(x >1)</code>	EDOM	DOMAIN	0.0	0.0	NaN
<code>acosh(x<1)</code>	EDOM	DOMAIN	NaN	NaN	NaN
<code>asin(x >1)</code>	EDOM	DOMAIN	0.0	0.0	NaN
<code>atan2(+ -0, + -0)</code>	EDOM	DOMAIN	0.0	0.0	+ -0.0, + -pi
<code>atanh(x >1)</code>	EDOM	DOMAIN	NaN	NaN	NaN
<code>atanh(+ -1)</code>	EDOM/ERANGE	SING	+ -HUGE (EDOM)	+ -HUGE_VAL (ERANGE)	+ -infinity
<code>cosh overflow</code>	ERANGE	-	HUGE	HUGE_VAL	infinity
<code>exp overflow</code>	ERANGE	-	HUGE	HUGE_VAL	infinity
<code>exp underflow</code>	ERANGE	-	0.0	0.0	0.0
<code>fmod(x, 0)</code>	EDOM	DOMAIN	x	NaN	NaN
<code>gamma(0 or -integer)</code>	EDOM	SING	HUGE	HUGE_VAL	infinity
<code>gamma overflow</code>	ERANGE	-	HUGE	HUGE_VAL	infinity
<code>hypot overflow</code>	ERANGE	-	HUGE	HUGE_VAL	infinity
<code>j0(x > X_TLOSS)</code>	ERANGE	TLOSS	0.0	0.0	correct answer
<code>j1(x > X_TLOSS)</code>	ERANGE	TLOSS	0.0	0.0	correct answer
<code>jn(x > X_TLOSS)</code>	ERANGE	TLOSS	0.0	0.0	correct answer
<code>lgamma(0 or -integer)</code>	EDOM	SING	HUGE	HUGE_VAL	infinity

* `-xc99=lib` を設定すると、例外の動作として IEEE 754スタイルの戻り値が強制的に返されます (例: `errno` が設定されずに例外が起こります)。

表 E-1 例外のケースと libm 関数 (続き)

関数	errno	エラーメッセージ	SVID	X/Open	IEEE*
lgamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
log(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (E RANGE)	infinity
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (E RANGE)	-infinity
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
loglp(-1)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (E RANGE)	-infinity
loglp(x<-1)	EDOM	DOMAIN	NaN	NaN	NaN
pow(0,0)	EDOM	DOMAIN	0.0	1.0 (no error)	1.0 (no error)
pow(NaN,0)	EDOM	DOMAIN	NaN	NaN	1.0 (no error)
pow(0,neg)	EDOM	DOMAIN	0.0	-HUGE_VAL	+infinity
pow(neg, non-integer)	EDOM	DOMAIN	0.0	NaN	NaN
pow overflow	ERANGE	-	+HUGE	+HUGE_VAL	+infinity
pow underflow	ERANGE	-	+0.0	+0.0	+0.0
remainder(x,0)	EDOM	DOMAIN	NaN	NaN	NaN
scalb overflow	ERANGE	-	+HUGE_VAL	+HUGE_VAL	+infinity
scalb underflow	ERANGE	-	+0.0	+0.0	+0.0
sinh overflow	ERANGE	-	+HUGE	+HUGE_VAL	+infinity
sqrt(x<0)	EDOM	DOMAIN	0.0	NaN	NaN
y0(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
y0(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y0(x > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	correct answer
y1(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
y1(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y1(x > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	correct answer

* -xc99=lib を設定すると、例外の動作として IEEE 754スタイルの戻り値が強制的に返されます (例: errno が設定されずに例外が起こります)。

表 E-1 例外のケースと libm 関数 (続き)

関数	errno	エラーメッセージ	SVID	X/Open	IEEE*
yn(n, 0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-infinity
yn(n, x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
yn(n, x> X_TLOSS)	ERANGE	TLOSS	0.0	0.0	correct answer

* -xc99=lib を設定すると、例外の動作として IEEE 754スタイルの戻り値が強制的に返されます (例: errno が設定されずに例外が起こります)。

例外のケースと libm 関数についての一般的注意事項

表 E-1 には、各規格の影響を受ける libm 関数がすべてリストされています。値 X_TLOSS は、<values.h> に定義されています。SVID では、<math.h> に対し HUGE を MAXFLOAT と定義するように要求しています。これはおよそ $3.4\text{e}+38$ です。HUGE_VAL は libc で無限大と定義されています。errno は、C プログラムおよび C++ プログラムにアクセス可能なグローバル変数です。

<errno.h> では、errno 用として考えられる値を 120 個程度定義しています。数学ライブラリが使用するものが 2 つあります。ドメインエラー用の EDOM と範囲エラー用の ERANGE です。intro(3) と perror(3) を参照してください。

- -xc99=lib の設定では、libm 関数は、例外的なケースに関して -xlibmieee が指定されたかのように動作します。たとえば、「例外的なケースにおいて数学ルーチンについて IEEE 754スタイルの戻り値が強制的に返されます。」一般的注意事項に記載されている -Xs、-Xt、-Xa あるいは -Xc によって示されるライブラリの動作はすべて有効ではなくなるということも意味します。

注 – /usr/lib で Solaris が提供する標準システムのライブラリはまだ 1999 ISO/IEC C 規格に準拠していません。準拠するまでは -xc99=lib を設定することはできません。

- ANSI C コンパイラは、-Xt、-Xa、-Xc、-Xs を切り換え、特にコンパイラによって実施される規格準拠レベルを管理します。これらのスイッチの詳細については、cc(1) を参照してください。

- `libm` に関する限り、`-xt` と `-xa` によってそれぞれ `SVID` と `X/Open` が動作します。`-xc` は厳密な `ANSI C` 動作に相当します。

付加スイッチ `-libmieee` は、指定されると、`IEEE 754` に従って値の返却を行います。`libm` と、`libsunmath` のデフォルト動作は、`Solaris 2.6`、`Solaris 7` および `Solaris 8` 上では `SVID` 準拠になっています。

- 厳密な `ANSI C` (`-xc`) の場合、`errno` は必ず設定され、`matherr()` の呼び出しは行われず、`X/Open` 値が返されます。
- `SVID` (`-xt` または `-xs`) の場合、関数 `matherr()` が例外についての情報付きで呼び出されます。この情報にはデフォルトの `SVID` 戻り値となる値が含まれています。

ユーザーが与えた `matherr()` により、戻り値が変化することもあります。`matherr(3m)` を参照してください。ユーザーが与えた `matherr()` がなければ、`libm` は `errno` の設定を行い、メッセージを標準エラー出力ファイルに出力し、265 ページの表 E-1 内で `SVID` 欄に示されている値を返します。

- `X/Open` (`-xa`) の場合、動作は `matherr()` が起動され、それに合わせて `errno` が設定される点で `SVID` の場合と同じです。ただし、標準エラー出力ファイルにはエラーメッセージの出力はなく、また多くの場合、`X/Open` の戻り値は `IEEE` の戻り値と同じです。
- `libm` の例外処理に対して、`-xs` は `-xt` と同じ動作を行います。すなわち、`-xs` でコンパイルされたプログラムは、表 E-1 に示されている `libm` 関数の `SVID` 準拠バージョンを使用します。
- `sqrt` が負の引数に出会った場合、インラインハードウェア浮動小数点付きでコンパイルされたプログラムは、効率上の観点から `EDOM` を設定したり `matherr()` を呼び出したりするため必要となる特別なチェックは行いません。そのままでは `EDOM` が設定される可能性がある場合には、関数値用に `NaN` が返されます。

`sqrt()` 関数コールを `fsqrt[sd]` 命令で置換する C プログラムは、`IEEE` 浮動小数点規格には準拠しますが、`System V Interface Definition` のエラー処理要求には今後準拠しなくなる可能性があります。

libm についての注意

SVID では、PLOSS (Partial Loss of Significance) と TLOSS (Total Loss of Significance) の 2 つの浮動小数点例外を規定しています。sqrt(-1) と異なり、これらには固有の数学的意味がありません。また、exp(+10000) と異なり、これらは浮動小数点記憶形式の固有の制限を反映しません。

