

**インテル® アーキテクチャ(IA)浮動小数点ユニット(FPU)、  
ストリーミング SIMD 拡張命令(SSE)、  
ストリーミング SIMD 拡張命令 2(SSE2)を  
使用した浮動小数点算術演算**

**バージョン 2.0**

**2000 年 7 月**

資料番号: 248608J-001

**【輸出規制に関する告知と注意事項】**

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

**【資料内容に関する注意事項】**

- ・本ドキュメントの内容を予告なしに変更することがあります。
  - ・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。
  - ・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。
  - ・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。
- インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。
- ・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センター

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

\*一般にブランド名または商品名は各社の商標または登録商標です。

Copyright © Intel Corporation 1999, 2000

## 目次

1.	はじめに .....	5
2.	IA FPU を使用した浮動小数点計算 .....	6
2.1	FPU アーキテクチャ .....	7
2.2	FPU ステータス・ワードと制御ワード .....	8
2.3	浮動小数点例外 .....	10
2.4	ソフトウェア例外処理 .....	14
2.5	NaN の処理 .....	15
2.6	FPU 命令 .....	16
2.7	例 .....	21
3	ストリーミング SIMD 浮動小数点命令 .....	33
3.1	ストリーミング SIMD 浮動小数点命令アーキテクチャ .....	33
3.2	SIMD コントロール/ステータス・レジスタ .....	34
3.3	浮動小数点例外 .....	35
3.4	ソフトウェア例外処理 .....	37
3.5	NaN の処理 .....	38
3.6	ストリーミング SIMD 浮動小数点命令 .....	38
3.7	SSE を使用したアプリケーションの開発 .....	41
3.8	例 .....	41
4	ストリーミング SIMD 浮動小数点命令 2 .....	45
4.1	ストリーミング SIMD 浮動小数点命令 2 アーキテクチャ .....	45
4.2	SIMD 制御/ステータス・レジスタ .....	45
4.3	浮動小数点例外 .....	46
4.4	ソフトウェア例外処理 .....	47
4.5	NaN の処理 .....	47
4.6	SSE2 .....	48
4.7	SSE2 を使用したアプリケーションの開発 .....	50
4.8	例 .....	51
5	相違点のまとめ .....	53
6	結論 .....	60

## 改訂履歴

改訂番号	改訂履歴	改訂時期
2.0	インテル® Pentium® 4 プロセッサに関する改訂	2000年7月
1.0	初版	1999年9月

## 参考資料

このアプリケーション・ノートでは次の資料を参考にした。これらの資料には、本書で取り上げた事項を理解するための背景情報が記載されている。

- [1] 『IEEE Standard for Binary Floating-Point Arithmetic』 ANSI/IEEE Std 754-1985
- [2] 『インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、上巻、中巻、下巻』 インテル、1999年。
- [3] 『Visual C++<sup>†</sup> 6.0 On-Line Manual』 Microsoft Corporation、1999年

## 摘要

本書では、インテル®アーキテクチャの浮動小数点ユニット(FPU)と浮動小数点命令(x87 命令)の概要と、ストリーミングSIMD 拡張命令(SSE)およびストリーミングSIMD 拡張命令2(SSE2)でインテル・アーキテクチャ・プロセッサに追加された新しい浮動小数点命令の概要について説明する。従来のx87 浮動小数点計算に対する新しい命令の機能、利点、および相違点と、2進浮動小数点算術に関するIEEE 規格のサポートのレベルについて説明する。

## 1. はじめに

インテル・アーキテクチャ(IA)プロセッサの浮動小数点計算は、インテル® Pentium® III プロセッサの導入以前は、IA 浮動小数点スタック・モデルを使用して、スカラ演算でのみ実行されていた。しかし、3D グラフィックス、ビデオ・デコーディング/エンコーディング、音声認識アルゴリズムなどを使用する新しいアプリケーションでは、必要なパフォーマンスがますます高度化している。これらのアプリケーションは、複数のデータ要素に対して同じ浮動小数点演算を並列で実行する。この性質は、SSE と SSE2 で実現された SIMD(Single Instruction, Multiple Data)計算モデルに最適である。

SSE は、3D グラフィックス・アプリケーションの処理を、従来のインテル・アーキテクチャ・プロセッサより高速化する。SSE のプログラミング・モデルは、インテル® MMX® テクノロジ命令モデルとよく似ているが、SSE 単精度浮動小数点命令は、パックド整数ではなく、パックド単精度浮動小数点数を操作する。また、SSE には、64 ビット SIMD 整数命令セットに対する拡張と、単精度浮動小数点スカラ命令が追加された。

SSE2 テクノロジでは、新しい SIMD 倍精度浮動小数点命令と新しい SIMD 整数命令が、IA-32 インテル・アーキテクチャに導入された。特に、SIMD 倍精度浮動小数点命令は、単精度より高い精度を必要とする科学計算アプリケーションの処理を高速化する。さらに、SSE2 の 128 ビット SIMD 整数命令は、整数のダイナミック・レンジの拡張を必要とする新しい高性能アプリケーションを実行でき、64 ビット・インテル MMX テクノロジ命令を使ってコーディングされていた既存のアプリケーションの処理も高速化する。このアプリケーション・ノートでは、x87 浮動小数点計算、SSE 単精度浮動小数点計算、および SSE2 倍精度浮動小数点計算に関する数値の精度の問題を扱う際に便利な、いくつかのコーディング手法について説明する。

第 2 章では、IA 浮動小数点ユニット(FPU)について説明する。これには、データ型、アーキテクチャ、制御ワードとステータス・ワード、浮動小数点例外、ソフトウェア例外処理、および NaN の処理が含まれる。また、FPU 命令についても簡単に説明する。丸め誤差とその影響(二重丸め誤差を含む)、極小数の検出、マスクされていないフォルトまたはトラップの発生、および IA-32 FPU 計算モデルと IEEE [1]モデルの相違点の例を示す。FPU アーキテクチャと操作の検討によって、IA FPU で実行される浮動小数点計算と、パックド単精度/倍精度浮動小数点数を操作する新しい命令を使用した計算の比較が可能になる。インテル・アーキテクチャ浮動小数点ユニットについてよく理解している読者は、この章を読まなくてもかまわない。SSE と SSE2 には、浮動小数点の加算、減算、乗算、除算(逆数の近似値を含む)、平方根(平方根の逆数の近似値を含む)、比較、および各種のデータ形式間の変換命令だけが含まれるため、この章では、それに対応する FPU 命令だけを取り上げる。

第 3 章では、SSE について説明する。これには、データ型、アーキテクチャ、コントロール/ステータス・レジスタ、浮動小数点例外、ソフトウェア例外処理、NaN の処理が含まれる。新しい命令についても簡単に説明する。SSE を使用した計算の 2 つの例を示す。

第 4 章では、SSE2 について説明する。この章は第 3 章と同じ構成である。

第 5 章では、FPU、SSE、および SSE2 を使用した計算の相違点を表にまとめて示す。また、簡単な計算の例を、単精度の SSE、倍精度の SSE2、拡張倍精度の FPU 命令を使用して実行し、各モデルの予想される精度を比較する。

## 2. IA FPU を使用した浮動小数点計算

浮動小数点数は、一般的に、符号ビット、指数部、および仮数部を連結してコード化される。正規の浮動小数点数は、0(正)または1(負)の符号ビット、 $[E_{\min}, E_{\max}]$ の範囲の指数、および $[1, 2]$ の範囲の仮数で構成され、最上位ビットの上位に2進小数点が暗黙的に置かれる(これは、整数ビットまたはJビット=1の意味である)。実際のコード化では、コード化された指数部が常に正になり、指定された指数範囲を超えないように、(指数部に正のバイアスを加算することによって)指数部がバイアスされる。正規の浮動小数点数の値は、次のようになる。

$$f = (-1)^{\text{符号}} \cdot \text{仮数} \cdot 2^{\text{指数バイアス}}$$

浮動小数点数の最下位ビット位置に対応する値は、*ulp*(unit-in-the-last-place)と呼ばれる。上記の浮動小数点数では、ulp は次のように表現される。

$$1 \text{ ulp} = 0.0\dots 01 \cdot 2^{\text{指数バイアス}} = 2^{\text{指数バイアス} - N + 1}$$

N は仮数部のビット数である。

IA FPU は、メモリに格納できる3種類の浮動小数点形式(単精度、倍精度、拡張倍精度)をサポートしている(これらの形式は、2進浮動小数点算術演算に関するIEEE規格で要求または推奨されている[1])。表1に各形式の特性を示す。FPU内では、浮動小数点数は常に80ビットで表現され、単精度数と倍精度数の指数範囲は15ビットに拡張される。仮数部のサイズは変わらないが、IA-32スタック単精度および倍精度形式では、64ビットの長さまで0でパディングする。拡張倍精度数は、FPU内でもメモリ内でも同じように表現される。ただし、メモリ形式では、単精度および倍精度の浮動小数点数の整数ビットは明示的に表現されない。

表1: 単精度、倍精度、IA-32 FP レジスタ・スタック単精度、IA-32 FP レジスタ・スタック倍精度、および拡張倍精度の浮動小数点形式の特性

浮動小数点形式	単精度(メモリ形式)	倍精度(メモリ形式)	IA-32 スタック単精度	IA-32 スタック倍精度	拡張倍精度
指数ビット	8	11	15	15	15
仮数ビット	24	53	24	53	64
浮動小数点数のサイズ(ビット)	32	64	80(40ビットの後続の0)	80(11ビットの後続の0)	80
$E_{\min}$	-126	-1022	-16382	-16382	-16382
$E_{\max}$	127	1023	16383	16383	16383
指数バイアス	127	1023	16383	16383	16383
浮動小数点数の範囲(絶対値)	$2^{-149} \sim (2 \cdot 2^{-23}) \cdot 2^{127}$	$2^{-1074} \sim (2 \cdot 2^{-52}) \cdot 2^{1023}$	$2^{-16405} \sim (2 \cdot 2^{-23}) \cdot 2^{16383}$	$2^{-16434} \sim (2 \cdot 2^{-52}) \cdot 2^{16383}$	$2^{-16445} \sim (2 \cdot 2^{-63}) \cdot 2^{16383}$
浮動小数点数の範囲(絶対値の近似値)	$10^{-44.85} \sim 10^{38.53}$	$10^{-323.3} \sim 10^{308.2}$	$10^{-4938} \sim 10^{4932}$	$10^{-4947} \sim 10^{4932}$	$10^{-4950} \sim 10^{4932}$

正規の浮動小数点数以外に、以下の値をコード化できる(拡張倍精度形式または浮動小数点レジスタ形式でのみ、先行ビットが明示的に表現される)。

- **デノーマル数:** 指数 0(実際には  $E_{\min}$  の値を示す)と、先頭ビットが 0 のゼロでない仮数
- **ゼロ:** 符号ビット 0 または 1、指数 0、および仮数 0
- **無限大:** 符号ビット 0 または 1、指数 11...1、および仮数 10...0
- **NaN(Not a Number):** 無視される符号ビット、指数 11...1、および 10...0(無限大のために予約済み)以外のゼロでない仮数。NaN には、対応する浮動小数点数値が存在しない。最初の小数ビットが 0 の NaN は、シグナル型 NaN(SNaN)である(SNaN は無効操作浮動小数点例外を発生させる)。最初の小数ビットが 1 の NaN は、クワイエット NaN(QNaN)である。特殊なタイプの QNaN として、QNaN 実数不定値(符号=1、指数=11...1、仮数=110...0)がある。この値は、特定の無効操作( $\infty - \infty$  など)の結果として返される。

拡張倍精度形式または FPU スタック形式では、以下の浮動小数点数のエンコーディングは、インテル・アーキテクチャでサポートしていない。

- 非正規浮動小数点数。11...1 以外のゼロでない指数と、先頭ビットが 0 の仮数で構成される。
- 疑似無限大。指数 11...1 と仮数 0 で構成される。
- 疑似 NaN。指数 11...1 と、先頭ビットが 0 のゼロでない仮数で構成される。

サポートしていないエンコーディングは、拡張倍精度形式または FPU スタック形式の数のみが存在する。これは単精度および倍精度メモリ形式では、暗黙的に J ビット = 1 と見なすためである。

## 2.1 FPU アーキテクチャ

IA FPU は、プロセッサの整数ユニットと並行して動作するコプロセッサである。実際のマイクロアーキテクチャは、プロセッサの世代によって異なる。現在のところ、32 ビット版インテル・アーキテクチャには、最大 2 つの整数ユニットと 2 つの浮動小数点ユニットが搭載している。

FPU は、8 つの 80 ビット・データ・レジスタを持つ。これらのデータ・レジスタは、メモリから値がロードされるか、または浮動小数点命令の結果のデスティネーションとなる。メモリからロードされる値が拡張倍精度形式でない(つまり、単精度浮動小数点数、倍精度浮動小数点数、整数、またはパワード BCD 整数の場合、値は自動的に拡張倍精度に変換される)。

FPU データ・レジスタは、循環スタックとして構成される。現在のスタック・トップ・レジスタ(最後にスタック上に置かれたデータ・アイテムを保持するレジスタ)のレジスタ番号は、図 2 に示すように、FPU ステータス・ワードの TOP フィールドに格納する。スタックのトップのレジスタは、常に ST(0)と呼ばれ、それに続くレジスタは、ST(1)、ST(2)、... ST(7)と呼ばれる。新しい値がスタックの ST(0)にプッシュされると、それまでの ST(0)は ST(1)になり、ST(1)は ST(2)になる(以下同様)。8 つの slots がすべてフルの場合は、スタック・オーバーフローが発生する。値がスタックの ST(0)からポップすると、それまでの ST(1)は ST(0)になり、ST(2)は ST(1)になる(以下同様)。スタックが空の場合は、スタック・アンダーフローが発生する。FXCH

命令は、スタック・トップ・レジスタとスタック内の他の任意のレジスタを非常に低いコストで入れ替えることで、スタック・モデルで発生するボトルネックを軽減できる。

FPU 実行環境は、以下のレジスタ、オペレーション・コード、およびポインタで構成される。

- 8つの FPU データ・レジスタ
- 制御ワードを格納する1つのコントロール・レジスタ
- ステータス・ワードを格納する1つのステータス・レジスタ
- 個々の FPU レジスタの内容が、有効なデータ、0、特殊な値、空のいずれであることを示す、1つのタグ・レジスタ
- 浮動小数点オペレーション・コード(FPU が最後に実行した非制御命令の 11 ビット・オペコード)
- FPU 命令ポインタ(FPU が最後に実行した非制御命令への 48 ビット・ポインタ)
- FPU オペランド(データ)ポインタ(FPU が最後に実行した非制御命令のデータへの 48 ビット・ポインタ)

プロセッサは、例外ハンドラに状態情報を渡すために、FPU オペレーション・コード、命令、およびオペランド(データ)を保存する(ソフトウェア例外ハンドラは、オペレーティング・システム・コンポーネントと、オペレーティング・システムによって起動するユーザ・コンポーネントで構成される)。