その代わり PLOSS と TLOSS は、fmod 用の特定アルゴリズム、および明確な境界により不意に正確度が落ちる三角関数用の特定アルゴリズムの制限を反映します。

libm のアルゴリズムは、ほとんどの IEEE の実装と同様、そのような不意の低下を被ることがなく、PLOSS のシグナルを出すことはありません。SVID 準拠要件を満たすために、ベッセル関数では、正確な結果を安全に計算できるにもかかわらず、大きい入力引数については TLOSS のシグナルを出します。

sin、cos、および tan の実装では、無限大での本質的な特異点を、無限引数について EDOM を設定して NaN を返すことにより、ほかの本質的な特異点と同様に処理します。

同様に SVID では、 x/y がオーバーフローすると考えられる場合には、fmod(x, y) は 0 でなくてはならないことを規定しています。しかし、IEEE 剰余関数から派生した fmod の libm の実装では、 x/y を明示的に計算せず、必ず厳密な結果をもたらします。

LIA-1 準拠

ここでは、LIA-1 は ISO/IEC 10967-1:1994 Information Technology Language Independent Arithmetic Part 1 のことを差します。言語に依存しない数値演算についての基準です。

C コンパイラ (cc) および Fortran 95 コンパイラ (f95) は、Compiler Collection のコンパイラに含まれており、以下の点で LIA-1 に準拠しています。行頭のアルファベットは、LIA-1 の第 8 節で使用されているものと対応しています。

a. データ型 (LIA 5.1)

LIA-1 準拠している型には、C の int および FORTRAN の INTEGER があります。この他にも LIA-1 準拠の型はありますが、ここでは取り上げません。その他特定の言語に対する仕様は、言語が LIA-1 に準拠してから決められます。

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.)
```

```
#include <values.h> defines MAXINT
```

d. DIV/REM/MOD (5.1.3)

C の / および % と FORTRAN の / および mod() によって、DIVtI(x,y) と REMtI(x,y) が提供されます。また、modaI(x,y) は、以下のコードによって使用できます。

```
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;
}
```

または

```
integer function modaI (x, y)
integer x, y, t
t = mod(x,y)
if (y .lt. 0 .and. t .gt. 0) t = t - y
if (y .gt. 0 .and. t .lt. 0) t = t + y
modaI = t
return
end
```

i. 記数法 (LIA 5.1.3)

LIA-1 の整数演算で認識される記数法を示します。

表 E-2 LIA-1 準拠の記数法

LIA	C	FORTRAN (C と異なる場合)
addI (x,y)	x+y	
subI (x,y)	x-y	
mulI (x,y)	x*y	
divtI (x,y)	x/y	
remtI (x,y)	x%y	mod (x,y)
modaI (x,y)	x%y	mod (x,y)
negI (x)	-x	
absI (x)	#include <stdlib.h> abs(x)	abs (x)
signI (x)	#define signI ((x) (x>0?1:(x<0?-1:0))	以下を参照
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

signI (x) の FORTRAN でのコード例を示します。