MMX テクノロジ命令が使用する MMX テクノロジ・レジスタは、FPU レジスタにマッピングする。このため、MMX テクノロジ操作から FPU 操作に移行するには、EMMS 命令を使用する必要がある。EMMS 命令は、FPU タグ・ワードを空(すべてのタグ・フィールドを 1)に設定し、MMX テクノロジ・レジスタを使用可能としてマークする。逆に、FPU 操作から MMX テクノロジ操作に移行する際は、特殊な命令を実行する必要はない。これは、MMX テクノロジ命令を実行すると、FPU タグ・ワードはフル(すべてのタグ・フィールドが 0)に設定され、TOP は 0 に設定されるからである。TOP が 0 でなかった場合や、0 でないタグがあった場合、これらの変更はアーキテクチャ上で参照可能である。情報が誤って失われないように、移行の前に、FPU 操作で FPU スタックを空にしておくことをお勧めする。また、MMX テクノロジ・コードは、FPU 操作に切り替えられる可能性がある場合は、後で使用するためにレジスタ内に値を残してはならない。

## 2.2 FPU ステータス・ワードと制御ワード

図 1 に示す 16 ビットの FPU 制御ワードは、浮動小数点例外のマスクとアンマスク、精度モード、および丸めモードを制御する。ビット 0~5(IM、DM、ZM、OM、UM、PM)は、例外マスク・ビットである。セットされている場合、これらのビットは、FPU 例外(それぞれ無効操作例外、デノーマル例外、ゼロ除算例外、オーバーフロー例外、アンダーフロー例外、および不正確結果例外)をマスクする(無効にする)。ビット 8 とビット 9 は、精度制御フィールド(PC)である。このフィールドは、FPU によって実行する浮動小数点計算の精度を指定する。計算結果の仮数部は、PC=00B の場合は 24 ビット、PC=10B の場合は 53 ビット、PC=11B の場合は 64 ビットである。PC=01B の値は予約している(初期の IA プロセッサでは、この値を使用して拡張倍精度を表していた)。PC フィールドは、浮動小数点の加算、減算、乗算、除算、および平方根計算にのみ影響を与える。ビット 10 とビット 11 は、丸め制御フィールド(RC)である。こ

のフィールドは、デスティネーション形式では結果を正確に表現できない場合に使用される、IEEE の丸め手法[1]を指定する。RC=00B の場合は最近値方向への丸め、RC=01B の場合は切り捨て、RC=10B の場合は切り上げ、RC=11B の場合はゼロ方向への丸めを実行する。最後に、ビット 12(X)は、無限大制御フラグである。このフラグは、下方互換性のために残されており、現在の役割はない。

浮動小数点演算の結果の丸めは、デスティネーション形式で制約された指数範囲を考慮に入れて、デスティネーションの精度(仮数部のビット数)に合わせて実行する。場合によっては、([1]の 7.4 節のように)あたかもデスティネーションの指数範囲には境界がないものとして、デスティネーションの精度に合わせた丸めによって得られる仮定上の結果を考慮に入れると便利である。

ハードウェアが返す実際の結果と仮定上の結果が異なるのは、仮定上の結果が極大(指数が  $E_{\max}$  より大きい)または極小(仮定上の結果が 0 でなく、指数が  $E_{\min}$  より小さい)の場合のみである。仮定上の結果が極大である場合は、実際の結果は、(絶対値で) に等しい値またはデスティネーション形式で表現できる最大の浮動小数点数( $FPMAX = 1.1\dots1 \cdot 2^{E_{\max}}$ )に等しい値になる。仮定上の結果が極小である場合は、二重丸め誤差を避けるために、実際の丸めを「取り消す」必要がある。実際の結果は、デノーマライズ(仮数部を右にシフトして左側に 0 を挿入し、 $E_{\min}$  に達するまで指数をインクリメントする)の実行後、デスティネーションの精度に合わせた丸めによって得られる。この結果は、絶対値で、最小のノーマル数( $FPMIN = 1.0 \cdot 2^{E_{\min}}$ )、デノーマル数、または 0 に等しい値になる。

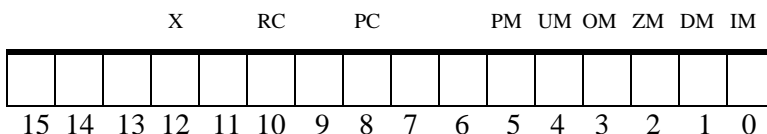


図1: FPU 制御ワード

図 2 に示す 16 ビットの FPU ステータス・ワードは、FPU の現在の状態を示す。ビット 0~5(IE、DE、ZE、OE、UE、PE)は、それぞれ無効操作例外、デノーマル例外、ゼロ除算例外、オーバーフロー例外、アンダーフロー例外、および不正確結果例外の例外フラグである。これらのビットは、例外フラグが最後にクリアされた後、1 つ以上の浮動小数点例外が検出されたことを示す。これらは「スティッキー」ビットであり、一度セットされると、直接にクリアされるまでその値を保持する。ビット 0(IE)と同時にセットされた場合、ビット 7(SF)は、スタック・オーバーフロー(一杯になったスタックに浮動小数点数をプッシュしようとした場合)またはスタック・アンダーフロー(空のスタックから浮動小数点数をポップしようとした場合)を示す。この場合、スタック・アンダーフロー( $C1 = 0$ )とオーバーフロー( $C1 = 1$ )は、ビット 9( $C1$ )で区別する。ビット 7(ES)は、いずれかのマスクされていない例外フラグをセットしたときにセットされる、例外サマリ・ビットである。ビット 8~10 とビット 14( $C0$ 、 $C1$ 、 $C2$ 、 $C3$ )は条件コードである。一部の浮動小数点比較命令は、これらのビットを使用して、( $C0$ 、 $C2$ 、および  $C3$  によって)命令の結果を示す。これらのビットは、一部の算術演算でも使用する。たとえば、不正確結果を示す PE フラグがセットされた場合、 $C1 = 1$  は、最後の丸めが切り上げであったことを示す。条件コードのその他の特殊な役割については、[2]を参照のこと。ビット 11、12、および 13 には、浮動小数点レジスタ・スタックのトップへの TOP ポインタが入る。ビット 14(B)は、FPU がビジーであることを示す。このビットは、下方互換性のためにのみ残されている。

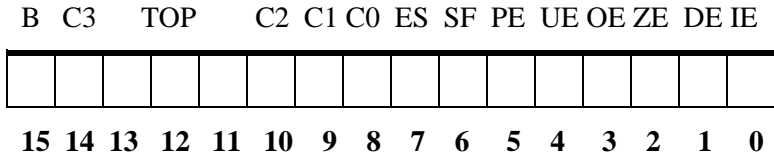


図2: FPU ステータス・ワード

## 例 1: 極小数の検出

次の計算は、極小数の検出の例を示している。ここでは、[1]で指定されているように、単精度の算術演算が 1 ステップで実行できるとする。2つの単精度浮動小数点数  $a$  と  $b$  の乗算の場合を考える。仮数部は 24 ビットで 2 進数で表される。指数部は 10 進数で表し、 $E_{\min} = -126$  がノーマル数の最小の指数である。

$$a = 1.1\dots10 \cdot 2^{-126} \approx 2.35 \cdot 10^{-38}$$

$$b = 1.0\dots01 \cdot 2^{-1} \approx 0.5$$

10 進数で表した場合、この積の限りなく正確な値は次のようになる。

$$a \cdot b = (2 - 2^{-22}) \cdot (1 + 2^{-23}) \cdot 2^{-127} = (2 - 2^{-45}) \cdot 2^{-127}$$

仮数部が有界でない場合、この積の限りなく正確な値は次のようになる。

$$a \cdot b = 1.11\dots1 \cdot 2^{-127}$$

この場合、仮数部は 46 ビットの 1 に拡張される。仮数部 24 ビットへの丸めと有界でない指数を使用して得られる仮定上の結果を使用して、結果が極小かどうかを分析できる。この仮定上の結果は、丸めモードによって異なる(以下の仮数部は 24 ビットである)。

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 最近値方向への丸めの場合(極小でない)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ 切り捨ての場合(極小)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 切り上げの場合(極小でない)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ ゼロ方向への丸めの場合(極小)}$$

(仮数部 24 ビットへの丸めと有界の指数を使用した)実際の丸めの結果も、丸めモードによって異なる(以下の仮数部は 24 ビットである)。次のように、切り捨てとゼロ方向への丸めではデノーマル数、最近値方向への丸めと切り上げでは最小のノーマル数の結果が得られる。

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 最近値方向への丸めの場合 (P フラグをセット)}$$

$$a \cdot b = 0.11\dots1 \cdot 2^{-126} \text{ 切り捨ての場合 (P、U フラグをセット)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 切り上げの場合 (P フラグをセット)}$$

$$a \cdot b = 0.11\dots1 \cdot 2^{-126} \text{ ゼロ方向への丸めの場合 (P、U フラグをセット)}$$

2.7 節の例 4 に、この計算のコード例が記載している。

## 2.3 浮動小数点例外

FPU は、無効(#I)(#IS - スタック・オーバーフローまたはアンダーフローと#IA - 無効算術演算に分類される)、デノーマル・オペランド(#D)、ゼロ除算(#Z)、オーバーフロー(#O)、アンダー

フロー(#U)、および不正確結果(精度)(#P)例外の 6 種類の浮動小数点例外を検出する。FPU 制御ワードには、各例外に対応するマスク/アンマスク(無効/有効)ビットがある。FPU ステータス・ワードには、各例外に対応する「スティッキー」例外フラグ・ビットがある。

無効操作例外、デノーマル・オペランド例外、およびゼロ除算例外は、計算前型の例外(浮動小数点フォルト)である。オーバーフロー例外、アンダーフロー例外、および不正確結果例外は、計算後型の例外(浮動小数点トラップ)である。

マスクされた(マスク・ビットがセットされた)例外が検出されると、FPU はその例外を自動的に処理して、あらかじめ定義された結果を生成し、プログラムの実行を続ける。

マスクされていない(マスク・ビットがクリアされている)例外が検出されると、FPU はソフトウェア例外ハンドラを起動してその例外を処理する。浮動小数点フォルト(#I、#D、#Z)の場合は、例外を発生させた命令の入力オペランドは変更されず、入力値に基づいてソフトウェア・ハンドラが結果を生成する。浮動小数点トラップの場合は、例外を発生させた命令の入力オペランドが上書きされることがあるが、結果はソフトウェア・ハンドラに渡され、ハンドラはそれを使用して最終的な結果を生成する。

どちらの場合も(マスクされた例外またはマスクされていない例外)、FPU ステータス・レジスタ内でその例外に対応するステータス・フラグがセットされる。

無効操作例外は、SNaN オペランド(一部の浮動小数点比較命令では QNaN)、サポートしていない形式の浮動小数点数(たとえば疑似無限大)、または許容範囲外の浮動小数点数(たとえば、大きすぎて浮動小数点数から整数への変換命令で整数に変換できない有限数)が原因で発生する。また、無効操作例外は、 $\infty - \infty$ 、 $0 \cdot \infty$ 、 $0/0$ 、 $\infty/\infty$ 、または 0 でない負の値の平方根( $-0.0$  の平方根は  $-0.0$  である)などの無効な計算でも発生する。

デノーマル・オペランド例外(IEEE 規格[1]で定義されていない唯一のカテゴリ)は、デノーマル・オペランドによって発生する。

ゼロ除算例外は、0 でないノーマル数またはデノーマル数を正または負の 0 で割った場合に発生する。

オーバーフロー例外は、結果が極大である場合に発生する。FPU スタックをデスティネーションとする FPU 命令では、この例外を検出するときに FPU 制御ワードの PC フィールドで指定される精度(仮数部のビット数)と 15 ビットの指数範囲が考慮に入れられる。

オーバーフロー例外がマスクされている場合は、結果は(絶対値で) $\infty$ または MAXFP になり、FPU ステータス・ワード内のオーバーフロー・ステータス・フラグと不正確結果ステータス・フラグがセットされる。

オーバーフロー例外がマスクされていない場合は、ソフトウェア・トラップ・ハンドラに渡す結果は、オーバーフローを発生させた浮動小数点命令のデスティネーションによって異なる。デスティネーションがメモリ・ロケーションの場合は(これは浮動小数点ストアの場合に限られる)、ソース・オペランドとデスティネーション・オペランドは変更されず、FPU ステータス・レジスタ内のオーバーフロー・フラグがセットされ、ソフトウェア例外ハンドラを起動する(2.7 節の例 8 を参照)。デスティネーションが FPU スタック上の浮動小数点レジスタの場合は、 $2^{24576}$  でスケールダウン(除算)された限りなく正確な結果がデスティネーションの精度に合わせて丸められ、オーバーフロー・ステータス・フラグをセットする。この場合は、結果が不正確であった場合にのみ、不正確結果ステータス・フラグがセットされる。丸めの実行後の仮数部が限りなく正確な仮数部より大きい場合は、FPU ステータス・レジスタ内の条件コード

C1 がセットされる。それ以外の場合は、C1 はクリアされる(C0 ~ C3 はスティッキー・ビットではない)。ソフトウェア例外ハンドラを起動する(2.7 節の例 9 を参照)。

アンダーフロー例外は、マスクされているかどうかによって異なる規則を適用する。しかし、FPU スタックをデスティネーションとする FPU 命令でのオーバーフローと同様に、アンダーフローが検出される時は、FPU 制御ワードの PC フィールドで指定する精度(仮数部のビット数)と 15 ビットの指数範囲が考慮に入れられる。

アンダーフロー例外がマスクされている場合は、結果が極小かつ不正確である場合にのみ、FPU ステータス・ワード内のアンダーフロー・ステータス・フラグをセットする。不正確結果ステータス・フラグもセットされる。

アンダーフロー例外がマスクされていない場合は、結果が単に極小である場合に、ソフトウェア・ハンドラを起動する。ソフトウェア・トラップ・ハンドラに渡される結果は、アンダーフローを発生させた浮動小数点命令のデスティネーションによって異なる。デスティネーションがメモリ・ロケーションである場合は(これは浮動小数点ストアの場合に限る)、ソース・オペランドとデスティネーション・オペランドは変更されず、FPU ステータス・レジスタ内のアンダーフロー・フラグがセットされ、ソフトウェア例外ハンドラを起動する。デスティネーションが FPU スタック上の浮動小数点レジスタである場合は、 $2^{24576}$  でスケールアップ(乗算)した限りなく正確な結果がデスティネーションの精度に合わせて丸められ、アンダーフロー・ステータス・フラグがセットされる。この場合は、結果が不正確であった場合にのみ、不正確結果ステータス・フラグをセットする。返された仮数部が限りなく正確な仮数部より大きい場合は、FPU ステータス・レジスタ内の条件コード C1 をセットする。それ以外の場合は、C1 はクリアされる。ソフトウェア例外ハンドラが起動される。

FPU 浮動小数点例外に優先する条件には、次のものがある。

- 浮動小数点スタックのアンダーフロー/オーバーフロー、サポートしていない形式または許容範囲外のオペランド、または SNaN オペランド(一部の浮動小数点比較命令では QNaN)を原因とする無効操作例外。
- QNaN オペランド(これは例外ではないが、QNaN オペランドの処理は優先順位の低い例外に優先する)。
- 上記以外の無効操作例外、またはゼロ除算例外。
- デノーマル・オペランド例外(マスクされている場合は、実行が続けられ、より優先順位の低い例外が発生することがある)。
- 数値オーバーフローまたはアンダーフロー例外。通常は不正確結果例外と一緒に発生する。
- 不正確結果例外