```
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
```

j. 式の評価

デフォルトでは、最適化が指定されていない場合は、式は C の場合は `int`、**FORTRAN** の場合は `INTEGER` の精度で評価されます。括弧も評価されます。`a+b+c` または `a*b*c` などの、括弧で囲まれていない結合式の評価順序は、指定されていません。

k. パラメータの受け取り方

ソースコード中にある上記の定義をインクルードします。

n. 通知

整数の例外は `x/0`、`x%0`、`mod(x,0)` です。デフォルトでは、これらの例外が `SIGFPE` を生成します。シグナルハンドラが指定されていない場合は、プロセスを終了してメモリーダンプを行います。

o. 選択のしくみ

`signal(3)` および `signal(3F)` が使用され、ユーザーが `SIGFPE` に対する例外処理を行うことができるようになります。

参考文献

以下の SPARC[®] マニュアルからは、SPARC 浮動小数点ハードウェアについてさらに情報が得られます。

『SPARC Architecture Manual Version 9』。PTR Prentice Hall。New Jersey。1994 年。

上記以外の参考文献は関連する章別に示します。規格の文書と試験プログラムを入手する方法についての情報もこの付録の最後に加えられています。

第 2 章「IEEE 演算機能」

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, Vol 5。1984 年。887-919 ページ

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., Coonen, J.T. 共著『The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming environments』。これは『The Relationship between Numerical Computation and Programming Languages』 Reid, J.K.(編集主任)の一環として発行されました。North-Holland Publishing Company。1982 年。

Kahan, W. 著『Implementation of Algorithms』 Computer Science Technical ReportNo. 20. University of California, Berkely CA. 1973 年。注文先 : National Technical Information Service. NTIS 文書番号 AD-769 124 (339 ページ)。1-703-487-4650 (ordinary orders) または 1-800-336-4700 (rush orders)。

Karpinski, R. 著『Paranoia: a Floating-Point Benchmark』 Byte 誌の 1985 年 2 月号。

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 Newsletter 16. 1981 年。

Rump, S.M. 著『How Reliable are Results of Computers?』。これは次の書籍の翻訳です。『Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?』 Jahrbuch Überblicke Mathematik 1983. 163-168 ページ。C Bibliographisches Institut AG 1984。

Sterbenz 著『Floating-Point Computation』。Prentice-Hall. 1974 年。(絶版ですが、ほとんどの大学図書館の蔵書になっています。)

Stevenson, D. 他、Cody, W., Hough, D., Coonen, J. 著。2 進浮動小数点演算機能についての規格草案を提案および分析している各種論文。IEEE Computer の 1981 年 3 月号。

『The Proposed IEEE Floating-Point Standard』 ACM SIGNUM Newsletter の特別号。1979 年 10 月。

第 3 章「数学ライブラリ」

Cody, William J., Waite, William 共著『Software Manual for the Elementary Functions』 Prentice-Hall, Inc 発行。Englewood Cliffs, New Jersey, 07632. 1980 年。

Coonen, J.T. 著『Contributions to a Proposed Standard for Binary Floating-Point Arithmetic』 博士号論文。University of California, Berkeley. 1984 年。

Tang, Peter Ping Tak 著『Some Software Implementations of the Functions Sin and Cos』 技術レポート 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』前刷り 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。communication。1988 年 7 月 18 日。

Tang, Peter Ping Tak 著『Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic』前刷り 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., Miller, Keith W. 共著。『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。

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., X. Li 共著『Faster Numerical Algorithms via Exception Handling』IEEE Trans. Comput. 第 48 巻、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。

付録 B「SPARC の動作と実装」

以下のドキュメントから、浮動小数点ハードウェアとメインプロセッサチップについての詳細情報が得られます。システムアーキテクチャ別に編成してあります。

Texas Instruments『SN74ACT8800 Family, 32-Bit CMOS Processor Building Blocks: Date Manual』第 1 版。Texas Instruments Incorporated。1988 年。

Weitek 『WTL 3170 Floating Point Coprocessor: Preliminary Data』。1988 年。Weitek Corporation。1060 E. Arques Avenue, Sunnyvale, CA 94086。

Weitek 『WTL 1164/WTL 1165 64-bit IEEE Floating Point Multiplier/Divider and ALU: Preliminary Data』。1986 年。Weitek Corporation。1060 E. Arques Avenue, Sunnyvale, CA 94086。

PowerPC 603 RISC Microprocessor User's Manual, Motorola, Inc., 1994 年。

規格書

American National Standard for Information Systems - ISO/IEC 9899:1990
Programming Language-C American National Standards Institute。1430 Broadway,
New York, NY 10018。

IEEE Standard for Binary Floating-Point Arithmetic。ANSI/IEEE Std 754-1985 (IEEE 754)。The Institute of Electrical and Electronics Engineers, Inc. 発行。345 East 47th Street, New York, NY 10017。1985 年。

IEEE Standard Glossary of Mathematics of Computing Terminology, ANSI/IEEE Std 1084-1986。The 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 Std 1003.1-1988。The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017。

System V Application Binary Interface (ABI)。AT&T (1-800-432-6600)。1989 年。

SPARC System V ABI Supplement (SPARC ABI)。AT&T (1-800-432-6600)。1990 年。

System V Interface Definition, 3rd Edition, (SVID89, SVID Issue 3)。Volume I-IV。Part Number 320-135。AT&T (1-800-432-6600)。1989 年。

X/OPEN Portability Guide。7 巻。Prentice-Hall, Inc. Englewood Cliffs, New Jersey 07632。1989 年。

試験プログラム

Paranoia、Z.Alex Liu による Berkley Elementary Function 試験プログラム、IEEE 試験ベクトル、Prof.W.Kahan による数値論理法 (丸めの乗算、除算、平方根を正確に算出するためのハードテストケースを生成する) などの、浮動小数点演算および数学ライブラリ用の試験プログラムを、ucbtest パッケージの Netlib から入手することができます。

ucbtest の入手先は、<http://www.netlib.org/fp/ucbtest.tgz> です。

用語集

この用語集は、コンピュータの浮動小数点演算機能に関する用語を中心にしています。並列処理に関連する用語についても説明しています。