浮動小数点命令は、整数命令またはシステム命令と並行して実行する場合がある。浮動小数点例外の報告方法が原因で、同時実行によって浮動小数点例外ハンドラに問題が起こる可能性がある。マスクされていない浮動小数点例外が発生した場合、FPU は浮動小数点命令の実行を停止し、「待機型」浮動小数点命令または WAIT/FWAIT 命令を次に実行するときにソフトウェア例外ハンドラだけを起動する(「待機型」浮動小数点命令とは、それ以前の浮動小数点命令によって発生した、未処理のマスクされていない浮動小数点例外の処理を開始する命令である。「非待機型」浮動小数点命令は、このような動作を行わない)。マスクされていない例外を発生させた浮動小数点命令のソース・オペランドやデスティネーション・オペランドが、例外を

検出してからソフトウェア・ハンドラが起動されるまでの間に変更されるのを防ぐには、マスクされていない例外を知らせるすべての浮動小数点命令の後に、例外を同期化する命令(任意の「待機型」浮動小数点命令または WAIT/FWAIT 命令)を置く必要がある。少なくとも、メモリ・オペランドを使用する命令については、この処置が必要である。オペランドが浮動小数点スタック上にある場合は、非浮動小数点命令によって変更されることはあり得ない。

## 例 2: マスクされていないアンダーフロー例外

次の例は、マスクされていないアンダーフロー例外を発生させる計算を示している。例 1 の 2 つの単精度浮動小数点数  $a$  と  $b$  の例を考える。

$$a = 1.1\dots10 \cdot 2^{-126} \approx 10^{-54.28}$$

$$b = 1.0\dots01 \cdot 2^{-1} \approx 0.5$$

アンダーフロー例外はアンマスク、不正確結果例外はマスク、精度制御フィールドは単精度に設定し、乗算のデスティネーションは 32 ビットのメモリ・ロケーションであるとする(この演算は、FMUL と FST の 2 つの命令を使用してコーディングできる。例 5 も参照)。

仮数部 24 ビットへの丸めと有界でない指数を使用した仮定上の結果は、丸めモードによって異なる。切り捨てとゼロ方向への丸めの場合に、結果は極小になる(例 1 を参照)。FMUL によって計算される実際の結果(IA-32 スタック単精度形式の数)は、レジスタ・スタック上に置かれ、仮定上の丸めの結果に等しくなる。IA-32 レジスタ・スタック単精度形式の 15 ビットの指数範囲を超えていないため、アンダーフロー例外は発生しない。

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 最近値方向への丸めの場合}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ 切り捨ての場合}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 切り上げの場合}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ ゼロ方向への丸めの場合}$$

FST 命令は、この結果を単精度に変換し、メモリ(32 ビット単精度形式)に格納する。

切り捨てまたはゼロ方向への丸めの場合、FST は極小の入力値を検出し、アンダーフロー例外を発生させる。P および U ステータス・フラグがセットされ、ソフトウェア例外ハンドラを起動する。ソース・オペランドとデスティネーション・オペランドは変更されない。

最近値方向への丸めまたは切り上げの場合は、P ステータス・フラグをセットし、 $1.0 \cdot 2^{-126}$  の結果がデスティネーション・アドレスに格納される。これらの 2 つの場合は(最近値方向への丸めまたは切り上げ)、不正確結果例外がアンマスクされていると、(以下に説明するように)不正確例外が発生する。

デスティネーション形式で表現される浮動小数点演算の結果が、有界でない仮数と有界でない指数範囲を使用して計算された限りなく正確な結果と異なる場合は、不正確結果(精度)例外が発生する。

不正確結果例外がマスクされており、アンダーフロー例外またはオーバーフロー例外が発生しなかった場合は、FPU ステータス・レジスタ内の不正確結果ステータス・フラグをセットし、丸められた結果がデスティネーション・オペランドに格納される。

不正確結果例外がアンマスクされており、アンダーフロー例外またはオーバーフロー例外が発生しなかった場合は、FPU ステータス・レジスタ内の不正確結果ステータス・フラグをセット

し、(上の場合と同様に)丸められた結果はデスティネーション・オペランドに格納され、ソフトウェア例外ハンドラが起動する。

不正確結果例外が、マスクされた数値オーバーフローまたはアンダーフローと一緒に発生した場合は、オーバーフロー/アンダーフロー・ステータス・フラグと不正確結果ステータス・フラグがセットされる。結果は、オーバーフロー/アンダーフロー例外について説明した方法で格納する。不正確結果例外がアンマスクされている場合は、FPU はソフトウェア例外ハンドラを起動する。

不正確結果例外が、マスクされていない数値オーバーフローまたはアンダーフローと一緒に発生し、デスティネーション・オペランドが FPU スタック上の浮動小数点レジスタである場合は、オーバーフロー/アンダーフロー・ステータス・フラグと不正確結果ステータス・フラグをセットする。結果は、マスクされていないオーバーフロー/アンダーフロー例外について説明した方法で格納される。FPU はソフトウェア例外ハンドラを起動する。

(マスクされていないか、またはマスクされている)不正確結果例外が、マスクされていない数値オーバーフローまたはアンダーフロー例外と一緒に発生し、デスティネーション・オペランドがメモリ・ロケーションの場合は(これは浮動小数点ストアの場合に限られる)、不正確結果状態は無視される。

## 2.4 ソフトウェア例外処理

ソフトウェア例外処理ハンドラの起動には、内部(ネイティブ)モードと外部(非同期ピン指向)モード(MS-DOS<sup>+</sup>互換モードとも呼ばれる)の 2 種類の動作モードを利用できる。どちらのメカニズムがアクティブになるかは、CR0.NE ビットで制御される。

内部モードが推奨モードである。内部モードを選択するには、コントロール・レジスタ CR0 の NE フラグをセットする(CR0.NE=1)。マスクされていない浮動小数点例外が発生すると、次の「待機型」浮動小数点命令(非制御 FPU 命令または WAIT 命令)の実行の直前か、次の MMX テクノロジ命令の前に、ソフトウェア割り込みベクタ 16(#MF)によってソフトウェア例外ハンドラを起動する。内部モードでは、浮動小数点例外の発生は同期化するが、次の「待機型」浮動小数点命令または次の MMX テクノロジ命令まで遅延される。

外部(非同期ピン指向)モード(MS-DOS 互換モード)を選択するには、コントロール・レジスタ CR0 の NE フラグをクリアする(CR0.NE=0)。このモードは、以前の浮動小数点コプロセッサを搭載したシステムとの互換性のために用意している。このモードの動作は、CPU ピンと外部ハードウェアの相互作用が関連するため、プラットフォーム固有である。このモードの利用モデルは次のとおりである。マスクされていない浮動小数点例外によって、プロセッサの FERR#ピンをアサートする(CR0.NE=1 かどうかを問わず、この動作は常に行われる)。FERR#ピンは、浮動小数点例外の原因となった命令の実行時にアサートすることも、次の「待機型」浮動小数点命令または次の MMX テクノロジ命令まで遅延することもある。このピンがいつアサートするかは、プロセッサによって異なる。たとえば、Intel<sup>®</sup>486<sup>™</sup> プロセッサでは、ほとんどの場合このピンはただちにアサートするが、移植可能なソフトウェアは、この動作に依存してはならない。FERR#ピンがアサートすると、外部ハードウェアは外部割り込み要求を生成するか、IGNNE#ピンをアサートする。どちらをどのように実行するかは、プラットフォーム固有である。次の「待機型」浮動小数点命令または次の MMX テクノロジ命令の実行時に、マスクされていない浮動小数点例外が未処理になっている場合は、外部ハードウェア割り込みが到着するか、IGNNE#ピンがアサートするまで、プロセッサは動かなくなる。外部ハードウェア

割り込みは、外部プログラマブル割り込みコントローラ(PIC)によって生成する。この割り込みは、(外部ハードウェアによって指定する割り込みベクタ番号の)INTR#ピンまたは(割り込みベクタ2の)#NMIピンを使用する。したがって、外部モードでは、マスクされていない浮動小数点例外に関連するハードウェア割り込みは非同期である。つまり、このハードウェア割り込みは、例外の原因となる命令を実行してから、次の「待機型」浮動小数点命令または次の MMX テクノロジ命令を実行するまでの任意の時点で到着する。特定の CPU バージョンおよび命令では、割り込みが到着する時点が正確にわかることもあるが、移植可能なソフトウェアは、この認識に依存してはならない。移植可能なソフトウェアは、例外に関連する割り込みを完全に非同期の方法で処理する必要がある。ただし、「待機型」浮動小数点命令または MMX テクノロジ命令が実行されると、未処理の浮動小数点例外がすべて「一掃される」のは確かである。したがって、これ以降の浮動小数点命令を実行しないコードは、非同期浮動小数点例外について配慮する必要はない。

ソフトウェア・ハンドラは、一般的に、格納された FPU ステート情報の確認、例外を発生させた状態の修正、ステータス・ワード内の例外フラグのクリアを実行し、正常な実行を再開できるように、割り込みをかけられたプログラムに復帰する。実行の再開は、オペレーティング・システムによって制御される。オペレーティング・システムは、修正した FPU ステートのリストアも行う。アプリケーションによっては、ステータス・フラグがセットしている浮動小数点例外をアンマスクすると、(既に説明したように)ソフトウェア・ハンドラが起動されるため、FPU ステータス・ワードをクリアしなければならない場合がある。

## 2.5 NaN の処理

すでに説明したように、ほとんどの浮動小数点命令で、QNaN(クワイエット NaN)を検出しても浮動小数点例外は発生しない(一部の浮動小数点比較命令を除く)。しかし、QNaN の処理は、一部の浮動小数点例外に優先する。たとえば、QNaN/0.0 の結果は QNaN になり、ゼロ除算例外は発生しない。表 2 に、FPU 命令が QNaN または SNaN オペランドを検出した場合や、マスクされた無効操作例外が発生した場合の、QNaN 結果の生成規則を示す。FPU 命令の両方のオペランドが SNaN になる可能性は、非常に低い。SNaN のうち少なくとも 1 つは FPU スタック上に置かれるが、SNaN が FPU スタック上に置かれるのは、FRSTOR 命令を使用してメモリから FPU ステート(8 つの浮動小数点レジスタを含む)をロードした場合に限る。SNaN はメモリ内で生成するが、FRSTOR 以外の命令は、SNaN を FPU スタックに移動する前または SNaN に依存する結果を FPU スタック上で生成する前に、SNaN を QNaN に変換する。

NaN に対応する数値や、NaN に等しい値は存在しない。しかし、ビット・フィールドの内容によって NaN を分類できるように、NaN の符号、指数部、および仮数部の概念を使用する。また、表 2 に示した動作(2 つのソース・オペランドが「反対の符号の」NaN である場合に、符号ビットが 0 の NaN を返す動作)は、インテル® Pentium® Pro プロセッサ以降の IA プロセッサにのみ適用される(それ以前のプロセッサでは、この場合の動作は一定ではない)。

表 2: FPU 命令の QNaN 結果の生成規則

ソース・オペランド	QNaN 結果
SNaN と QNaN	QNaN ソース・オペランド
2 つの SNaN	仮数が大きい方の SNaN を、QNaN(クワイエット NaN)に変換して返す。仮数が同じであれば、「正の」SNaN を QNaN に変換して返す。
2 つの QNaN	仮数が大きい方の QNaN を返す。仮数が同じであれば、「正の」QNaN を返す。
SNaN と実数値	SNaN を QNaN(クワイエット NaN)に変換して返す。
QNaN と実数値	QNaN ソース・オペランド
NaN オペランドはないが、無効操作例外が報告される	QNaN 実数不定値

## 2.6 FPU 命令

この節では、インテル・アーキテクチャ FPU 命令と、FPU 命令によって発生する浮動小数点例外について順番に簡単に説明する。一部の命令には、浮動小数点レジスタ・スタックから値をポップするバージョンや、オペランドの順序を逆転するバージョンがある。2.7 節の例では、SSE と SSE2 に対応する機能を持つ FPU 命令を取り上げる。オペランドを使用しない FPU 命令や、オペランドを実質的に変更しない FPU 命令は、非算術命令と呼ばれる。このような命令は、スタック・オーバーフローまたはアンダーフローによる無効例外だけを発生させる。その他の FPU 命令は、算術命令と呼ばれ、(スタック・オーバーフローおよびアンダーフロー以外に)6 種類の数値例外を発生させる。すべての FPU 命令についての詳細は、[2]を参照のこと。

### 1. データ転送命令

- FLD: floating-point load - メモリからの 32 ビット、64 ビット、または 80 ビット浮動小数点値または FPU スタック・レジスタからの 80 ビット浮動小数点値を、FPU レジスタ・スタックの ST(0)にプッシュする。浮動小数点例外: スタック・オーバーフロー、I、D
- FST/FSTP: floating-point store - レジスタ・スタック・トップ ST(0)の値を、32 ビットまたは 64 ビットメモリ・ロケーションか、80 ビット浮動小数点スタック・レジスタに格納する。FSTP は、レジスタ・スタック・トップから 80 ビットのメモリ・ロケーションへのストアも実行でき、常にレジスタ・スタック・トップをポップする。浮動小数点例外: スタック・アンダーフロー、I、O、U、P
- FXCH: スタック・レジスタ ST(i)とスタック・レジスタ・トップ ST(0)の内容を交換する。浮動小数点例外: スタック・アンダーフロー

- FCMOVcc: EFLAG レジスタの CF、ZF、PF ビットに基づいた、スタック・レジスタ ST(i)からスタック・レジスタ・トップ ST(0)への条件付き浮動小数点移動。浮動小数点例外: スタック・アンダーフロー
- FILD: 16 ビット、32 ビット、または 64 ビット整数を、メモリから FPU スタック・レジスタ ST(0)にロードする。浮動小数点例外: スタック・オーバーフロー
- FIST/FISTP: レジスタ・スタック・トップ ST(0)の値を、16 ビットまたは 32 ビットのメモリ・ロケーションに整数としてストアする。FISTP は、レジスタ・スタック・トップから 64 ビットのメモリ・ロケーションへのストアも実行でき、常にレジスタ・スタック・トップをポップする。浮動小数点例外: スタック・アンダーフロー、I、P
- FBLD: メモリ内の 80 ビット BCD 値を拡張倍精度実数形式に変換し、FPU スタック上にプッシュする。浮動小数点例外: スタック・オーバーフロー
- FBSTP: FPU スタック・レジスタ・トップ ST(0)の内容を、80 ビットのメモリ・ロケーションに BCD 形式でストアし、ST(0)をポップする。浮動小数点例外: スタック・アンダーフロー、I、P

## 2. 定数ロード命令

- FLDZ、FLD1、FLDPI、FLDL2T、FLDL2E、FLDLG2、FLDLN2: それぞれ、+0.0、+1.0、 $\log_2 10$ 、 $\log_2 e$ 、 $\log_{10} 2$ 、 $\log_e 2$  の値を浮動小数点スタックの ST(0)にロードする。浮動小数点例外: スタック・オーバーフロー

## 3. 基本算術命令

- FADD/FADDP: floating-point add – スタック・レジスタ・トップ ST(0)の内容と 32 ビットまたは 64 ビット・メモリの実数を加算し、結果をレジスタ・スタック上にプッシュする。2つのスタック・レジスタ(少なくとも1つはトップ・レジスタでなければならない)の内容を加算し、いずれかのレジスタを結果で置き換える。FADDP は、スタック・レジスタだけを操作し、レジスタ・スタック・トップをポップする。浮動小数点例外: スタック・アンダーフロー、I、D、O、U、P
- FIADD: メモリ内の 16 ビットまたは 32 ビット整数を拡張倍精度形式に変換し、FPU レジスタ・スタック・トップ ST(0)の浮動小数点値に加算する。浮動小数点例外: I、D、O、U、P
- FSUB/FSUBP/FSUBR/FSUBRP: floating-point subtract – FSUB/FSUBP は、FADD/FADDP によく似ている(メモリ・オペランドを使用する場合は、スタック・レジスタ・トップ ST(0)に第 1 オペランドが入る)。FSUBR/FSUBRP は、浮動小数点逆減算を実行する。つまり、オペランドの順序を逆にして、FSUB/FSUBP と同じ操作を実行する。浮動小数点例外: スタック・アンダーフロー、I、D、O、U、P
- FISUB/FISUBR: subtract integer (converted to double-extended format) from floating-point – FIADD によく似ている(スタック・レジスタ・トップ ST(0)に、第 1 オペランドとデスティネーションが入る)。FISUBR は、オペランドの順序を逆にして、FISUB と同じ操作を実行する。浮動小数点例外: I、D、O、U、P

- FMUL/FMULP: floating-point multiply – FADD/FADDP によく似ている。浮動小数点例外: スタック・アンダーフロー、I、D、O、U、P
- FIMUL: multiply floating-point and integer (converted to double-extended format) – FIADD によく似ている。浮動小数点例外: I、D、O、U、P
- FDIV/FDIVP/FDIVR/FDIVRP: floating-point divide – FDIV/FDIVP は、FADD/FADDP によく似ている(メモリ・オペランドを使用する場合は、スタック・レジスタ・トップ ST(0) に被除数が入る)。FDIVR/FDIVRP は、浮動小数点逆除算を実行する。つまり、オペランドの順序を逆にして、FDIV/FDIVP と同じ操作を実行する。浮動小数点例外: スタック・アンダーフロー、I、D、Z、O、U、P
- FIDIV/FIDIVR: divide floating-point to integer (converted to double-extended format) – FIADD によく似ている(スタック・レジスタ・トップ ST(0)に、被除数とデスティネーションが入る)。FIDIVR は、オペランドの順序を逆にして、FIDIV と同じ操作を実行する。浮動小数点例外: I、D、Z、O、U、P
- FSQRT: レジスタ・スタックのトップの値の浮動小数点平方根を求める。浮動小数点例外: スタック・アンダーフロー、I、D、P
- FRNDINT: FPU 制御ワードで指定される丸めモードを使用して、レジスタ・スタックのトップの浮動小数点値を整数に丸める。浮動小数点例外: スタック・アンダーフロー、I、D、O、U、P
- FABS: レジスタ・スタックのトップの浮動小数点数の絶対値を求める。浮動小数点例外: スタック・アンダーフロー
- FCHS: ST(0)の符号を変更する。浮動小数点例外: スタック・アンダーフロー
- FPREM: partial remainder – ST(0)を ST(1)で割ったときに得られる剰余で、ST(0)を置き換える(除算にはゼロ方向への丸めを使用する)。浮動小数点例外: スタック・アンダーフロー、I、D、U
- FPREM1: IEEE partial remainder – ST(0)を ST(1)で割ったときに得られる IEEE 剰余[2]で、ST(0)を置き換える(除算には最近値方向への丸めを使用する)。浮動小数点例外: スタック・アンダーフロー、I、D、U
- EXTRACT: ST(0)の浮動小数点値を指数部と仮数部に分けて、指数部を ST(0)に格納し、仮数部を(元の符号と指数 0x3fff を使用して)レジスタ・スタックにプッシュする。浮動小数点例外: スタック・アンダーフロー、スタック・オーバーフロー、I、D、Z

#### 4. 比較命令と分類命令

- FCOM/FCOMP/FCOMPP: compare real - FPU レジスタ・スタックのトップの値と、メモリ内の 32 ビット実数、メモリ内の 64 ビット実数、または FPU スタック上の他の値を比較する。FCOMP は、レジスタ・スタック・トップ ST(0)をポップする。FCOMPP は、FPU スタックの上位 2 つの値だけを比較でき、比較が完了したら、それらの値をスタックからポップする。比較の結果は、FPU ステータス・ワードのビット C3、C2、および C0 に格納される。この操作は「順序化可能」な比較である。つまり、QNaN オペランドを検出すると、無効操作例外が発生する。浮動小数点例外: スタック・アンダーフロー、I、D

- FUCOM/FUCOMP/FUCOMPP: unordered compare real – FCOM/FCOMP/FCOMPP によく似ているが、メモリ・オペランドとの比較はできない。また、QNaN オペランドが検出されても、無効操作例外は発生しない。浮動小数点例外: スタック・アンダーフロー、I、D
- FICOM/FICOMP: メモリ内の 16 ビットまたは 32 ビット整数と、FPU スタックのトップの浮動小数点値を比較する。FICOMP は、レジスタ・スタック・トップ ST(0)をポップする。比較の結果は、FPU ステータス・ワードのビット C3、C2、および C0 に格納される。QNaN オペランドを検出しても、無効操作例外は発生しない。浮動小数点例外: スタック・アンダーフロー、I、D
- FCOMI/FCOMIP: FPU レジスタ・スタックのトップの値と FPU スタック上の他の値を比較し、EFLAGS を設定する。FCOMIP は、レジスタ・スタック・トップ ST(0)をポップする。QNaN オペランドを検出すると、無効操作例外が発生する。浮動小数点例外: スタック・アンダーフロー、I
- FUCOMI/FUCOMIP: FCOMI/FCOMIP によく似ているが、QNaN オペランドを検出しても、無効操作例外は発生しない。浮動小数点例外: スタック・アンダーフロー、I
- FTST: レジスタ・スタック・トップ ST(0)の内容と 0.0 を比較する。比較の結果は、FPU ステータス・ワードのビット C3、C2、および C0 に格納される。浮動小数点例外: スタック・アンダーフロー、I、D
- FXAM: レジスタ・スタック・トップ ST(0)の内容を、サポートなし、NaN、0、デノーマル数、ノーマル数、無限大、または空として分類する。結果は FPU ステータス・ワードのビット C3、C2、および C0 に格納される。浮動小数点例外: なし

## 5. 三角関数命令

- FSIN: ST(0)を ST(0)の正弦で置き換える。浮動小数点例外: スタック・アンダーフロー、I、D、U、P
- FCOS: ST(0)を ST(0)の余弦で置き換える。浮動小数点例外: スタック・アンダーフロー、I、D、P(U は発生しない)
- FSINCOS: ST(0)を ST(0)の正弦で置き換え、FPU スタック上に余弦をプッシュする。浮動小数点例外: スタック・アンダーフロー、I、D、U、P
- FPTAN: tangent - ST(0)を  $\tan(\text{ST}(0))$  で置き換え、FPU スタック上に 1.0 をプッシュする (偏角の絶対値が  $2^{63}$  より小さい場合)。浮動小数点例外: スタック・アンダーフロー、I、D、U、P
- FPATAN: arctangent - ST(1)を  $\arctan(\text{ST}(1)/\text{ST}(0))$  で置き換え、レジスタ・スタック ST(0)をポップする。浮動小数点例外: スタック・アンダーフロー、I、D、U、P

必要に応じて、偏角を小さくするために、仮数部 66 ビットの の値(内部に格納)が使用される。

## 6. 対数命令、指数命令、およびスケール命令

- FYL2X: ST(1)を  $ST(1) * \log_2 ST(0)$  で置き換え、ST(0)をポップする。浮動小数点例外: スタック・アンダーフロー、I、D、Z、O、U、P
- FYL2XP1: ST(1)を  $ST(1) * \log_2 (ST(0) + 1.0)$  で置き換え、ST(0)をポップする。浮動小数点例外: スタック・アンダーフロー、I、D、O、U、P
- F2XM1: ST(0)を  $2^{ST(0)} - 1$  で置き換える。浮動小数点例外: スタック・アンダーフロー、I、D、U、P
- FSCALE: ST(0)を ST(1)でスケールリングする。浮動小数点例外: スタック・アンダーフロー、I、D、O、U、P

## 7. FPU 制御命令(特に断らない限り、浮動小数点例外は発生しない)

- FINIT/FNINIT: 未処理のマスクされていない浮動小数点例外がないかどうかチェックした後(FINIT)、またはチェックを行わずに(FNINIT)、最近値方向への丸め、64 ビットの精度、および例外のマスクに FPU を初期設定する。
- FLDCW: 2 バイトのメモリ・ロケーションから FPU 制御ワードをロードする。この操作によって FPU ステータス・ワード内の未処理の例外がアンマスクされた場合は、次の「待機型」FPU 命令の実行時にその例外を生成する。
- FSTCW/FNSTCW: 未処理のマスクされていない浮動小数点例外がないかどうかチェックした後(FSTCW)、またはチェックを行わずに(FNSTCW)、FPU 制御ワードを 2 バイトのメモリ・ロケーションにストアする。
- FSTSW/FNSTSW: 未処理のマスクされていない浮動小数点例外がないかどうかチェックした後(FSTSW)、またはチェックを行わずに(FNSTSW)、FPU ステータス・ワードを 2 バイトのメモリ・ロケーションまたは AX レジスタにストアする。
- FCLEX/FNCLEX: 未処理のマスクされていない浮動小数点例外がないかどうかチェックした後(FCLEX)、またはチェックを行わずに(FNCLEX)、浮動小数点例外フラグをクリアする。
- FLDENV: (プロセッサの動作モードに基づいて)14 バイトまたは 28 バイトのメモリ領域から FPU 環境をロードする。1 にセットしたステータス・フラグが、FPU ステータス・ワード内のアンマスクされている例外ビットにロードした場合は、次の「待機型」浮動小数点命令の実行時にその例外が生成される。
- FSTENV/FNSTENV: 未処理のマスクされていない浮動小数点例外がないかどうかチェックした後(FSTENV)、またはチェックを行わずに(FNSTENV)、(プロセッサの動作モードに基づいて)14 バイトまたは 28 バイトのメモリ領域に FPU 環境をストアし、すべての浮動小数点例外をマスクする。
- FRSTOR: (プロセッサの動作モードに基づいて)94 バイトまたは 108 バイトのメモリ領域から FPU ステータスをロードする。この操作によって FPU ステータス・ワード内の未処理の例外がアンマスクされた場合は、次の「待機型」FPU 命令の実行時にその例外を生成する。

- FSAVE/FNSAVE: 未処理のマスクされていない浮動小数点例外がないかどうかチェックした後(FSAVE)、またはチェックを行わずに(FNSAVE)、(プロセッサの動作モードに基づいて)94 バイトまたは 108 バイトのメモリ領域に FPU ステータスをストアし、FPU を再初期化する。
- FINCSTP: FPU ステータス・レジスタの TOP フィールドをインクリメントする(タグ・レジスタとデータ・レジスタには影響を与えない)。
- FDECSTP: FPU ステータス・レジスタの TOP フィールドをデクリメントする(タグ・レジスタとデータ・レジスタには影響を与えない)。
- FFREE: 任意の ST(i)のタグを空に設定する。
- FNOP: ノー・オペレーション
- FWAIT/WAIT: 未処理のマスクされていない浮動小数点例外がないかどうかチェックし、それを処理する。

すべての「非待機型」浮動小数点命令は、FPU 制御命令のカテゴリに含まれる。FNINIT、FNSTENV、FNSAVE、FNSTSW、FNSTCW、および FNCLEX は、未処理のマスクされていない浮動小数点例外の有無をチェックしないが、その他の点では、各命令に対応する上記の「待機型」命令と同じように動作する。したがって、これらの命令が、マスクされていない浮動小数点例外の原因となった他の浮動小数点命令の後に続く場合、これらの命令を実行する前にソフトウェア例外ハンドラが起動することはない。FNSTSW と FNSTCW を除いて、これらの命令は、FPU ステータス・ワードをクリアするか、または浮動小数点例外をマスクするので、未処理の例外は失われる。FNSTSW と FNSTCW は、次の「待機型」浮動小数点命令で未処理の例外を処理する。

## 2.7 例

この節では、浮動小数点計算(特に、FPU 命令を使用した計算)のさまざまな例を示す。

コード例は C([3]を参照)で開発され、主に FPU 命令を表す IA-32 アセンブリ言語文を使用している。以下の例で使用する“mov”命令は、メモリと整数レジスタの間または整数レジスタ同士の間で整数データを移動する。IA-32 アセンブリ言語は、メモリとの間のデータ転送にサイズ指定子を使用する。たとえば、DWORD PTR は 32 ビット・データを表し、TBYTE PTR は 80 ビット・データを表す。

以下の例で使用する浮動小数点定数は、IEEE 規格[1]で規定した形式の 16 進数で指定される(この形式は、10 進数では表現できない値を指定できる)。たとえば、単精度値 0x00800001 は、1 ビットの符号(0)、8 ビットのバイアス付き指数部(00000001)、および 24 ビットの仮数部(100...001)に分けられる。バイアスなしの指数は  $1 - 127 = -126$  であるため、コード化値 0x00800001 の浮動小数点数の値は、 $+1.00...01 * 2^{126}$  になる。

### 例 3: 等しいかどうかのテストに対する丸め誤差の影響

この例では、浮動小数点算術演算に関連する制限を示す。IEEE 規格[1]に厳密に従っても、丸め誤差が発生して、精度が失われることがある。浮動小数点式“fpexpr”の予想される結果が“res”である場合、次のようなテストはほとんどの場合に失敗する。

```
if (fexpr == res)
    printf ("SUCCESS\n");
else
    printf ("FAIL\n");
```

上のテストの代わりに、計算した結果と予想する結果が、実数値“eps”で指定される一定の小さな区間内に入るかどうかをテストするのをお勧めする。

```
if (-eps < fexpr - res && fexpr - res < eps)
    printf ("SUCCESS\n");
else
    printf ("FAIL\n");
```

このテストの例として、 $\sqrt{x}$  が  $x$  の平方根の限りなく正確な値であり、 $(\sqrt{x})_m$  がデスティネーションの精度に合わせて最近値方向に丸められた  $\sqrt{x}$  の値であるとする。以下のコードは、 $x$  のいくつかの単純な値について、次の等式が成り立つかどうかをチェックする(計算は単精度で実行する)。

$$((\sqrt{x})_m * (\sqrt{x})_m)_m = x$$

```
#include <stdio.h>
void main () {
    float x, y, z;
    char *px, *py;
    int i;
    unsigned short cw, *pcw; // control word and pointer to it
    pcw = &cw;
    // set control word
    cw = 0x003f; // round to nearest, 24 bits, floating-point exc. disabled
    // cw = 0x043f; // round down, 24 bits, floating-point exc. disabled
    // cw = 0x083f; // round up, 24 bits, floating-point exc. disabled
    // cw = 0x0c3f; // round to zero, 24 bits, floating-point exc. disabled
    __asm {
        mov eax, DWORD PTR pcw
        fldcw [eax]
    }
    for (i = 0 ; i < 11 ; i++) {
        x = (float)i; // x = 1.0, 2.0, ..., 10.0
        // compute y = sqrt (x)
        px = (char *)&x;
        py = (char *)&y;
        __asm {
            mov eax, DWORD PTR px
            fld DWORD PTR [eax]
            fsqrt
            mov eax, DWORD PTR py
            fstp DWORD PTR [eax]
        }
    }
}
```

```

    }
    z = y * y;
    printf ("x = %f = 0x%x\n", x, *(int *)&x);
    printf ("y = %f = 0x%x\n", y, *(int *)&y);
    printf ("z = %f = 0x%x\n", z, *(int *)&z);
    if (z == x)
        printf ("EQUAL\n\n");
    else
        printf ("NOT EQUAL\n\n");
    }
}