注 – || の付いた用語は並列処理と関連のあるものです。

2 の補数

2 進数の真の補数。1 から各桁を差し引き、次に最下位桁に 1 を加え、必要な桁上げを行なって作ります。たとえば 1101 の 2 の補数は 0011 になります。

binade

連続した 2 つの 2 の累乗値の間隔。

IEEE 規格 754

Institute of Electrical and Electronics Engineers が発展させた 2 進浮動小数点演算機能の規格。1985 年に公表されました。

IPC ||

プロセス間通信³⁴を参照。

LWP ||

軽量プロセス³⁴を参照。

MBus ||

MBus は、プロセッサ、メモリー、I/O 相互接続に関するバス仕様です。相互運用 CPU モジュール、I/O インタフェースやメモリーコントローラなどを製造する複数のベンダーに対しては、MBus 仕様が SPARC International によってライセンス提供されています。MBus は、読み取り要求と応答を単一のバス上で結合する回路交換プロトコルです。MBus level I では単一プロセッサ信号が定義され、MBus level II では write-invalidate キャッシュの一貫性機構のためのマルチプロセッサ拡張が定義されています。

MIMD ||

複数命令複数データ (MIMD)、共有メモリーアーキテクチャを参照。

mt-safe ||

Solaris 環境では、ライブラリ内の関数は **mt-safe** であるかそうでないかのいずれかです。**mt-safe** コードは「再入可能」コードとも呼ばれます。すなわち、複数のスレッドが単一のモジュール内で同時に特定の関数を呼び出すことができ、それを制御するのは関数コードです。複数のスレッド間で共有されるデータはモジュール関数によってのみアクセスされることが前提となっています。モジュールのクライアントが可変の大域データを利用できる場合は、インタフェース内で適切なロックが利用可能になっていなければなりません。また、クライアントが適切なタイミングでロックを一貫的に使用できない場合は、モジュール関数を再入可能にすることはできません。シングルロック戦略も参照してください。

mutex ロック ||

相互排他機構を実装するための同期変数。条件変数、相互排他も参照。

NaN

Not a Number (数ではない、非数) の略。浮動小数点形式で符号化された記号エンティティ。

SIMD ||

単一命令複数データを参照。

SISD ||

単一命令単一データを参照。

SPMD ||

単一プログラム複数データを参照。

stderr

標準エラー (Standard Error) は、標準エラー出力を指す UNIX の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されたときにオープンされます。

ulp

unit in last place (最後の位の単位) の略。2 進形式では、仮数の最下位ビット、すなわちビット 0 が最後の位の単位です。

ulp(x)

作業形式で切り捨てられた x の ulp を表わします。

write-back ||

キャッシュと主メモリー間の一貫性を維持するための書き込み方針。

write-back 方針 (copy back または store in と呼ばれる) では、ローカルキャッシュ内のブロックに対する書き込みだけが行われます。書き込みは、キャッシュメモリーと同じ速度で発生します。変更されたキャッシュのブロックは、対応するメモリーアドレスが別のプロセッサによって参照されるときだけ、主メモリーに書き込まれます。プロセッサはキャッシュブロック内部に何度でも書き込みできますが、主メモリーに対する書き込みは参照されたときに限られます。すべてのデータがメモリーに書き込まれるわけではないため、write-back 方針ではバス帯域幅に関する要件が緩和されます。キャッシュ、一貫性、write-through も参照してください。

write-invalidate ||

書き込みが発生するまでローカルキャッシュからの読み取りを行うことによって、キャッシュの一貫性を維持する方針。変数の値を変更する場合、書き込み側のプロセッサは、まず最初にほかのキャッシュ内にある変数のすべてのコピーを無効にします。これで、書き込み側のプロセッサは、別のプロセッサが変数を要求するまで、変数のローカルコピーを自由に更新することができます。書き込み側のプロセッサはバス上に無効化シグナルを発行し、キャッシュ内に変数のコピーが存在するかどうかを検査します。コピーが存在する場合、そのワードを含むブロックが無効にされます。このような仕組みのために、複数の読み取りが許可されますが、書き込みできるのは単一の

プロセッサに限られます。**write-invalidate** 方針では、他のコピーを無効にするために最初の書き込みだけでバスが使用されます。その後のローカルな書き込み操作ではバス上のトラフィックは発生しないため、バス帯域幅に関する要件が緩和されます。キャッシュ、キャッシュのローカル性、一貫性、見せかけの共有、**write-update** も参照してください。

write-through ||

キャッシュと主メモリー間の一貫性を維持するための書き込み方針。**write-through** 方針 (**store through** と呼ばれる) では、ローカルキャッシュ内のブロックだけでなく主メモリーに対する書き込みも行われます。この方針には、主メモリーにデータの最新コピーが維持されるという利点があります。キャッシュ、一貫性、**write-back** も参照してください。

write-update ||

write-update 方針 (**write-broadcast** と呼ばれる) では、すべてのキャッシュ内にある共有変数のコピーを即時更新することによって、キャッシュの一貫性を維持します。すべての書き込みはバスを通じて共有データのコピーを更新するため、この方針は **write-through** の一形式と言えます。**write-update** 方針には、新しい値がキャッシュに早めに反映されるため、待ち時間が短くなるという利点があります。キャッシュ、キャッシュのローカル性、一貫性、見せかけの共有、**write-invalidate** も参照してください。

XDBus ||

XDBus 仕様では、長いバックプレーンを高いクロック率で駆動するために、低インピーダンスの GTL (Gunning Transceiver Logic) トランシーバシグナルが使用されます。XDBus では複数のインタリーブメモリーのバンクを備えた数多くの CPU が使用できるため、処理能力が増強されます。この仕様では、バスを効率的に利用するために、分割要求／応答によるパケット交換プロトコルが使用されます。また、インタリーブ機構も定義されているため、1、2、4 個の独立したバスデータパスを単一のバックプレーンとして使用して、処理能力を高めることができます。XDBus は、**write-invalidate**、**write-update**、競合的キャッシングによる一貫性維持方式をサポートしています。また、いくつかの競合制御機構も用意されています。キャッシュ、一貫性、競合的キャッシング、**write-invalidate**、**write-update** も参照してください。

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目的の浮動小数点形式で正規数として表現できない場合に発生する状態。

一貫性 ||

複数のキャッシュを持つシステムにおいて、すべてのプロセッサが常に同一のメモリーイメージを参照できるようにする機構。

インラインテンプレート ト

Compiler Collection のコンパイラのインラインパス中に、定義した関数コールと置換さ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などの基本関数のハードウェア実装にアクセスするときに、インラインテンプレートファイル (libm.il) 内の数学ライブラリによって使用されます。

回路交換 ||

キャッシュと主メモリー間、および複数のキャッシュ相互間で通信を行うための機構。複数のキャッシュ間、あるいはキャッシュと主メモリー間で、専用のコネクション (回路) が確立されます。回路が活動中の間は、バスを使用した他の通信は行えません。

隠しビット

ハードウェアが丸めを正確に行うために実行時に使用する、ソフトウェアからはアクセスできない特別なビット。たとえば IEEE 倍精度演算では、56 ビットの結果を算出した後、それを 53 ビットに丸めるために 3 つの隠しビットを使用します。

完全連想キャッシュ ||

m 個のエントリを持つ完全連想キャッシュは、 m 方向のセット連想キャッシュです。すなわち、 m 個のブロックを含む単一のセットがあります。キャッシュのエントリは、そのセット内の m 個のブロックのうち任意のブロックに置くことができます。キャッシュ、キャッシュのローカル性、直接マップのキャッシュ、見せかけの共有、セット連想キャッシュ、write-invalidate、write-update も参照してください。

基数

数体系の基となる数。たとえば 2 は 2 進記数法の基数であり、10 は 10 進記数法の基数です。SPARC ワークステーションでは 2 を基数とする演算を採用しており、IEEE 規格 754 は、2 を基数とする演算規格です。

キャッシュ ||

プロセッサと主メモリー間のバッファとして動作する小型で高速のハードウェア制御メモリー。キャッシュには、一番最近使用された命令とデータのメモリー位置 (アドレスと内容) のコピーが格納されます。アドレスの参照時には、まず最初にキャッシュが検索されます。目的の命令やデータがキャッシュ内に存在しない場合は、キャッシュミスが発生します。データの内容はバスを通じて主メモリーから、実行中の命令で指定された CPU レジスタに取り込まれ、そのコピーがキャッシュにも書き込まれます。同じアドレスがすぐ後で使用される場合は、そのアドレスがキャッシュ内で見つかります (キャッシュヒットの発生)。そのアドレスへの書き込みが行われると、ハードウェアはキャッシュへの書き込みを行うだけでなく、主メモリーへの write-through 書き込みを生成する場合があります。

結合規則、回路交換、直接マップのキャッシュ、完全連想キャッシュ、MBus、パケット交換、セット連想キャッシュ、write-back、write-through、XDBus も参照してください。

キャッシュのローカル性 ||

プログラムは、コードやデータに対して同一の確率でアクセスするわけではありません。最近アクセスされたデータをキャッシュに保存しておけば、メモリーにアクセスしなくても必要なデータをローカルに検索できる確率が高まります。ローカル性の原則とは、特定の短い期間についてはプログラムが比較的狭いアドレス空間をアクセスすることを意味します。ローカル性には、一時的なものと空間的なものの 2 種類があります。