```

丸めモードが最近値方向への丸めに設定している場合、完全な平方ではない  $x$  の値について、 $x$  と  $z$  が等しいかどうかをテストすると、 $x = 3.0$ 、 $5.0$ 、および  $10.0$  では成功し、 $x = 2.0$ 、 $6.0$ 、 $7.0$ 、および  $8.0$  では失敗する。丸めモードが最近値方向への丸めでない場合は、完全な平方ではないすべての  $x$  の値について、このテストは失敗する。

#### 例 4: 極小数の検出

この例は、例 1 の極小数を検出するテストのコード例を示している。

```

#include <stdio.h>

void main () {
    float a, b, c; // single precision numbers (of size 4 bytes)
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc; // pointers to single precision numbers
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    // will compute c = a * b
    psw = &sw;
    pcw = &cw;
    // clear and read status word, set control word
    cw = 0x033f; // round to nearest, 64 bits, fp exc.disabled
    // cw = 0x073f; // round down, 64 bits, fp exc.disabled
    // cw = 0x0b3f; // round up, 64 bits, fp exc.disabled
    // cw = 0x0f3f; // round to zero, 64 bits, fp exc. disabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMPUTATION sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x00fffffe;
    a = *(float *)&u; // a = 1.11...10 * 2^-126
    pb = (char *)&b; u = 0x3f000001;
    b = *(float *)&u; // b = 1.00...01 * 2^-1
    pc = (char *)&c;
    // compute c = a * b
    __asm {
        mov eax, DWORD PTR pa;
        fld DWORD PTR [eax]; // push a on the FPU stack
        mov eax, DWORD PTR pb;

```

```

fld DWORD PTR [eax]; // push b on the FPU stack
fmulp st(1), st(0); // a * b in st(1), pop st(0)
mov eax, DWORD PTR pc;
fstp DWORD PTR [eax]; // c = a * b from FPU stack to memory, pop st(0)
mov eax, DWORD PTR psw
fstsw [eax]
}
printf ("AFTER COMPUTATION sw = %4.4x\n", sw);
printf ("c = %8.8x = %f\n", *(unsigned int *)&c, c);
}

```

最近値方向への丸めまたはプラス無限大方向への丸めの場合、出力は不正確結果ステータス・フラグをセットし、結果が  $1.0 * 2^{-126}$  (最小の単精度ノーマル数) になるのを示す。

```

BEFORE COMPUTATION sw = 0000
AFTER COMPUTATION sw = 0220
c = 00800000 = 0.000000

```

マイナス無限大方向への丸めまたはゼロ方向への丸めの場合、出力は、不正確結果ステータス・フラグとアンダーフロー・ステータス・フラグをセットし、結果が  $0.11...1 * 2^{-126}$  (最大の単精度デノーマル数) になるのを示す。

```

BEFORE COMPUTATION sw = 0000
AFTER COMPUTATION sw = 0030
c = 007fffff = 0.000000

```

## 例 5: FPU 命令を使用した純粋な IEEE 計算

次の例は、x87 浮動小数点計算モデルと IEEE 規格[1]で推奨される浮動小数点計算モデルの主要な相違点を示す。2 進浮動小数点計算に関する IEEE 規格は、すべての中間計算を IEEE でサポートする形式での実行を推奨している。FPU 計算は、IEEE 規格に完全に準拠しているが、IEEE モデルより良いモデルを使用する。たとえば、計算の最終結果の精度が単精度である場合、中間結果は IEEE モデルより大きい指数範囲(この例では 8 ビットに対して 15 ビット)と、多くの場合はより高い精度を使用して、浮動小数点スタック上に保持する(精度は FPU 制御ワードの精度制御フィールドで指定し、通常は 53 または 64 ビットに設定される。この値は IEEE 単精度形式の 24 ビットより大きい)。したがって、FPU 計算モデルは、IEEE モデルより高い精度が得られ、中間計算によるオーバーフロー、アンダーフロー、および精度の低下を回避できる。

一方、中間結果が単精度または倍精度形式であると見なして、中間計算でアンダーフローまたはオーバーフローを発生させる場合は、FPU スタック上の浮動小数点演算を、メモリとの間のストアおよびロード操作と組み合わせる必要がある。この例として、単精度計算  $d = (a * b) / c$  の場合を考える( $a = 1.0 * 2^{115}$ 、 $b = 1.0 * 2^{125}$ 、 $c = 1.0 * 2^{120}$  とする)。単精度計算では、中間結果  $a * b = 1.0 * 2^{240}$  によってオーバーフロー例外が発生する。オーバーフロー例外と不正確例外がマスクされている場合、デフォルトの中間結果は正の無限大になり、最終結果も正の無限大になる。この「純粋な」IEEE モデルとは異なり、一般的な FPU 環境で実行する計算では、中間結果  $a * b = 1.0 * 2^{240}$  を FPU スタック上に格納し(指数範囲が 15 ビットのため、オーバーフローは発生しない)、 $d = (a * b) / c = 1.0 * 2^{120}$  が正しく計算され、この値をメモリに格納する。以下のコード例は、最初の計算では FPU スタック・モデルを使用し、2 番目の計算では「純粋な」IEEE 単精度モデルを使用する(この場合、IEEE モデルで規定されるすべての可能な例外を発生させるためには、すべての中間結果をメモリに格納する必要がある)。

ただし、ここで説明した方法は、倍精度計算についても有効だとは限らない(この方法は、単精度計算については常に有効である)。結果が倍精度デノーマル数になる場合、二重丸め誤差が発生する可能性がある。つまり、FPU スタック上で計算された結果は 64 ビットに丸められるため、fst 命令がこの値をデノーマル数に変換するときに、誤差が発生する可能性がある。この場合、単精度計算では二重丸め誤差は発生しない(二重丸め誤差の詳細は、例 6 を参照)。倍精度計算の問題を解決するには、53 ビットの精度に丸めるときに、オペランドを適切にスケールリングして、FPU スタック上でデノーマライズを強制的に発生させればよい。

```
#include <stdio.h>
void main () {
    float a, b, c, d; // single precision floating-point numbers
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc, *pd; // pointers to single precision numbers
    unsigned short sw, *psw; // status word and pointer to it
    // will compute d = (a * b) / c
    psw = &sw;
    // clear and read status word; set rounding to nearest,
    // and 64-bit precision
    __asm {
        finit
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMP. sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x79000000;
    a = *(float *)&u; // a = 1.0 * 2^115
    pb = (char *)&b; u = 0x7e000000;
    b = *(float *)&u; // b = 1.0 * 2^125
    pc = (char *)&c; u = 0x7b800000;
    c = *(float *)&u; // c = 1.0 * 2^120
    pd = (char *)&d;
    // compute d = (a * b) / c holding the intermediate result
    // a * b = 2^240 on the FPU stack
    __asm {
        mov eax, DWORD PTR pa;
        fld DWORD PTR [eax]; // push a on the FPU stack
        mov eax, DWORD PTR pb;
        fld DWORD PTR [eax]; // push b on the FPU stack
        fmulp st(1), st(0); // a * b = 2^240 in st(1), pop st(0)
        mov eax, DWORD PTR pc;
        fld DWORD PTR [eax]; // push c on the FPU stack
        fdivp st(1), st(0) // st(1) / st(0) = 2^120 in st(1), pop st(0)
        mov eax, DWORD PTR pd;
        fstp DWORD PTR [eax]; // d = 2^120 from FPU stack to mem., pop st(0)
        // read status word
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("AFTER FIRST COMP. sw = %4.4x\n", sw);
    printf ("d = %8.8x = %f\n", *(unsigned int *)&d, d); // d = 2^120
    // compute d = (a * b) / c saving the intermediate result
    // a * b = 2^240 to memory
    // round to nearest, 64-bit precision, floating-point exc. disabled
```

```

__asm {
    fclex mov eax, DWORD PTR pa;
    fld DWORD PTR [eax]; // push a on the FPU stack
    mov eax, DWORD PTR pb;
    fld DWORD PTR [eax]; // push b on the FPU stack
    fmulp st(1), st(0); // a * b = 2^240 in st(1), pop st(0)
    mov eax, DWORD PTR pd;
    fstp DWORD PTR [eax]; // d = a * b from the FPU stack to mem, pop st(0)
    fld DWORD PTR [eax]; // push d = +Inf from memory on the FPU stack
    mov eax, DWORD PTR pc;
    fld DWORD PTR [eax]; // push c on the FPU stack
    fdivp st(1), st(0) // st(1) / st(0) = +Inf in st(1), pop st(0)
    mov eax, DWORD PTR pd;
    fstp DWORD PTR [eax]; // d = +Inf from the FPU stack to mem, pop st(0)
    // read status word
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("AFTER SECOND COMP. sw = %4.4x\n", sw);
printf ("d = %8.8x = %f\n", *(unsigned int *)&d, d);
}

```

出力は、第1の計算(FPU スタック・モデル)では正確な結果を示し(ステータス・フラグはセットされない)、「純粋な」IEEE 単精度モデルでは無限大を示す(不正確結果ステータス・フラグとオーバーフロー・ステータス・フラグがセットされる)。

```

AFTER FIRST COMP. sw = 0000
d=7b800000= 1329227995784915900000000000000000000000.000000
AFTER SECOND COMP. sw = 0028
d = 7f800000 = 1.#INF00

```

## 例 6: 二重丸め誤差

次の例は、正しく丸められた結果が得られるはずの浮動小数点計算で、二重丸め誤差が発生する可能性を示す。二重丸め誤差は、たとえば、計算の限りなく正確な結果  $R$  を特定のデスティネーション精度に合わせて丸める必要があるとき、最初にデスティネーション精度より高精度または大きい指数範囲に合わせて  $R$  が丸められた場合に発生する。たとえば、指数がデスティネーション形式の範囲内に入る場合、'rn53' が仮数部 53 ビットへの最近値方向への丸め、'rn64' が 64 ビットへの丸めを表すとき、次の式は、左辺に二重丸め誤差が発生する可能性があるため、常に成り立つとは限らない。

$$((R)_{rn64})_{rn53} = (R)_{rn53}$$

$R$  が正規の浮動小数点数の単純な算術演算(加算、減算、乗算、除算、および平方根)の結果であるとき、最初の丸めの高精度形式の仮数部(上の式では 64 ビット)が、デスティネーション精度の仮数部(上の式では 53 ビット)のビット数の 2 倍より大きい場合は、二重丸め誤差は発生しないことがわかっている。

以下の例は、 $1.00\dots01 \cdot 2^{-126} * 1.00010\dots0 \cdot 2^{-1}$  の単精度の結果(仮数部は 24 ビットである)を、2 つの方法で計算する。第 1 の方法では、デスティネーションの精度(仮数部 24 ビット)に合わせて丸めた結果を、FPU スタック(指数範囲 15 ビット)上に置いた後、メモリ(仮数部 24 ビットと指数部 8 ビット)に格納する。この方法では、結果の  $0.100010\dots0 \cdot 2^{-126}$  が 1ulp だけ小さすぎる

ため、二重丸め誤差が発生する。第 2 の方法では、FPU スタック上で結果を倍精度に丸めた後 (仮数部 53 ビットと指数範囲 15 ビット)、メモリに格納する (仮数部 24 ビットと指数部 8 ビット)。この場合は、二重丸め誤差は発生せず、正しく最近値方向に丸められた結果  $0.100010\dots 01 \cdot 2^{-126}$  が得られる。ただし、倍精度の結果を目的とする場合は、この方法で二重丸め誤差を回避できるとは限らない。これは、1 つ上の精度では仮数部が 64 ビットに制限されるため、目的の 53 ビットの精度の 2 倍以上にならないからである。

```
#include <stdio.h>
void main () {
    float a, b, c; // single precision floating-point numbers
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc; // pointers to single precision numbers
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    // will compute c = a * b
    psw = &sw;
    pcw = &cw;
    // clear status flags, read status word, set control word
    cw = 0x003f; // round to nearest, 24 bits, fp exc. disabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE FIRST COMP. sw = %4.4x\n", sw);
    pa = (char *)&a; u = 0x00800001;
    a = *(float *)&u; // a = 1.00...01 * 2^-126
    pb = (char *)&b; u = 0x3f080000;
    b = *(float *)&u; // b = 1.00010...0 * 2^-1
    pc = (char *)&c;
    c = 123.0; // initialize c to random value
    // compute c = a * b with 24 bits of precision;
    // result a * b with 'unbounded' exponent on FPU stack
    __asm {
        mov eax, DWORD PTR pa
        fld DWORD PTR [eax] // push a on the FPU stack
        mov eax, DWORD PTR pb
        fld DWORD PTR [eax] // push b on the FPU stack
        fmulp st(1), st(0) // a * b in st(1), pop st(0)
        mov eax, DWORD PTR pc
        fstp DWORD PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
        // read status word
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("AFTER FIRST COMP. sw = %4.4x\n", sw);
    printf ("AFTER FIRST COMP. c = %8.8x = %f\n", *(unsigned int *)&c, c);
    // c = 0.100010...00 * 2^-126
    // clear status flags, read status word, set control word
    cw = 0x023f; // round to nearest, 53 bits, fp exc. disabled
    __asm {
```

```

    fclex
    mov eax, DWORD PTR pcw
    fldcw [eax]
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("BEFORE SECOND COMP. sw = %4.4x\n", sw);
// compute c = a * b with 53 bits of precision;
// result a * b with `unbounded' exponent on FPU stack
__asm {
    mov eax, DWORD PTR pa
    fld DWORD PTR [eax] // push a on the FPU stack
    mov eax, DWORD PTR pb
    fld DWORD PTR [eax] // push b on the FPU stack
    fmulp st(1), st(0) // a * b in st(1), pop st(0)
    mov eax, DWORD PTR pc
    fstp DWORD PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
    // read status word
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("AFTER SECOND COMP. sw = %4.4x\n", sw);
printf ("AFTER SECOND COMP. c = %8.8x = %f\n", *(unsigned int *)&c, c);
    // c = 0.100010...01 * 2-126
}

```

出力は次のようになる。

```

BEFORE FIRST COMP. sw = 0000
AFTER FIRST COMP. sw = 0030
AFTER FIRST COMP. c = 00440000 = 0.000000
BEFORE SECOND COMP. sw = 0000
AFTER SECOND COMP. sw = 0230
AFTER SECOND COMP. c = 00440001 = 0.000000

```

## 例 7: マスクされていないゼロ除算例外の発生

以下の例は、除算命令(FDIVP、 を 0.0 で割った場合)を原因とする、マスクされていないゼロ除算浮動小数点例外フォルトの発生を示している。同期化を行う命令は FSTP である(FDIVP の直後に置かれた FWAIT でもよい)。

`__try/except` 構文で、マスクされていない浮動小数点例外を処理できる。マスクされていない浮動小数点例外が“`__try`”ブロック内の文によって発生した場合は、実行される処置は、`__except ()`の引数になるフィルタ式の値(またはフィルタ式によって返される値)によって異なる。この値が“`EXCEPTION_EXECUTE_HANDLER`”である場合は、“`__except ()`”の後に続くコードが実行される(その他のオプションについては、[3]を参照)。

```

#include <stdio.h>
#include <excpt.h>
void main () {
    float f;
    unsigned short cw, *pcw; // control word and pointer to it
    pcw = &cw;

```

```

// clear status flags, set control word
cw = 0x033b; // round to nearest, 64 bits, zero-divide exceptions enabled
__asm {
    fclex
    mov eax, DWORD PTR pcw
    fldcw [eax]
}
__try {
    printf ("TRY BLOCK BEFORE DIVIDE BY 0\n");
    __asm {
        fldpi // load 3.1415926... in ST(0)
        fldz // load 0.0 in ST(0); 3.1415... in ST(1)
        fdivp st(1), st(0)
        // divide ST(1) by ST(0), result in ST(1), pop
        fstp f // store ST(0) in memory and pop stack top
    }
    printf ("TRY BLOCK AFTER DIVIDE BY 0 \n");
} __except(EXCEPTION_EXECUTE_HANDLER) {
    printf ("EXCEPT BLOCK\n");
}
}

```

マスクされていない例外が発生すると、ソフトウェア・ハンドラが起動される(以下の例外ブロック)。出力は次のようになる。

```

TRY BLOCK BEFORE DIVIDE BY 0
EXCEPT BLOCK

```

FSTP 命令がコメントアウトされている場合は、例外は報告されず、出力は次のようになる。

```

TRY BLOCK BEFORE DIVIDE BY 0
TRY BLOCK AFTER DIVIDE BY 0

```

### 例 8: 例外の原因となる命令がメモリ・ロケーションをデスティネーションとする場合のマスクされていないオーバーフロー例外の発生

次の例では、例外の原因となる命令がメモリ・ロケーションをデスティネーションとする場合に、オーバーフロー浮動小数点例外トラップを発生させる( $1.0 \cdot 2^{115} * 1.0 \cdot 2^{125}$  の単精度計算の場合)。

```

include <stdio.h>
#include <excpt.h>
void main () {
    float a, b, c; // single precision floating-point numbers
    unsigned int u; // unsigned integer (of size 4 bytes)
    char *pa, *pb, *pc; // pointers to single precision numbers
    unsigned short t[5], *pt;
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    psw = &sw;
    pcw = &cw;
    // clear exception flags, read status word, // set control word
    cw = 0x0337; // round to nearest, 64 bits, // overflow exceptions enabled

```

```

__asm {
    fclex
    mov eax, DWORD PTR pcw
    fldcw [eax]
    mov eax, DWORD PTR psw
    fstsw [eax]
}
printf ("BEFORE COMP. sw = %4.4x\n", sw);
pa = (char *)&a; u = 0x79000000;
a = *(float *)&u; // a = 1.0 * 2^115
pb = (char *)&b; u = 0x7e000000;
b = *(float *)&u; // b = 1.0 * 2^125
pc = (char *)&c;
c = 0.0;
pt = t;
__try {
    printf ("TRY BLOCK BEFORE OVERFLOW\n");
    // compute c = a * b
    __asm {
        mov eax, DWORD PTR pa
        fld DWORD PTR [eax] // push a on the FPU stack
        mov eax, DWORD PTR pb
        fld DWORD PTR [eax] // push b on the FPU stack
        fmulp st(1), st(0) // a * b in st(1), pop st(0)
        // cause the overflow exception
        mov eax, DWORD PTR pc
        fstp DWORD PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
        fwait // trigger floating-point exception if any
    }
    printf ("TRY BLOCK AFTER OVERFLOW\n");
} __except(EXCEPTION_EXECUTE_HANDLER) {
    printf ("EXCEPT BLOCK\n");
    // clear exception flags, read status word, // set control word
    cw = 0x033f; // round to nearest, 64 bits, // exceptions disabled
    __asm {
        mov eax, DWORD PTR psw
        fnstsw [eax]
        fnclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
    }
    printf ("sw = %4.4x\n", sw); // sw=0xb888: B=1, TOP=111, ES=1, OE=1
    __asm {
        mov eax, DWORD PTR pt
        fstp TBYTE PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
    }
    printf ("t = %4.4x%4.4x%4.4x%4.4x%4.4x\n", t[4],t[3],t[2],t[1],t[0]);
    // t = 2^240
}
}

```

出力は、FPU ステータス・ワードの値( $sw=0xb888$ )を示す(ステータス・ワードは、 $B=1$ 、 $TOP=111$ 、 $ES=1$ 、および  $OE=1$  に設定される)。結果( $0x40ef8000000000000000$ 、 $2^{240}$  を拡張倍精度でコード化した値)は、FPU スタック上で変更されない。

```
BEFORE COMP. sw = 0000
TRY BLOCK BEFORE OVERFLOW
EXCEPT BLOCK
sw = b888
t = 40ef8000000000000000
```

オーバーフロー浮動小数点例外を発生させる命令は、FSTP である。FSTP が FPU スタック上の値を 32 ビット単精度メモリ形式に変換しようとするときに、例外が発生する。

### 例 9: 例外の原因になる命令が FPU スタック・レジスタをデスティネーションとする場合のマスクされていないオーバーフロー例外の発生

オーバーフロー浮動小数点例外トラップを発生させる第 2 の例では、例外の原因となる命令は FPU スタック上の位置をデスティネーションとする(プラス無限大方向への丸めを使用した、 $1.0...01 \cdot 2^{16000} * 1.0...01 \cdot 2^{16000}$  の倍精度計算の場合)。

```
#include <stdio.h>
#include <float.h>
#include <excpt.h>
void main () {
    unsigned short a[5], b[5], c[5], *pa, *pb, *pc;
    unsigned short sw, *psw; // status word and pointer to it
    unsigned short cw, *pcw; // control word and pointer to it
    psw = &sw;
    pcw = &cw;
    // clear exception flags, read status word, // set control word
    cw = 0x0b37; // round up, 64 bits, overflow exc. enabled
    __asm {
        fclex
        mov eax, DWORD PTR pcw
        fldcw [eax]
        mov eax, DWORD PTR psw
        fstsw [eax]
    }
    printf ("BEFORE COMP. sw = %4.4x\n", sw);
    // a = 1.0 * 2^16000, b = 1.0 * 2^16000
    a[4] = 0x7e7f; a[3] = 0x8000; a[2] = 0x0000; a[1] = 0x0000; a[0] = 0x0001;
    b[4] = 0x7e7f; b[3] = 0x8000; b[2] = 0x0000; b[1] = 0x0000; b[0] = 0x0001;
    pa = a; pb = b; pc = c;
    __try {
        printf ("TRY BLOCK BEFORE OVERFLOW\n");
        // compute c = a * b
        __asm {
            mov eax, DWORD PTR pa
            fld TBYTE PTR [eax] // push a on the FPU stack
            mov eax, DWORD PTR pb
            fld TBYTE PTR [eax] // push b on the FPU stack
            fmulp st(1), st(0) // a * b in st(1), pop st(0)
            fwait // trigger floating-point exception if any
        }
    }
```

```

printf ("TRY BLOCK AFTER OVERFLOW\n");
} __except(EXCEPTION_EXECUTE_HANDLER) {
printf ("EXCEPT BLOCK\n");
// clear exceptions, read status word, set control word
cw = 0x0b3f; // round up, 64 bits, exceptions disabled
__asm {
mov eax, DWORD PTR psw
fnstsw [eax]
fnclex
mov eax, DWORD PTR pcw
fldcw [eax]
}
printf ("sw = %4.4x\n", sw); // sw=0xbaa8:
// B=1, TOP=111, C1=1, ES=1, PE=1, OE=1
__asm {
mov eax, DWORD PTR pc
fstp TBYTE PTR [eax] // c = a * b from FPU stack to memory, pop st(0)
}
printf ("c = %4.4x%4.4x%4.4x%4.4x%4.4x\n", c[4],c[3],c[2],c[1],c[0]);
// c = 2^32000 / 2^24576 = 2^7424 (biased exponent is 0x5cff)
}
}

```

出力は次のようになる。

```

BEFORE COMP. sw = 0000
TRY BLOCK BEFORE OVERFLOW
EXCEPT BLOCK
sw = baa8
c = 5cff80000000000000000003

```

この出力は、FPU スタック上の結果が  $2^{24576}$  でスケールダウンされたことを示している (0x5cff80000000000000000003 は、 $2^{32000-24576} = 2^{7424}$  を拡張倍精度でコード化した値である)。FPU ステータス・ワード (sw = baa8) は、B=1、TOP=111、C1=1、ES=1、PE=1、OE=1 に設定する (C1=1 は、結果の仮数部が限りなく正確な結果の仮数部より大きいという意味である)。

オーバーフロー浮動小数点例外を発生させる例外は、FMUL である。32000 の指数が、拡張倍精度形式の 15 ビットの指数範囲を超えるために、例外が発生する。

### 3 ストリーミング SIMD 浮動小数点命令

インテルのストリーミング SIMD 浮動小数点命令(SSE)(整数命令および浮動小数点命令)は、高度なマルチメディアおよび通信アプリケーションのパフォーマンスを向上させるために開発された。新しいレジスタ、データ型、および命令(浮動小数点命令だけではない)は、SIMD 実行モデルに統合され、アプリケーションの処理を高速化する。SSE で使用される基本的な浮動小数点データ型は、単精度データ型である。このデータ型は、表 1 に示した特性を持つ(このデータ型は、FPU の単精度メモリ形式と同じである)。つまり、このデータ型で表現できる値は、0、デノーマル数、正規化有限数、無限大、および NaN である。単精度データ型は、画像処理、オーディオ合成、音声認識、テレフォニー、2D および 3D グラフィックスなどのアプリケーションに効果的である。

#### 3.1 ストリーミング SIMD 浮動小数点命令アーキテクチャ

SSE 浮動小数点命令は、新しい 8 つの 128 ビット・レジスタ・セット(図 3)を操作する。各レジスタは直接にアドレス指定できる(FPU レジスタ・スタック・モデルでは、各レジスタを直接アドレス指定ができない。ただし、FXCH 命令によって、この制限を一部回避できる)。新しいレジスタはデータを保持するが、メモリのアドレス指定には使用できない(アドレス指定は、既存の IA-32 アドレス指定モードを使用して行う)。

XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

図 3: SIMD 浮動小数点レジスタ

各 SIMD レジスタは、4 つの単精度浮動小数点数(図 4 の X1、X2、X3、X4。X1 が最下位の位置を占める)で構成される、新しい形式のパックド 128 ビット・データを保持する。

127	96 95	65 64	32 31	0
X4	X3	X2	X1	

図 4: パックド単精度浮動小数点データ型

メモリ内では、この新しいデータ型はリトル・エンディアン形式で、連続する 16 バイトに格納される。しかし、メモリとレジスタの間またはレジスタ同士の間へのアクセスとデータ転送は、必ず 128 ビット転送または 32 ビット転送として行われる。アライメントの合っていないロー

ドおよびストアの場合を除いて、メモリ・オペランドは 16 バイトにアライメントしている必要がある。

SSE 浮動小数点命令は、スカラ命令ではパックド・オペランドの最下位オペランド(またはオペランドのペア)を操作し、並列(パックド)命令では 4 つのオペランドすべて(またはオペランドの一部)を操作する。操作の結果は、デスティネーション・レジスタまたはメモリ・ロケーション内の対応する位置に置かれる。スカラ演算の場合、上位 3 つの要素(ビット 127~32 を占める)は、第 1 ソース・オペランドからコピーされる。

### 3.2 SIMD コントロール/ステータス・レジスタ

SSE は、新しい 32 ビットのコントロール/ステータス・レジスタ(図 5)を使用する。ビット 31~16(図には示していない)とビット 6 は予約済みであり、0 でない値を書き込んで서는ならない。FPU 命令/ステートと SSE/ステートの間の相互作用は、それらを使用するアプリケーションのセマンティクスによって確立する。MMX テクノロジ・レジスタへの参照を含む SSE または SSE2(第 4 章)を検出すると、浮動小数点操作から MMX 操作への移行が発生するのに注意すること。この移行の際は、FPU ステータス・レジスタ内で TOP=0 に設定し、タグ・レジスタを 0(一杯になった FPU スタックを示す)に設定する。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FZ	RC	RC	PM	UM	OM	ZM	DM	IM	Res	PE	UE	OE	ZE	DE	IE

図 5: コントロール/ステータス・レジスタ MXCSR の下位 16 ビット

ビット 15~7 は、ゼロ・フラッシュ・モード、丸めモード、および浮動小数点例外のマスク/アンマスクをセットする。ビット 5~0 には、浮動小数点例外ステータス・フラグが入る。浮動小数点数を操作するすべての SSE は、常に単精度計算を実行する。したがって、MXCSR レジスタには、実行される浮動小数点計算の精度を指定する精度制御フィールド(PC)はない。

ビット 15 は、ゼロ・フラッシュ・モードを制御する。このビットを使用して、多くの極小値を生成するアプリケーションを高速化できる。(MXCSR の FZ ビットをセットすると)ゼロ・フラッシュ・モードを選択し、アンダーフロー例外がマスクされている場合は、浮動小数点命令によって生成する極小の結果は、正しい符号の 0 で置き換えられ、MXCSR 内のアンダーフロー・ステータス・フラグと不正確ステータス・フラグをセットする。パフォーマンス上の理由で、SSE 浮動小数点命令を利用するアプリケーションを実行するときは、ゼロ・フラッシュ・モードを選択するのを推奨する。FZ ビットがクリアされているか、アンダーフロー例外がアンマスクされている場合は、ゼロ・フラッシュ・モードは無効になる。

ビット 14 とビット 13 は、丸め制御フィールド(RC)である。このフィールドは、デスティネーションの単精度形式では結果を正確に表現できない場合に、どの IEEE 丸め法[1]を使用するかを指定する(RC=00B は最近値方向への丸め、RC=01B は切り捨て、RC=10B は切り上げ、RC=11B はゼロ方向への丸めを表す)。浮動小数点演算後の結果の丸めは、第 2 章で説明した方法で実行される。

ビット 12~7(PM、UM、OM、ZM、DM、IM)は、例外フラグ・マスクである。これらのビットがセットされている場合、SIMD 浮動小数点例外(それぞれ、不正確結果例外、アンダーフロー例外、オーバーフロー例外、ゼロ除算例外、デノーマル・オペランド例外、および無効操作例外)はマスクされる(無効にされる)。

ビット 5~0(PE、UE、OE、ZE、DE、IE)は、それぞれ、不正確結果例外、アンダーフロー例外、オーバーフロー例外、ゼロ除算例外、デノーマル・オペランド例外、および無効操作例外の例外フラグである。これらのフラグは、フラグが最後にクリアされた後、1つ以上の浮動小数点例外を検出したことを示す(これらは「スティッキー」ビットである)。FPU ステータス・ワード内のステータス・フラグとは異なり、MXCSR ステータス・フラグは、SSE 浮動小数点命令の単精度オペランドの4つのサブオペランドのうち任意の要素を計算した結果によってセットされる。命令が複数のサブオペランドを操作する場合は、ステータス・フラグの値は、各サブオペランドに対応する複数の仮定上のステータス・フラグの論理和(OR)になる。