一時的なローカル性 (時間的なローカル性) とは、最近アクセスされたデータが再利用される傾向を指します。たとえば、大部分のプログラムにはループが含まれているため、命令やデータが反復的にアクセスされる傾向があります。一時的なローカル性では、メモリーへのアクセスを避けるために、最

近アクセスされたデータをプロセッサに近いキャッシュ内に保存します。キャッシュ、競合的キャッシング、見せかけの共有、**write-invalidate**、**write-update** も参照してください。

空間的なローカル性とは、最近アクセスされたデータとアドレスが近いほかのデータが参照される傾向を指します。たとえば、配列やレコードの要素に対するアクセスではこのような傾向が自然に見られます。キャッシングで空間的ローカル性を利用するためには、メモリーからキャッシュにブロック(複数の連続したワード)を移動してプロセッサに近づけます。キャッシュ、競合的キャッシング、見せかけの共有、**write-invalidate**、**write-update** も参照してください。

競合的キャッシング ||

競合的キャッシングでは、**write-invalidate** と **write-update** の組み合わせを使用して、キャッシュの一貫性を維持します。競合的キャッシングでは、共有データをエージングするためのカウンタが使用されます。共有データは、使用時期がもっとも古いものから (LRU:Least-Recently-Used アルゴリズムにもとづいて) キャッシュから削除されます。これにより、共有データは私的なデータに戻るため、キャッシュ一貫性プロトコルが (バックプレーン帯域を通じて) メモリーにアクセスし、同期化された複数のデータコピーを維持する必要がなくなります。キャッシュ、キャッシュのローカル性、見せかけの共有、**write-invalidate**、**write-update** も参照してください。

共通の例外

3つの浮動小数点例外 (オーバフロー、無効な演算、およびゼロ除算) は、**ieee_flags(3m)** と **ieee_handler(3m)** のために「共通の例外」と総称されています。エラー時に共通してトラップされるのでこのように呼ばれます。

共有メモリーアーキテクチャ ||

バスで接続されたマルチプロセッサシステムでは、すべてのプロセッサによって共有される大域メモリーを通じてプロセスまたはスレッドが通信を行います。この共有データセグメントは、協同するプロセスのアドレス空間内で専用データとスタックセグメントの間に配置されます。**fork()** によって生成されるそれ以降のタスクのアドレス空間には、共有データセグメント以外のすべてのものがコピーされます。共有メモリーを使用するには、プログラム言語の拡張機能と専用のライブラリルーチンが必要です。

軽量プロセス ||

Solaris のスレッドはユーザーレベルのライブラリとして実装されており、軽量プロセス (LWP) と呼ばれるカーネルの制御スレッドが使用されます。Solaris 環境では、プロセスはメモリーを共有する LWP の集合です。各 LWP は UNIX プロセスのスケジューリング優先順位を持ち、そのプロセスの資源を共有します。LWP はロックなどの同期機構を使用して、共有メモリーに対するアクセスを調整します。LWP はコードやシステムコールを実行する仮想 CPU と考えることができます。スレッドライブラリは、カーネルがプロセッサのプール上にある LWP をスケジュールするのと同様にして、プロセス内の LWP のプール上にあるスレッドをスケジュールします。各 LWP はカーネルによって別々に振り分けられ、独立したシステムコールを実行し、独立したページフォルトを発生させ、マルチプロセッサシステム上で並列的に動作します。カーネルは、LWP のスケジューリングクラスと優先順位に従って、利用可能な CPU 資源を LWP に割り当てます。

結合規則 ||

キャッシュ、直接マップキャッシュ、完全連想キャッシュ、セット連想キャッシュを参照。

結合スレッド ||

Solaris では、特定の軽量プロセス (LWP) に恒久的に割り当てられたスレッドが結合スレッドと呼ばれます。結合スレッドは、プロセス内部だけではなくシステム内で活動状態のスレッドすべてに対して厳密な優先順位をもってリアルタイムベースでスケジュールすることができます。LWP は、スケジュールに関して他の UNIX プロセスと同じデフォルト優先順位でスケジュールされるエンティティです。

コンテキストスイッチ

SunOS オペレーティングシステムのようなマルチタスク・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。その時間の終わりで、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、新しいプロセスを実行する準備をします。CPU は、古いほうのプロセスについてレジスタを保存し、それから新しいプロセスのレジスタをロードします。古いプロセスから新しいプロセスに切り換えることをコンテキストスイッチと言います。コンテキストを切り換えるのに消費される時間

は、システムのオーバーヘッドです。所要時間は、レジスタの個数と、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別な命令があるかどうかで決定されます。

シグナルを発生しない NaN

例外を発生させることなく、ほとんどの算術演算を通じて伝達される NaN (非数)。

シグナルを発生する NaN

オペランドとして現われるたびに無効な演算例外を発生させる NaN (非数)。

指数

浮動小数点の構成要素のうち、表現された数値を決定するため基数を累乗する際の整数べきを示している要素。

順次プロセス ||

1 つのプロセスが終わってから次のプロセスが開始されるような方法で実行されるプロセス。多重プロセス、プロセスも参照してください。

条件変数 ||

Solaris スレッドに対して条件変数を適用すれば、条件が満たされるまでスレッドを不可分 (**atomic**) にブロックすることができます。条件は、**mutex** ロックの保護のもとで検査されます。条件が偽である場合、プロセスは条件変数上でブロックし、**mutex** ロックを解放して条件が変化するのを待ちます。別のスレッドが条件を変更すると、そのスレッドによって条件変数にシグナルが送信されるため、待機中のスレッドが起き上がり、**mutex** を再取得して条件を再評価します。条件変数を使用すれば、同一プロセス内のスレッドと他のプロセスを同期させることができます。ただし、条件変数を書き込み可能なメモリー内に割り当てられており、関連するプロセス間で共有され、上記の動作用に初期設定されている必要があります。

シングルロック戦略 ||

シングルロック戦略では、アプリケーション内のスレッドが実行されると、そのスレッドはアプリケーション全体に対する単一の **mutex** ロックを取得し、ロックを解放してからブロックします。シングルロック戦略では、単一

のロックに対する同期を取るために、システム内のすべてのモジュールとライブラリによる協同が必要です。特定の時間に共有データにアクセスできるスレッドは1つに限られるため、各スレッドはメモリーを整合的に参照します。共有メモリーが整合性のある状態に設定されてからロックが解放されること、およびほかのスレッドを実行するのに十分な頻度でロックが解放されるという条件が満たされれば、シングルロック戦略は単一プロセッサシステムでは非常に有効です。また単一プロセッサシステムでは、I/O 操作中にロックが設定されたままだと、多重性の効果が低下します。マルチプロセッサシステムでは、シングルロック戦略は適用できません。

スヌーピング ||

キャッシュの一貫性を維持するためのもっとも一般的なプロトコルは、スヌーピングと呼ばれます。キャッシュコントローラはバスを監視 (スヌープ) して、共有ブロックのコピーがキャッシュに含まれているかどうかを確定します。

読み取りの場合は、別個のプロセッサのキャッシュ内に複数のコピーが存在する場合がありますが、プロセッサは最新のコピーを必要とするため、すべてのプロセッサは書き込み後の新しい値を取得しなければなりません。

キャッシュ、競合的キャッシング、見せかけの共有、**write-invalidate**、**write-update** も参照してください。

書き込みの場合は、キャッシュへの書き込みに関する排他的アクセス権をプロセッサが持っていなければなりません。非共有ブロックに対する書き込みを行なっても、バス上のトラフィックは発生しません。共有データに対する書き込みを行うと、その他のすべてのコピーが無効になるか、書き込まれた値で共有コピーが更新されます。キャッシュ、競合的キャッシング、見せかけの共有、**write-invalidate**、**write-update** も参照してください。

スピンロック ||

スレッドは、別のタスクによってロックが解放されるまで、スピンロックを使用してロック変数を何度も検査します。すなわち、待機中のスレッドは、ロックが解除されるまで、ロック上でスピン (空回り) します。待機中のスレッドは臨界領域の内部でロックを設定し、臨界領域内でのタスクが完了すると、スピンロックを解除して他のスレッドが臨界領域に入れるようにします。スピンロックと **mutex** ロックの違いは、保留中の **mutex** ロックを取得

しようとする LWP のブロックと解放が行われるのに対し、スピンロックでは LWP は解放されないという点です。mutex ロックも参照してください。

スレッド ||

単一の UNIX プロセスのアドレス空間内部での制御の流れ。Solaris スレッドでは軽量プロセス形式の多重タスクが提供されるため、スケジューリングと通信の負荷を最小限に抑えた上で、複数の制御スレッドを共通のユーザーアドレス空間に置くことができます。スレッドは、同一のアドレス空間、ファイル記述子 (1 つのスレッドによってオープンされたファイルは、他のスレッドによって読み取ることができます)、データ構造、オペレーティングシステム状態を共有します。スレッドには、ローカル変数を追跡してアドレスを戻すためのプログラムカウンタとスタックがあります。スレッドは、共有データとスレッド同期操作を通じて相互に会話をします。結合スレッド、軽量プロセス、マルチスレッド、非結合スレッドも参照してください。

正確度

ある数が別の数にどれだけ近いかを表す尺度。たとえば、計算結果の正確度は、計算誤差により数学的に正確な結果とどの程度の差がつくかをしばしば示します。正確度は、「結果は小数点第 6 位まで正確である」のように有効桁数で表現されたり、もっと一般的に「結果の算術符号は正しい」のように関連する数学的な特性の保持で表現されたりします。

正規数

IEEE 演算機能で、0 でもなく極大 (すべて 1) でもないバイアス指数を持つ数。上限下限の決まった小さな相対誤差を伴う、通常の範囲の実数の部分集合を表わしています。

制御フローモデル ||

フォン・ノイマンのモデルにもとづくコンピュータ。このモデルは、制御の流れ、すなわちプログラムの各ステップで実行される命令を指定するものです。サン・マイクロシステムズ社のすべてのワークステーションは、フォン・ノイマンのモデルにもとづいています。データフローモデル、要求方式のデータフローも参照してください。

精度

表現可能な数の記録密度の定量的測度。たとえば、53 個の有効ビットを持つ精度の 2 進浮動小数点形式では、(正規数の範囲において) 2 つの隣接した 2 の累乗の間には 253 個の表現可能な数があります。精度を、ある数が別の数にどれだけ近いかを示す正確度と混同しないでください。

セット連想キャッシュ

セット連想キャッシュでは、一定数 (少なくとも 2 通り) の位置に各ブロックを配置することができます。各ブロックを n 通りの位置に配置できるセット連想キャッシュは、 n 方向のセット連想キャッシュと呼ばれます。 n 方向のセット連想キャッシュは、それぞれ n 個のブロックからなる 2 つ以上のセットによって構成されます。ブロックは、各セット内の任意の位置 (要素) に配置することができます。連想レベル (セット内のブロック数) を増加させると、キャッシュのヒット率が高まります。キャッシュ、キャッシュのローカル性、見せかけの共有、`write-invalidate`、`write-update` も参照してください。

セマフォ

E.W.Dijkstra によって開発された特別な目的のデータ型で、特定の資源または共有資源の集合に対するアクセスを制御します。セマフォは整数値 (負にすることはできません) を持っていますが、この値に対しては 2 つの操作を実行できます。シグナル (V または `up`) 操作は値を 1 だけ増加させます。通常、これは資源が空き状態になったことを意味します。待機 (P または `down`) 操作は値を 1 だけ減少させます (負にすることはできません)。通常、これは空き状態の資源が使用されそうになっていることを示します。セマフォロックも参照してください。

セマフォロック

非同期スレッドを調整することによって、臨界資源に対するアクセスを制御するための同期機構。セマフォも参照してください。

ゼロ格納

算術演算の結果がアンダーフローしたとき、その結果をフラッシュしてゼロにすること。

相互接続ネットワーク トポロジ //

相互接続トポロジは、プロセッサの接続方法を記述したものです。すべてのネットワークは複数のスイッチから構成されますが、それらのスイッチのリンクはプロセッサメモリーのノードや他のスイッチに接続されます。トポロジには、スター、リング、バス、完全接続ネットワークという 4 種類の一般形式があります。スタートポロジは、単一のハブプロセッサとそれに直接接続された他のプロセッサから構成されます。ハブでないプロセッサは、直接には相互接続されていません。リングトポロジでは、すべてのプロセッサが単一のリング上にあり、通常はそのリング上で片方向の通信が行われます。バストポロジでは、すべてのノードが線状に接続されています。したがって、両方向の通信が行われ、特定の時間にバスを使用するプロセッサを決めるためには、なんらかの調整が必要です。完全接続 (クロスバー) ネットワークでは、各プロセッサがほかのプロセッサに対して両方向のリンクを持ちます。

一般に購入可能な並列プロセッサでは、マルチステージのネットワークトポロジが使用されています。マルチステージのネットワークトポロジの特徴は、2 次元のグリッドとブール型 n キューブです。

相互排他 //

多重実行環境において、特定のスレッドが他のスレッドと競合することなく臨界資源を更新できること。臨界領域、臨界資源も参照してください。

多重性 //

2 つ以上の活動状態のスレッドまたはプロセスを並列的に実行すること。単一プロセッサ環境では、複数のスレッドを高速で切り換えることによって見せかけの多重性が実現されます。マルチプロセッサシステム上では、真の並列実行を実現することができます。非同期制御、マルチプロセッサシステム、スレッドも参照してください。

多重プロセス //

複数のプロセッサ上で並列的に実行されるプロセス、あるいは単一のプロセッサ上で非同期的に実行されるプロセス。多重プロセスは相互に会話し、ほかのプロセスからの情報受信や外部イベントの発生を待つために、一時的に実行を中断する場合があります。プロセス、順次プロセスも参照してください。

段階的アンダーフロー

浮動小数点演算がアンダーフローすると、0 の代わりに非正規数が返されます。このアンダーフロー処理方式により、小さい数に対して浮動小数点演算を行なった場合に正確度の損失を最小に抑えます。

単一プログラム複数 データ (SPMD) ||

別個のデータの同時処理が独立して行われるような形式の非同期並列性。SPMD では、プロセッサが同時に別個の命令 (if-then-else 文の別個の分岐など) を実行する場合があります。

単一プロセッサシステム ||

特定の時間には単一のプロセッサだけが活動状態になるシステム。この単一のプロセッサは、伝統的な単一命令単一データのモデルだけでなくマルチスレッドアプリケーションを実行することもできます。マルチスレッド、単一命令単一データ、シングロック戦略も参照してください。

単一命令複数データ (SIMD) ||

数多くの処理要素が存在するが、それらが同一の命令を同時に実行するように指示されるシステムモデル。すなわち、単一のプログラムカウンタを使用して、プログラムの単一コピーが順次処理されます。正則数値計算のように、全体的に更新する必要があるデータを多く含む問題を解決する場合は、SIMD が特に効果的です。通常、科学計算やエンジニアリングのアプリケーション (画像処理、粒子シミュレーション、有限集合方式など) では SIMD パラダイムが使用されます。配列処理、パイプライン、ベクトル処理も参照してください。

単一命令単一データ (SISD) ||

単一のプロセッサを使用して、命令内で指定されたデータ項目を操作する一連の命令の取り出しと実行を行う、伝統的な単一プロセッサモデル。これは、フォン・ノイマンによる伝統的なコンピュータ処理モデルです。

単精度

コンピュータのワードを 1 つ使用して 1 つの数を表わすこと。

チェーニング

パイプライン式アーキテクチャのハードウェア機構であり、1つの操作が行われた場合、その結果が宛先レジスタへの書き込みと同時に別の操作でオペランドとして即時に利用できるようにするもの。チェーンされた2つの操作の合計サイクル時間は、両命令に対するスタンドアロンサイクル時間の和より短くなります。たとえば TI8847 では連続した `fadd`、`fsub`、`fmul` (精度は同じ) のチェーンをサポートしていますが、チェーンされた `faddd/fmULD` が必要とするサイクル数が 12 サイクルであるのに対し、チェーンされていない `faddd/fmULD` が連続している場合、17 サイクルを必要とします。

直接マップのキャッシュ ||

直接マップのキャッシュは、1方向のセット連想キャッシュです。すなわち、キャッシュには単一のブロックが格納され、単一の要素で単一のセットが形成されます。キャッシュ、キャッシュのローカル性、見せかけの共有、完全連想キャッシュ、セット連想キャッシュ、`write-invalidate`、`write-update` も参照してください。

データフローモデル ||

このコンピュータモデルはデータに対する処理を指定するもので、命令の順序は無視します。すなわち、計算処理は、命令の利用可能性ではなくデータ値の利用可能性にもとづいて先に進みます。制御フローモデル、要求方式のデータフローも参照してください。

データレース ||

マルチスレッド環境において、2つ以上のスレッドが共有資源に同時にアクセスする状況。スレッドが資源にアクセスする順序によっては、結果が不定になる場合があります。データレースと呼ばれるこのような状況では、同一の入力でプログラムを繰り返し実行しても、それぞれ異なる結果が得られる場合があります。相互排他、`mutex` ロック、セマフォロック、シングルロック戦略、スピンロックも参照してください。

デッドロック ||

2つ以上の独立した有効なプロセスが資源を要求して競合するときに発生する状況。たとえば、プロセス P が資源 X と Y をこの順序で要求すると同時に、プロセス Q が資源 Y と X をこの順序で要求すると想定します。プロセ

ス P が資源 X を取得すると同時に、プロセス Q が資源 Y を取得すると、どちらのプロセスも先に進めなくなります。その理由は、各プロセスが他方のプロセスに割り当てられた資源を必要とするためです。

デノーマル化数

非正規数の旧用語。

デフォルトの結果

例外を発生させた浮動小数点演算の結果としてもたらされる値。

トポロジ ||

相互接続ネットワークトポロジを参照。

配列処理³⁴ ||

同時に動作する複数のプロセッサ。配列のすべての要素に単一の操作を並列的に適用できるように、各プロセッサは配列の 1 つの要素を処理します。

バイアス指数

記憶された指数の範囲を負でなくするために選ばれた定数 (バイアス) と底が 2 の指数との和。たとえば、2-100 の指数は、IEEE 単精度形式では以下のように記憶されます。

$$(-100) + (127 \text{ の単精度バイアス}) = 27$$

倍精度

精度を維持・向上させるため、2 ワードを使用して 1 つの数を表わすこと。SPARC[®] ワークステーションでは、64 ビットの IEEE 倍精度方式です。

バックプレーン ||

MBus、マルチプロセッサバス、XDBus を参照。

バリア ||

データがアクセスされない場合でもタスクを調整するための同期機構。バリアはゲートの同義語です。並列実行されるプロセッサやスレッドは別々の時刻にゲートに到着しますが、すべてのプロセッサがゲートに到着するまではゲートを通過することはできません。たとえば、一日の終わりに銀行の出納係全員がその日に預け入れられた金額と引き出された金額の合計を計算しなければならないと想定します。これらの合計金額は銀行の副支店長に報告さ

れ、副支店長は借方総額と貸方総額が一致することを確認します。各出納係はそれぞれ独自のペースで自分の仕事を行います。すなわち、取引金額の計算が完了する時刻はまちまちです。バリア機構は、借方総額と貸方総額が一致するまでは各出納係が家に帰ることを禁止する役割を果たします。借方と貸方が合わない場合、各出納係は自分のデスクに戻って誤りを見つけなければなりません。満足のゆく計算結果が得られれば、バリアは取り除かれます。

パイプライニング Ⅱ