### 例 10: ゼロ・フラッシュ・モードの影響

以下の計算は、ゼロ・フラッシュ・モードの影響の例を示す。SSE 浮動小数点命令を使用する場合は、2進浮動小数点演算に関する IEEE 規格で規定された純粋な単精度計算を1ステップで実行する。たとえば、MULSS を使用してスカラ単精度乗算を実行できる。FPU 計算では、これは不可能であり、2つの命令(FMUL と FST)を使用する必要がある。FPU 計算のメリットは、FMUL で得られる高精度の結果が中間計算の段階に便利なことである。たとえば、例 1 の2つの単精度浮動小数点数  $a$  と  $b$  の乗算の場合を考える。アンダーフロー例外と不正確結果例外がマスクされ、ゼロ・フラッシュ・モードを選択しているとする。

$$a = 1.1\dots10 \cdot 2^{-126} \approx 10^{-54.28}$$

$$b = 1.0\dots01 \cdot 2^{-1} \approx 0.5$$

仮数部 24 ビットへの丸めと有界でない指数を使用した仮定上の結果は、丸めモードによって異なる(以下の仮数部は 24 ビットである)。

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 最近値方向への丸めの場合(極小でない)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ 切り捨ての場合(極小)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 切り上げの場合(極小でない)}$$

$$a \cdot b = 1.11\dots1 \cdot 2^{-127} \text{ ゼロ方向への丸めの場合(極小)}$$

(仮数部 24 ビットへの丸めと有界の指数を使用した)実際の丸めの結果も、丸めモードによって異なる。ゼロ・フラッシュ・モードの影響も受けて(極小の結果が正しい符号の 0 にフラッシュされる)、以下の結果が得られる。

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 最近値方向への丸めの場合(P フラグをセット)}$$

$$a \cdot b = +0.0 \text{ 切り捨ての場合(P、U フラグをセット)}$$

$$a \cdot b = 1.0\dots0 \cdot 2^{-126} \text{ 切り上げの場合(P フラグをセット)}$$

$$a \cdot b = +0.0 \text{ ゼロ方向への丸めの場合(P、U フラグをセット)}$$

## 3.3 浮動小数点例外

SIMD 浮動小数点命令は、FPU 命令と同様に、無効操作、デノーマル・オペランド、ゼロ除算、オーバーフロー、アンダーフロー、および不正確結果(精度)の6種類の浮動小数点例外を発生させる。MXCSR 制御/ステータス・ワードには、各例外に対応するマスク/アンマスク(無効/有効)ビットと、「スティッキー」例外ステータス・フラグがある。

無効操作例外、デノーマル・オペランド例外、およびゼロ除算例外は、計算前型の例外(浮動小数点フォルト)である。オーバーフロー例外、アンダーフロー例外、および不正確結果例外は、計算後型の例外(浮動小数点トラップ)である。

マスクされた(マスク・ビットがセットされた)例外が検出されると、SIMD FPU はその例外を自動的に処理して、あらかじめ定義した結果を生成し、プログラムの実行を続ける。ただし、プロセッサは1つの命令を実行する間に、複数の浮動小数点例外を検出する場合がある。このため、MXCSR 内のステータス・フラグは、すべてのサブオペランド計算で発生した例外条件の論理和(OR)としてセットする。

マスクされていない(マスク・ビットがクリアされている)例外を検出すると、SIMD FPU は、SIMD 浮動小数点割り込みベクタ 19 によって、ソフトウェア例外ハンドラを起動し、例外を処理する。FPU 浮動小数点例外の場合とは異なり、この動作は他の SIMD 浮動小数点命令が例外を発生するのを待たずにただちに行われる。プロセッサはソース・レジスタ・オペランドを、ソフトウェア・ハンドラを起動する前の状態のまま変更しない。同様に、COMISS または UCOMISS(以下を参照)によってマスクされていない例外が発生した場合は、EFLAGS レジスタを修正しない。この動作は、計算後型例外の x87 モデルとは異なる(x87 モデルでは、ソフトウェア・ハンドラに結果が渡される)。SSE 浮動小数点命令によって SIMD 例外が発生した場合は、その例外が計算前型か計算後型かを問わず、結果は例外ハンドラに渡されない。マスクされていない計算後型の例外の場合、MXCSR 内の例外ステータス・フラグは、すべてのサブオペランド計算で発生した例外条件の論理和(OR)によって更新される(計算前型の例外条件の場合もフラグは更新されるが、それらの例外はすべてマスクされていなければならない)。

SSE の例外条件は FPU の例外条件と同様であるが、単精度形式でサポートされないオペランドは存在しない(したがって、サポートされないオペランドが原因で無効操作例外は発生しない)。

SIMD 浮動小数点例外に優先する条件は、FPU 例外の場合と同様である。

- 無効操作例外(SNaN オペランド、一部の浮動小数点比較命令では NaN)、または許容範囲外のオペランド
- QNaN オペランド(これは例外ではないが、QNaN オペランドの処理は優先順位の低い例外に優先する)。
- 上記以外の無効操作例外、またはゼロ除算例外。
- デノーマル・オペランド例外(マスクされている場合は、実行が続けられ、より優先順位の低い例外が発生する場合がある)。
- 数値オーバーフローまたはアンダーフロー例外。通常は不正確結果例外と一緒に発生する。
- 不正確結果例外

ただし、MXCSR 内のステータス・フラグの値をセットした後、それに対応する浮動小数点例外をアンマスクしても、ソフトウェア例外ハンドラは起動されない。この動作は、FPU ステータス・ワードおよび制御ワードの場合とは異なる。この場合、FPU では後続する最初の「待機型」浮動小数点命令の実行時に、浮動小数点例外を処理する。

FPU 命令で発生する浮動小数点例外と SSE 浮動小数点命令で発生する浮動小数点例外の 2 番目の重要な相違は、SIMD 命令ではマスクされていない例外が、次の浮動小数点命令を待たずに、ただちに報告される点である。

3 番目の相違点として、SSE 浮動小数点ロード/ストア命令は FPU ロード/ストア命令とは異なり、数値例外を発生させない。この理由の 1 つは、SIMD ロード/ストアではデータ型変換が行われないため、オーバーフロー例外、アンダーフロー例外、および不正確結果例外が起こり得ないことである。

ただし、極小の結果が 0 にフラッシュされる(つまり、MXCSR 内で FZ ビットと UM ビットがセットされている)場合、不正確トラップがアンマスクされて(PM ビットがクリアされて)いれば、不正確結果例外に対してソフトウェア例外ハンドラを起動する。例外ハンドラに渡される結果は、正しい符号の 0 である。

最後に、一部の命令(以下で説明する COMISS と UCOMISS)は、処理の結果を EFLAGS レジスタに書き込む。マスクされていない浮動小数点例外(無効操作例外またはデノーマル・オペランド例外)を検出した場合は、EFLAGS の値は更新されない。

### 3.4 ソフトウェア例外処理

SSE 浮動小数点命令のソフトウェア・ハンドラが実行する一般的な動作は、FPU 命令のソフトウェア・ハンドラの動作とよく似ている。ただし、SSE 浮動小数点命令では、以前にステータス・フラグをセットした浮動小数点例外をアンマスクしてもソフトウェア・ハンドラは起動しないため、MXCSR ステータス・フラグをクリアする必要はない。パックド・オペランドを操作する SSE 浮動小数点命令で発生した、マスクされていない浮動小数点例外を処理する場合の主な問題は、MXCSR 内のステータス・フラグからは、4 つのサブオペランドのうちどれが例外を発生させたかが不明な点である。また、マスクされていない x87 例外とは異なり、浮動小数点トラップ(オーバーフロー、アンダーフロー、および不正確)が発生した場合は、結果はデスティネーション SIMD レジスタに返されない。例外を正しく処理するには(また、IEEE 規格 [1] の推奨事項に適合するには)、ソフトウェアはマスクされていない例外を発生させた命令をサブオペランドごとにエミュレートし、4 つの要素のうちどの要素がその例外を発生させたかを特定する必要がある(1 つの命令で、マスクされていない例外が 2 つ以上発生する場合もある)。例外に対応するオペランドの要素と、操作の結果(浮動小数点トラップの場合)が、ユーザ・ハンドラに渡される。例外を発生させた命令を含むアプリケーションの実行を続ける場合は、ユーザ・ハンドラが返した結果と、例外を発生させなかった残りのサブオペランド(存在する場合)についてエミュレートした結果を組み合わせ、割り込みをかけられたアプリケーションの実行を再開する必要がある。

### 3.5 NaN の処理

FPU 命令の場合と同様に、ほとんどの SIMD 浮動小数点命令では、QNaN が検出されても浮動小数点例外は発生しない(浮動小数点最大/最小命令と、一部の浮動小数点比較命令を除く)。ただし、結果の生成規則は、(表 2 に示した)FPU 命令の規則とは異なる。すでに説明したように、NaN の処理は、一部の浮動小数点例外に優先する。表 3 に、SSE の QNaN 結果の生成規則を示す。

表 3: SSE の QNaN 結果の生成規則

ソース・オペランド	QNaN の結果
SNaN と QNaN	ソース 1 の NaN(ソース 1 が SNaN の場合は QNaN に変換する)
2 つの SNaN	ソース 1 の NaN(QNaN に変換する)
2 つの QNaN	ソース 1 の NaN
SNaN と実数値、オペランドが 1 つだけの場合は SNaN	SNaN が QNaN(クワイエット NaN)に変換する
QNaN と実数値、オペランドが 1 つだけの場合は QNaN	QNaN ソース・オペランド
NaN オペランドはないが、無効操作例外が報告される	QNaN 実数不定値

### 3.6 ストリーミング SIMD 浮動小数点命令

SSE は、主に浮動小数点(算術)命令で構成されるが、すべての MMX テクノロジ・データ型および 32 ビット IA-32 データ型のキャッシュを制御する命令(本書では説明しない。[2]を参照)も含まれる。パックド・オペランドの 4 つの要素すべてを操作する命令には、サフィックス“PS”(“packed single precision”)が付く。パックド・オペランドの最下位の要素だけを操作する命令には、サフィックス“SS”(“scalar single precision”)が付く。パックド命令がメモリからオペランドを読み出す場合は、暗黙的に 128 ビットの読み出しを実行する。スカラー命令がメモリからオペランドを読み出す場合は、暗黙的に 32 ビットの読み出しを実行する。浮動小数点数を操作する SSE を以下に示す。

#### 1. データ移動命令

- MOVAPS/MOVUPS: move aligned/unaligned packed single precision floating-point; メモリと SIMD 浮動小数点レジスタの間または SIMD レジスタ同士の間で、128 ビットのデータを転送する。浮動小数点例外: なし
- MOVHPS/MOVLPS: move aligned, high/low packed single precision floating-point; メモリと SIMD 浮動小数点レジスタの上位/下位半分の間で、64 ビットのデータを転送する(下位/上位半分は変更されない)。浮動小数点例外: なし

- MOVHLP/MOVLHP: move high/low to low/high packed single precision floating-point; ソース・レジスタの上位/下位 64 ビットを、デスティネーション・レジスタの下位/上位 64 ビットに転送する(上位/下位 64 ビットは変更されない)。浮動小数点例外: なし
- MOVMSKPS: move mask packed, single precision floating-point to r32; 4 つのパックド・オペランドの最上位ビットを、32 ビット IA-32 整数レジスタ r32 に転送する。浮動小数点例外: なし
- MOVSS: move scalar single precision floating-point; メモリまたは SIMD レジスタの最下位 32 ビットと、SIMD レジスタの最下位 32 ビットの間で、32 ビットのデータを転送する。浮動小数点例外: なし

## 2. 算術命令

- ADDPS/ADDSS/SUBPS/SUBSS/MULPS/MULSS: add/subtract/multiply packed/scalar, single precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。浮動小数点例外: I、D、O、U、P
- DIVPS/DIVSS: divide packed/scalar, single precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。浮動小数点例外: I、Z、D、O、U、P
- SQRTPS/SQRTSS: square root packed/scalar, single precision floating-point; ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。デスティネーションは SIMD レジスタである。浮動小数点例外: I、D、P

## 3. 最大命令と最小命令

- MAXPS/MAXSS/MINPS/MINSS: maximum/minimum packed/scalar, single precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。浮動小数点例外: I、D(無効操作例外は、任意の NaN オペランドによって発生する)

## 4. 比較命令

- CMPPS/CMPSS: compare packed/scalar, single precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。結果は 1(真)または 0(偽)の 32 ビット・マスクである。浮動小数点例外: I、D(無効操作例外は、lt、le、nlt、nle の比較では、任意の NaN オペランドによって発生する。それ以外の比較では、SNaN でのみ発生する)
- COMISS/UCOMISS: compare scalar single precision floating-point ordered/unordered and set EFLAGS; 第 1 ソース・オペランドとデスティネーションは、SIMD レジスタの最下位の要素である。第 2 ソース・オペランドは、SIMD レジスタの最下位の要素またはメモリ・ロケーションである。比較の結果は、EFLAGS レジスタの ZF、PF、および CF ビットに書き込まれる。浮動小数点例外: I、D(無効操作例外は、COMISS では任意の NaN オペランドによって発生する。UCOMISS では SNaN オペランドで発生する)

## 5. 変換命令

- CVTPI2PS: MMX テクノロジ・レジスタまたはメモリ内の 2 つのパックド 32 ビット符号付き整数を、SIMD レジスタの(下位 2 つの位置の)2 つの単精度浮動小数点数に変換する。浮動小数点例外: P
- CVTSI2SS: 整数レジスタまたはメモリ内の 1 つの 32 ビット符号付き整数を、SIMD レジスタの(最下位の位置の)1 つの単精度浮動小数点数に変換する。浮動小数点例外: P
- CVTTPS2PI/CVTTSS2PI: SIMD レジスタ内の下位 2 つのパックド単精度浮動小数点数またはメモリ内の 2 つの単精度浮動小数点数を、MMX テクノロジ・レジスタ内の 2 つのパックド 32 ビット符号付き整数に変換する。CVTTSS2PI は、MXCSR 内で指定された丸めモードではなく、ゼロ方向への丸めモード(切り捨て)を使用する。浮動小数点例外: I、P
- CVTSS2SI/CVTTSS2SI: SIMD レジスタの最下位の位置の単精度浮動小数点数またはメモリ内の 1 つの単精度浮動小数点数を、整数レジスタ内の 32 ビット符号付き整数に変換する。CVTTSS2SI は、MXCSR 内で指定された丸めモードではなく、ゼロ方向への丸めモード(切り捨て)を使用する。浮動小数点例外: I、P

## 6. 論理命令(ビット単位)

- ANDPS/ANDNPS/ORPS/XORPS: packed logical AND, AND-NOT, OR, XOR; 浮動小数点例外: なし

## 7. 逆数近似命令

- RCPPS/RCPSS: packed/scalar, single precision floating-point reciprocal approximation(相対誤差は最大  $1.5 \cdot 2^{-12}$ ); ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。デスティネーションは SIMD レジスタである。浮動小数点例外: なし
- RSQRTPS/RSQRTSS: packed/scalar, single precision floating-point square root reciprocal approximation(相対誤差は最大  $1.5 \cdot 2^{-12}$ ); ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。デスティネーションは SIMD レジスタである。浮動小数点例外: なし

## 8. ステート管理命令

- FXSAVE/FXRSTOR: メモリ内の 512 バイト領域との間で、FP/MMX テクノロジ・ステートと SIMD ステートをセーブ/リストアする。対象となるステートは、CS(コード・セグメント・ディスクリプタ)、IP(命令ポインタ)、FOP(浮動小数点オペレーション・コード)、FTW(FPU タグ・ワード)、FSW(FPU ステータス・ワード)、FCW(FPU 制御ワード)、MXCSR(SIMD 制御/ステータス・ワード)、DS(データ・セグメント・ディスクリプタ)、DP(データ・ポインタ)、8 つの FPU スタック/MMX テクノロジ・レジスタ、8 つの SIMD レジスタである。浮動小数点例外: なし
- STMXCSR/LDMXCSR: 32 ビットのメモリ・ロケーションとの間で、SIMD 制御/ステータス・ワードをストア/ロードする。浮動小数点例外: なし