演算が複数の段階に変形されるハードウェア機構であり、各段階が完了するのに (通常の場合) 1 サイクルかかるものです。パイプラインはそれぞれのサイクルで新しい演算の発生が可能になると動作を始めます。パイプ中の各命令の間に依存関係がなければ、それぞれのサイクルで新しい結果をもたらすことができます。チェーニングは、依存しあっている命令同士のパイプライニングを意味します。依存しあっている命令同士をチェーンすることができなければ (たとえばハードウェアがそれらの命令のチェーニングをサポートしていない場合)、パイプラインは効果を示しません。

パイプライン Ⅱ

データに適用される機能全体が個々の処理フェーズに分割できる場合、データの個別部分は異なる処理フェーズ間を流れます。たとえば、コンパイラには、字句解析、構文解析、型検査、コード生成などのフェーズがあります。最初のプログラムやモジュールが字句解析のフェーズを通過すれば、それを構文解析のフェーズに渡すと同時に、2 番目のプログラムやモジュールに対する字句解析を開始することができます。配列処理、ベクトル処理も参照してください。

パケット交換 Ⅱ

共有メモリーアーキテクチャにおいて、キャッシュがほかのキャッシュおよび主メモリーと通信するための機構。パケット交換では、パケットと呼ばれる小さいセグメントにトラフィックが分割されます。パケットはバスに対して多重化されます。パケットには識別情報が含まれており、これによってキャッシュとメモリーハードウェアはパケットが自分自身に送信されたものか、そのパケットを最終的な宛先に転送するのかを確定することができます。パケット交換によって、バストラフィックを多重化すること、無秩序 (順序のない) パケットをバス上に配置することができます。無秩序パケットは、宛先 (キャッシュまたは主メモリー) で再組み立てされます。キャッシュ、共有メモリーも参照してください。

パラダイム Ⅱ

コンピュータによる問題解決を公式化するためのモデル。パラダイムは、実際の問題の理解と解決のためのコンテキストを提供します。パラダイムはモデルであるため、現実の問題の詳細部分を抽象化することによって、問題を解決しやすくします。ただし、ほかの抽象化モデルと同様に、パラダイムは現実の世界を近似するだけなので不正確になる場合もあります。複数命令複数データ、単一命令複数データ、単一命令単一データ、単一プログラム複数データも参照してください。

非結合スレッド Ⅱ

Solaris スレッドでは、LWP のプールに対してスケジュールされたスレッドは非結合スレッドと呼ばれます。スレッドライブラリは LWP の呼び出しと割り当てを行なって、実行可能スレッドを実行します。スレッドが同期機構 (mutex ロックなど) によってブロックされると、スレッドの状態がプロセスメモリに保存されます。その後、スレッドライブラリは別のスレッドを LWP に割り当てます。結合スレッド、マルチスレッド、スレッドも参照してください。

非正規数

IEEE 演算機能で、バイアス指数 0 を持つ 0 でない浮動小数点数。非正規数は、0 と最小ノーマル数の間にある各数です。

非同期制御 Ⅱ

特定のイベントが発生したという通知 (シグナル) を受け取った上で特定の操作を開始するようなコンピュータ制御方式。非同期制御では、ロックと呼ばれる同期機構に基づいて、複数のプロセッサを調整します。相互排他、mutex ロック、セマフォロック、シングロック戦略、スピンロックも参照してください。

複数命令複数データ (MIMD) Ⅱ

数多くのプロセッサがさまざまなデータに対して異なる命令を同時に実行できるようなシステムモデル。また、これらのプロセッサは、あたかも別個のコンピュータであるかのようにきわめて自律的に動作します。これらのプロセッサは中央コントローラを持たず、通常は互いに独立して動作します。これは銀行の日常業務に似ています。すなわち、各出納係は互いに相談することではなく、取引の各手順を同時に実行することはありません。データ使用上

の矛盾が発生しないかぎり、独自に各自の作業を行います。取引の処理は、顧客の注文やタイミングとは関係なく進められます。ただし、顧客 A と顧客 B が AB の共同預金口座を同時に利用することは避けなければなりません。MIMD ではロックと呼ばれる同期機構を使用して、共有資源に対するアクセスを調整します。相互排他、mutex ロック、セマフォロック、シングルロック戦略、スピンロックも参照してください。

複数読み取り単一書き込み ||

多重実行環境において、書き込みのためにデータにアクセスする最初のプロセスが排他的アクセス権を獲得し、多重書き込みアクセスや同時読み取り/書き込みアクセスを禁止すること。ただし、データの読み取りは、複数のプロセスに許可されます。

浮動小数点数体系

表現可能な数同士の間のスケーリングが固定しておらず、絶対的な定数でない実数の部分集合を表現するための体系。この体系は、底、符号、仮数、および指数 (通常バイアスされています) を要素としています。数値は、バイアスを除いた指数べきまで累乗した底と仮数との符号付き積です。

プロセス ||

単一の連続したスレッド、現在の状態、関連するシステム資源によって特徴づけられる活動単位。

プロセス間通信 ||

活動状態のプロセス間で転送されるメッセージ。回路交換、分散メモリアーキテクチャ、MBus、メッセージ転送、パケット交換、共有メモリー、XDBus も参照してください。

ブロック状態 ||

スレッドが資源やデータを待っている状態。たとえば、保留中のディスク読み取り要求からのデータや、ほかのスレッドが資源のロックを解除するのを待っている状態。

分散メモリアーキテクチャ //

相互接続ネットワークトポロジの各ノードにおけるローカルメモリーとプロセッサの組み合わせ。各プロセッサが直接アクセスできるのは、システムメモリーの一部分だけに限られます。2つのプロセッサ間の通信はメッセージ転送によって行われ、グローバルメモリーや共有メモリーは使用されません。したがって、データ構造を共有する必要があるとき、プログラムはその構造を所有するプロセスに対して送信/受信メッセージを発行します。プロセス間通信、メッセージ転送も参照してください。

並列処理 //

マルチプロセッサシステムでは、数多くのスレッドまたはプロセスを同時に活動状態にできる場合に真の並列処理が実現されます。多重性、マルチプロセッサシステム、マルチスレッド、単一プロセッサも参照してください。

並列性 //

多重プロセス、マルチスレッドを参照。

ベクトル処理 //

一連のデータを同一の方法で処理すること。通常、要素がベクトルである行列や配列データの操作に適用されます。ベクトル処理では、パイプライン処理を利用することができます。配列処理、パイプラインも参照してください。

マルチスレッド //

複数のスレッドまたはプロセッサを同時に活動状態にできるようなアプリケーション。マルチスレッドアプリケーションは、単一プロセッサシステムとマルチプロセッサシステムの両方で実行することができます。結合スレッド、`mt-safe`、シングロック戦略、スレッド、非結合スレッド、単一プロセッサも参照してください。

マルチタスク //

単一プロセッサシステムにおいて、数多くのスレッドが並列的に動作しているように見えること。これは、複数のスレッドを高速で切り換えることによって実現されます。

マルチプロセッサ Ⅱ

マルチプロセッサシステムを参照。

マルチプロセッサシステム Ⅱ

複数のプロセッサが同時に活動状態になれるようなシステム。別個のプロセスを実行する各プロセッサは、完全に非同期的に動作します。ただし、プロセッサが臨界システム資源やシステムコードの臨界領域にアクセスするときは、プロセッサ間の同期が重要になります。臨界領域、臨界資源、マルチスレッド、単一プロセッサシステムも参照してください。

マルチプロセッサバス Ⅱ

共有メモリーによるマルチプロセッサマシンでは、各 CPU とキャッシュモジュールがバスを通じて接続されます。メモリーと入出力もバスを通じて接続されます。バスはキャッシュ一貫性プロトコルを実現します。キャッシュ、一貫性、MBus、XDBus も参照してください。

丸め

厳密でない結果が生じた場合は、それに対し切り上げまたは切り捨て処理を行なって表現可能な値にしなければなりません。切り上げを行なった場合、結果は増大されて次の表現可能値になります。切り捨てを行なった場合は、縮小されて直前の表現可能値になります。

丸め誤差

実数が丸められて機械表現可能な数になる際に入ってくる誤差。ほとんどの浮動小数点演算に丸め誤差が伴います。IEEE 規格 754 では、1 回の浮動小数点演算について、その結果に複数の丸め誤差が伴うことは認められていません。

見せかけの共有 Ⅱ

2 つのスレッドによって独立にアクセスされる 2 つの関連のないデータが、同一のブロック内に存在する場合にキャッシュ内で発生する状態。このようなブロックは、正当な理由なく複数のキャッシュ間で「ピンポン」のように交換される場合があります。このような状態を検出してデータ構造を再編成し、見せかけの共有を除去すれば、キャッシュの性能が大幅に向上します。キャッシュ、キャッシュのローカル性も参照してください。

メッセージ転送 ||

分散メモリーアーキテクチャにおいて、プロセスが相互に通信するための機構。メッセージを格納するための共有データ構造はありません。メッセージ転送によって、プロセスはほかのプロセスにデータを送信し、受信側のプロセスは着信するデータと同期を取ることができます。

メモリー ||

後で取り出すための情報を保持できる媒体。通常、この用語は、マシン命令によって直接アドレス指定できるコンピュータの内部記憶を参照するために使用されます。キャッシュ、分散メモリーアーキテクチャ、共有メモリーも参照してください。

有効数字

浮動小数点数の値を決定するために基数の符号付きべき乗で乗算される浮動小数点の部分。正規数の有効数字は、小数点の左側のゼロでない1桁と、右側の小数から成ります。

要求方式のデータフロー ||

グラフ還元モデルなどのように、あるタスクの結果が実行可能なほかのタスクで必要な場合は、プロセッサによってそのタスクが実行可能にされます。グラフ還元プログラムは、計算が進むにつれて計算値によって置き換えられる可約式から構成されます。通常、還元は並列的に実行されます。並列的な還元の制約となるのは、以前の還元からのデータが利用可能かどうかだけです。制御フローモデル、データフローモデルも参照してください。

ラップ数

IEEE 演算機能で、そのままではオーバーフローしたり、アンダーフローしてしまう値に対し、ラップされた値が正規数の範囲に位置付けられるように、指数に固定オフセットを加えて作った数。ラップされた結果は、現在 SPARC ワークステーション上では生成されません。

臨界領域 ||

共有変数にアクセスするコードなどのように、単一のスレッドによって一度に実行され、ほかのスレッドによって割り込みされない分割不能なコード領域。相互排他、**mutex** ロック、セマフォロック、シングルロック戦略、スピンドロックも参照してください。

臨界資源 ||

一度に単一のスレッドだけで使用できる資源。複数の非同期スレッドが臨界資源を使用する必要がある場合は、同期機構によってそれぞれの要求を満たします。相互排他、**mutex** ロック、セマフォロック、シングルロック戦略、スピンロックも参照してください。

例外

不可分 (**atomic**) な算術演算を試みたが、ユニバーサルに受け入れ可能と考えられる結果が得られなかった場合には、演算例外が発生します。「不可分な (**atomic**)」と「受け入れ可能」の意味は、時と所に応じて変化します。

ロック ||

共有データに対するアクセスを順次処理するための方針を実施する機構。スレッドまたはプロセスは特定のロックを使用して、そのロックによって保護されている共有メモリーにアクセスします。データのロックとアンロックは、プログラマだけがロックの対象物を決めるという意味で自発的と言えます。データレース、相互排他、**mutex** ロック、セマフォロック、シングルロック戦略、スピンロックも参照してください。

ワード

コンピュータ内で単一エンティティとして格納、アドレス付け、伝送、および演算の対象となる、順序付けのある文字の集合。SPARC ワークステーションの場合、1 ワードは 32 ビットです。

索引

数字

10 進表現

精度, 17

正の最小正規数, 17

正の最大正規数, 17

範囲, 17

A

adb, 67

addrans

乱数発生機能, 54

C

C ドライバ

例、C から FORTRAN のサブルーチンを呼び出す, 151

convert_external

2 進浮動小数点, 54

データ変換, 54

D

dbx, 67

E

errno.h

errno の値の定義, 267

F

-fast, 169

floatingpoint.h

ハンドラのタイプを定義する

C および C++, 75

fmod, 269

-fnonstd, 169

G

Goldberg 稿

IEEE 標準, 177, 191

概要, 175

参考資料, 238

システムの側面, 211

謝辞, 238

詳細, 224

はじめに, 176

まとめ, 237

丸め誤差, 177

H

HUGE

IEEE 標準との互換性, 264

HUGE_VAL

IEEE 標準との互換性, 264

I

IEEE 拡張倍精度記憶形式

4 倍精度

SPARC アーキテクチャ, 11

INF

SPARC アーキテクチャ, 12

x86 アーキテクチャ, 15

NaN

x86 アーキテクチャ, 17

最上位

明示的先行ビット

x86 アーキテクチャ, 13

小数部

x86 アーキテクチャ, 13

正規数

SPARC アーキテクチャ, 12

x86 アーキテクチャ, 15

バイアス指数

x86 アーキテクチャ, 13

非正規数

SPARC アーキテクチャ, 12

x86 アーキテクチャ, 15

ビットフィールドの割り当て

x86 アーキテクチャ, 13

符号ビット

x86 アーキテクチャ, 14

IEEE 規格 754

拡張倍精度記憶形式, 3

単精度形式, 3

倍精度形式, 3

IEEE 形式

言語のデータ型との関係, 5

IEEE 単精度形式

NaN、非数, 8

INF、正の無限大, 6

混合数、有効数字, 7

小数部, 6

正規数

正の最大, 7

正規数のビットパターン, 6

精度、正規数, 7

デノーマル数, 7

バイアス指数, 6

バイアス指数、暗黙のビット, 7

非正規数のビットパターン, 6

ビットの割り当て, 6

ビットパターンと対応する値, 7

ビットフィールドの割り当て, 6

符号ビット, 6

IEEE 倍精度記憶形式

INF 正の無限大, 9

NaN、非数, 11

暗黙のビット, 9

仮数, 9

小数部, 8

SPARC の記憶, 8

x86 の記憶, 8

正規数, 10

精度, 10

デノーマル数, 10

バイアス指数, 8

非正規数, 10

ビットパターンと対応する値, 10

ビットフィールドの割り当て, 8

符号ビット, 8

ieee_flags, 43

自然発生した例外ビットを検証する

C の例, 125

丸め精度, 45

丸め方向, 44

累積例外フラグ, 44

例外フラグの設定

C の例, 128

ieee_functions

ビットマスク演算, 39

浮動小数点例外, 40

ieee_handler, 75

共通の例外でのトラップ, 58

例、シーケンスの呼び出し, 68

- 例外での異常終了
 - FORTTRAN の例, 149
- 例外のトラップ
 - C の例, 129
- ieee_retrospective
 - nonstandard_arithmetic の実施, 46
 - アンダーフロー例外フラグのチェック, 169
 - 精度, 46
 - 非標準 IEEE モードに関する情報の出力, 45
 - 浮動小数点状態レジスタ (FSR), 46
 - 浮動小数点例外, 45
 - 不要な例外に関する情報の出力, 45
 - 丸め, 46
 - 例外メッセージを抑制する, 47
- ieee_sun
 - IEEE 推奨関数, 39
- ieee_values
 - 4 倍精度値, 41
 - INF を表現する, 41
 - NaN を表現する, 41
 - 正規数を表現する, 41
 - 単精度値, 42
 - 浮動小数点値を表現する, 41
- ieee_values 関数
 - C の例, 111
- INF
 - ゼロ除算のデフォルト結果, 59

L

- lcrans
 - 乱数発生機能, 54
- libm
 - SVID 準拠, 265
 - 関数の一覧, 29
 - デフォルトディレクトリ
 - 実行可能ファイル, 29
 - ヘッダーファイル, 29
 - 標準インストール, 29
- libm 関数
 - 4 倍精度, 38
 - 単精度, 38

- 倍精度, 38
- libm による SVID の動作
 - Xt コンパイラオプション, 268
- libm による X/Open の動作
 - Xa コンパイラオプション, 268
- libmil(インラインテンプレートも参照), 265
- libsunmath
 - 関数の一覧, 31
 - デフォルトディレクトリ
 - 実行可能ファイル, 31
 - ヘッダーファイル, 31
 - 標準インストール, 31

M

- MAXFLOAT, 267

N

- NaN, 1, 13, 265, 269
- nonstandard_arithmetic
 - IEEE 段階的アンダーフローのオフ, 169
 - アンダーフロー
 - 段階的, 48

P

- PATH 環境変数、設定, xiv
- Pi
 - 無限に正確な値, 54

S

- shufrans
 - 疑似乱数を動かす, 54
- standard_arithmetic
 - IEEE動作のオン, 170
- store 0, 22
 - アンダーフロー結果のフラッシュ, 26
- SVID 例外
 - errno を EDOM に設定する

不適格なオペランド, 264
errno を ERANGE に設定する
オーバーフローまたはアンダーフロー, 264
matherr, 264
PLOSS, 269
TLOSS, 269
System V Interface Definition (SVID), 263

V

values.h
エラーメッセージの定義, 267

X

-Xa, 267
-Xc, 267
-Xt, 267
X_TLOSS, 267

あ

アクセスできるマニュアル, xvi
アンダーフロー
nonstandard_arithmetic, 48
しきい値, 26
小数点演算, 22
段階的, 23
アンダーフローしきい値
拡張倍精度, 22
単精度, 22
倍精度, 22

お

オペレーティングシステム数学ライブラリ
libm.a, 29
libm.so, 29

か

数の配列の生成
FORTRAN の例, 108

き

基数変換
10 進から2 進へ, 20
2 進から10 進へ, 20
書式付き入出力, 20

く

クロック速度, 171

こ

コンパイラオプション
-Xa
libm による X/Open の動作, 268
-Xt
libm による SVID の動作, 268
コンパイル、アクセス, xv

さ

三角関数
引数還元, 53

し

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, 58
自然発生した例外ビットを検証する
C の例, 125
自然発生した例外フラグを検証する
C の例, 127
書体と記号について, xiii
小数点演算の正確度
浮動小数点形式と整数形式, 4

す

数値の集合の変換, 18

数直線

10 進表現, 18

2 倍, 25

バイナリ表現, 18

せ

正確度, 269

しきい値, 28

小数点演算, 4

有効数字 (～の数), 18

正規数

正の最小, 22, 26

正の最大, 7

ゼロにフラッシュ (Store 0 を参照), 22

た

単位、最後の place 内, 53

段階的アンダーフロー

誤差特性, 24

単精度の値を表現する

C の例, 106

単精度の表現

C の例, 106

単精度形式, 6

て

データ型

IEEE 形式との関係, 5

と

トラップ

ieee_retrospective, 46

例外での異常終了, 138

は

倍精度の値を表現する

C の例, 106

FORTRAN の例, 107

倍精度の表現

C の例, 106

FORTRAN の例, 107

ひ

比較不可能な比較

NaN, 60

浮動小数点の値, 60, 61

引数還元

三角関数, 53

非正規数

小数点演算, 22

ふ

浮動小数点

例外リスト, 4

浮動小数点オプション, 160

浮動小数点キュー (FQ), 163

浮動小数点形式と整数形式間での変換, 4

浮動小数点ステータスレジスタ (FSR), 158, 163

浮動小数点例外, 1

ieee_functions, 39

ieee_retrospective, 46

共通の例外, 58

自然発生した例外ビット, 125

定義, 58

デフォルトの結果, 58

トラップ優先度, 61

フラグ, 62

切り捨て, 62

現在の, 62

優先順位, 63

例外での異常終了, 138

例外の一覧, 58

浮動小数点例外のトラップ

C の例, 129

へ

平方根命令, 165, 265

ベッセル関数, 269

ま

マニュアル索引, xvi

マニュアルページ、アクセス, xiv

マニュアル、アクセス, xvi

丸めエラー

正確度

誤差, 23

丸め精度, 4

丸め方向, 4

C の例, 118

ら

乱数機能

shufrans, 54

乱数生成, 108

れ

例外での異常終了

C の例, 138

例外のトラップ

C の例, 129

例外フラグの設定

C の例, 128