FXSAVE と FXRSTOR は、それぞれ FSAVE と FRSTOR に代わるものであり、より多くの情報をより高速にセーブ/リストアするように最適化されている。

その他の SSE には、整数データを操作する命令や、SIMD レジスタ同士の間または SIMD レジスタとメモリ・ロケーションの間で(デスティネーションは常に SIMD レジスタである)、単精度浮動小数点数に対応する 32 ビット・フィールドをシャッフルする命令がある。多くの SSE は、x87 浮動小数点命令が検出された場合、元の MMX 命令と同じように動作する。最後に、キャッシュ制御命令は、MMX テクノロジ・レジスタまたは SIMD 浮動小数点レジスタからメモリにデータを書き込む際に、キャッシュ汚染を最小限に抑えるように、プログラム上で制御できる。

### 3.7 SSE を使用したアプリケーションの開発

SSE 浮動小数点命令は、すべての IA-32 実行モード(プロテクト・モード、実アドレス・モード、および仮想 8086 モード)から利用できる。ただし、アプリケーションは、SSE 浮動小数点命令を使用する前に、プロセッサとオペレーティング・システムが SSE をサポートしていることを確認しなければならない。

- エミュレーションが無効にされている: CR0.EM(ビット 2)=0
- プロセッサが SSE をサポートしている: CPUID.XMM(EDX ビット 25)=1
- プロセッサが FXSAVE/FXRSTOR をサポートしている: CPUID.FXSR(EDX ビット 24)=1
- OS がコンテキスト・スイッチ時の SIMD FP ステートをサポートしている: CR4.OSFXSR(ビット 9)=1

さらに、追加された非浮動小数点 SIMD 命令のサポートの有無を確認するためのチェックが必要である(詳細は[2]を参照)。また、SIMD 浮動小数点例外をアンマスクする場合は、アプリケーションは、オペレーティング・システムがアンマスクされた SSE 例外をサポートしているのを確認しなければならない。

- OS がアンマスクされた SIMD 例外をサポートしている: CR4.OSXMMEXCPT(ビット 10)=1

### 3.8 例

次の 2 つの例は、SSE を使用した簡単な計算を示している。

#### 例 11: SSE の使用

最初の例では、ゼロ・フラッシュ・モードを選択し、丸めモードを最近値方向に設定し、浮動小数点例外をマスクした条件で、SIMD オペランド(1.0, 1.0, 1.0, 1.0)を(0.0...01 · 2<sup>-126</sup>, 0.0, 1.1...11 · 2<sup>127</sup>, SNaN)で割る。(左側の要素から順に)第 1 の除算で、MXCSR 内のデノーマル・オペランド・ステータス・フラグをセットし、さらにオーバーフロー・ステータス・フラグをセットする(結果が単精度浮動小数点数の最大値より大きくなる)。第 2 の除算で、ゼロ除算例外が発生し、対応するステータス・フラグをセットする。第 3 の除算で、極小かつ不正確な結果を生成するが、ゼロ・フラッシュ・モードが選択されているため、結果は+0.0で置き換えられ、MXCSR 内のアンダーフロー・ステータス・フラグと精度ステータス・フラグをセットする。最後の除算で、無効操作例外が発生し、MXCSR 内の無効ステータス・フラグがセットされる。この SIMD 演算の結果は、(+inf, +inf, 0.0, QNaN)になる。結果の QNaN は、第 1 入力オ

ペランドの SNaN をクワイエット NaN に変換したものである。MXCSR のすべてのステータス・フラグをセットするが、MXCSR レジスタは入力オペランドの値を知らないため、どのサブオペランドがどの例外を発生させたかは不明である。例外をアンマスクした場合は、ソフトウェア・ハンドラは正しい結果を求めるために、命令の実行を一度に 1 要素ずつエミュレートする必要がある。

```
#include <stdio.h>
void main () {
    char *mem;
    unsigned int uimem[4];
    int mxcsr, *pmxcsr;
    mem = (char *)uimem;
    // set and then read new value of MXCSR
    mxcsr = 0x00009f80;
    // ftz = 1, rc = 00 (to nearest), traps disabled, flags clear
    pmxcsr = &mxcsr;
    __asm {
        mov eax, DWORD PTR pmxcsr
        ldmxcsr [eax]
        stmxcsr [eax]
    }
    printf ("BEFORE SIMD DIVIDE: MXCSR = 0x%8.8x\n", mxcsr);
    // load first set of operands
    uimem[0] = 0x3f800000; // 1.0
    uimem[1] = 0x3f800000; // 1.0
    uimem[2] = 0x3f800000; // 1.0
    uimem[3] = 0x3f800000; // 1.0
    __asm {
        mov eax, DWORD PTR mem;
        movups XMM1, [eax];
    }
    // load second set of operands
    uimem[0] = 0x00000001; // 0.0...01 * 2^-126
    uimem[1] = 0x00000000; // 0.0
    uimem[2] = 0x7f7fffff; // 1.1...1 * 2^127
    uimem[3] = 0x7fbf0000; // SNaN
    __asm {
        mov eax, DWORD PTR mem;
        movups XMM2, [eax];
    }
    // perform SIMD divide and store result to memory
    __asm {
        divps XMM1, XMM2;
        mov eax, DWORD PTR mem;
        movups [eax], XMM1;
    }
    // read new value of MXCSR
    __asm {
        mov eax, DWORD PTR pmxcsr
        stmxcsr [eax]
    }
    printf ("AFTER SIMD DIVIDE: MXCSR = 0x%8.8x\n", mxcsr);
    printf ("res = %8.8x %8.8x %8.8x %8.8x = %f %f %f %f\n",
```

```

    uimem[0], uimem[1], uimem[2], uimem[3],
    *(float *)&uimem[0], *(float *)&uimem[1],
    *(float *)&uimem[2], *(float *)&uimem[3]);
}

```

The output is:

```

BEFORE SIMD DIVIDE: MXCSR = 0x00009f80
AFTER SIMD DIVIDE:  MXCSR = 0x00009fbf
Res = 7f800000 7f800000 00000000 7fff0000 =
      1.#INF00 1.#INF00 0.000000 1.#QNaN0

```

この例では、MOVUPS を使用して SIMD レジスタとメモリの間で SIMD オペランドを移動したため、メモリ・オペランドが 16 バイトにアライメントされている必要はない。次の例では、MOVAPS を使用する。この場合は、メモリ・オペランドが 16 バイトにアライメントされている必要がある。

### 例 12: SSE を使用した $1.0 / (\text{sqrt}(a) - 1.0)$ の例

2 番目の例では、 $1.0 / (\text{sqrt}(a) - 1.0)$  の簡単な計算を示す。a の 4 つの値のうち 1 つは、1.0 に非常に近い値である ( $a = 1.0...01 \cdot 2^0 = 1 + 2^{-23}$ )。この場合、(テイラー級数展開によって得られる) 正確な結果は、次のようになる。

$$R = (1 + 2^{-25} - 2^{-50} + 2^{-74} - 2^{-97} + \dots) \cdot 2^{24}$$

単精度オペランド  $a = 1.0...01$  は、計算の始めに XMM1 の最下位の要素に格納される。

```

#include <stdio.h>
void main () {
    char *mem;
    unsigned int *uimem;
    mem = (char *)(((int)malloc (144) + 16) & ~0x0f); // 16-byte aligned
    uimem = (unsigned int *)mem;
    // load x[i] in XMM1, i = 0,3
    uimem[0] = 0x40000000; // 2.0
    uimem[1] = 0x40400000; // 3.0
    uimem[2] = 0x40800000; // 4.0
    uimem[3] = 0x3f800001; // 1.0 + 1 ulp (1.0 + 2^-23)
    __asm {
        mov eax, DWORD PTR mem;
        movaps XMM1, [eax];
    }
    // load y[i] = 1.0 in XMM2, i = 0,3
    uimem[0] = 0x3f800000; // 1.0
    uimem[1] = 0x3f800000; // 1.0
    uimem[2] = 0x3f800000; // 1.0
    uimem[3] = 0x3f800000; // 1.0
    __asm {
        mov eax, DWORD PTR mem;
        movaps XMM2, [eax];
    }
    // calculate 1.0 / (sqrt(x[i]) - 1.0), i = 0,3
    __asm {
        // calculate sqrt(x[i]) in XMM1, i = 0,3
        sqrtps XMM1, XMM1;
        // calculate sqrt(x[i]) - 1.0 in XMM1, i = 0,3

```

```
    subps XMM1, XMM2;
    // calculate 1.0 / (sqrt (x[i]) - 1.0) in XMM2, i = 0,3
    divps XMM2, XMM1;
    // store result in memory
    mov eax, DWORD PTR mem;
    movaps [eax], XMM2;
}
printf ("res = %8.8x %8.8x %8.8x %8.8x = %f %f %f %f\n",
        uimem[0], uimem[1], uimem[2], uimem[3],
        *(float *)&uimem[0], *(float *)&uimem[1],
        *(float *)&uimem[2], *(float *)&uimem[3]);
}
```

出力は次のようになる。

```
res = 401a827a 3faed9ec 3f800000 7f800000 =
      2.414214 1.366025 1.000000 1.#INF00
```

この出力は、 $a = 1 + 2^{-23}$  の計算でオーバーフローの発生を示している。この場合、正確な結果を得るには、単精度の範囲では不十分である(この問題がアプリケーションに影響を与える場合)。次の章の例では、SSE2 と拡張倍精度形式を使用して、同じ計算をもう一度実行する。

## 4 ストリーミング SIMD 浮動小数点命令

インテルのストリーミング SIMD 浮動小数点命令(SSE2)は、従来の世代の IA プロセッサより MMX テクノロジー/科学計算アプリケーションのパフォーマンス向上のために開発された。SSE2 の多くの命令は、浮動小数点データを操作する。SSE2 のプログラミング・モデルは、MMX テクノロジー命令と SSE のプログラミング・モデルによく似ているが、SSE2 は、2つの新しいデータ型(128 ビット幅の整数と、2つの倍精度値で構成される 128 ビット幅の浮動小数点数)を操作する。新しい浮動小数点命令は、SIMD 実行モデルに統合され、アプリケーションの処理を高速化する。SSE2 で使用する基本的な浮動小数点データ型は、倍精度データ型である。このデータ型は、表 1 に示した特性を持つ(このデータ型は、FPU の倍精度メモリ形式と同じ形式である)。つまり、このデータ型で表現できる値は、0、デノーマル数、正規化有限数、無限大、および NaN である。倍精度データ型が採用されたのは、多くの技術/科学計算アプリケーションに便利だからである。より高い精度またはより広い有効範囲が必要な計算には、FPU 命令で提供される拡張倍精度形式を使用する必要がある。

### 4.1 ストリーミング SIMD 浮動小数点命令アーキテクチャ

浮動小数点計算用の SSE2 は、SSE 浮動小数点命令(図 3)と同じ 8 つの 128 ビット SIMD 浮動小数点レジスタ・セット(XMM レジスタ)を操作する。各レジスタは直接アドレス指定できる。コンテキスト・スイッチの際に SSE2 ステートをセーブ/リストアするには、OS が SSE ステートのセーブ/リストアをサポートしていれば十分であり、それ以外の新しいサポートは不要である。

各 SIMD レジスタは、2つの倍精度浮動小数点数(図 6 の X1、X2。X1 が下位の位置を占める)で構成される新しい形式のパックド 128 ビット・データを保持する。

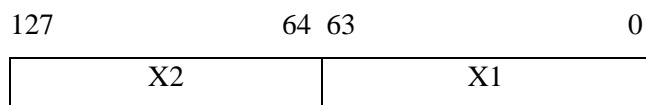


図 6: パックド倍精度浮動小数点データ型

メモリ内では、この新しいデータ型はリトル・エンディアン形式で、連続する 16 バイトに格納する。しかし、メモリとレジスタの間またはレジスタ同士の間へのアクセスとデータ転送は、必ず 128 ビット転送または 64 ビット転送として行われる。アライメントの合っていないロードおよびストアの場合を除いて、メモリ・オペランドは 16 バイトにアライメントしている必要がある。

### 4.2 SIMD 制御/ステータス・レジスタ

SSE で導入した浮動小数点制御/ステータス・レジスタ(MXCSR)は、SSE2 にも使用される。浮動小数点数を操作する SSE2 は、常に倍精度計算を実行する。したがって、MXCSR レジスタには、実行される浮動小数点計算の精度を指定する精度制御フィールド(PC)はない。

例外フラグは、フラグが最後にクリアされた後、1 つ以上の浮動小数点例外を検出したことを示す(これらは「スティッキー」ビットである)。SSE の場合と同様に MXCSR ステータス・フラグは、SSE2 浮動小数点算術命令の倍精度オペランドのうち任意の 1 つまたは 2 つの要素を計算した結果としてセットする。命令が 2 組のサブオペランドを操作する場合は、ステータ

ス・フラグの値は、各サブオペランドに対応する 2 つの仮定上のステータス・フラグの論理和 (OR) になる。

SSE2 浮動小数点命令は、スカラ命令ではパックド・オペランドの下位オペランド(またはオペランドのペア)を操作し、並列(パックド)命令では上位と下位のオペランド(またはオペランドの一部)を操作する。操作の結果は、デスティネーション・レジスタまたはメモリ・ロケーション内の対応する位置に置かれる。スカラ演算の場合、上位の要素(ビット 64 ~ 127 を占める)は、第 1 ソース・オペランドからコピーされる。

### 4.3 浮動小数点例外

SSE2 は、FPU 命令および SSE と同様に、無効操作、デノーマル・オペランド、ゼロ除算、オーバーフロー、アンダーフロー、および不正確結果(精度)の 6 種類の浮動小数点例外を発生させる。MXCSR 制御/ステータス・ワードには、各例外に対応するマスク/アンマスク(無効/有効)ビットと、「スティッキー」例外ステータス・フラグ・ビットがある(MXCSR レジスタの使用方法は、SSE と同じである)。

無効操作例外、デノーマル・オペランド例外、およびゼロ除算例外は、計算前型の例外(浮動小数点フォルト)である。オーバーフロー例外、アンダーフロー例外、および不正確結果例外は、計算後型の例外(浮動小数点トラップ)である。

マスクされた(マスク・ビットがセットされた)例外を検出すると、SIMD FPU はその例外を自動的に処理して、あらかじめ定義された結果を生成し、プログラムの実行を続ける。ただし、プロセッサは 1 つの命令を実行する間に、複数の浮動小数点例外を検出する場合がある。このため、MXCSR 内のステータス・フラグは、すべてのサブオペランド計算で発生した例外条件の論理和(OR)としてセットされる。

マスクされていない(マスク・ビットがクリアされている)例外を検出すると、SIMD FPU は、SIMD 浮動小数点割り込みベクタ 19 によって、ソフトウェア例外ハンドラを起動し、例外を処理する。FPU 浮動小数点例外の場合とは異なり、この動作は、他の SIMD 浮動小数点命令が例外を発生させるのを待たずに、ただちに行われる。プロセッサは、ソース・レジスタ・オペランドを、ソフトウェア・ハンドラを起動する前の状態のまま変更しない。同様に、COMISS または UCOMISS(以下を参照)によってマスクされていない例外が発生した場合は、EFLAGS レジスタは修正されない。この動作は計算後型例外の FPU モデルとは異なる(FPU モデルでは、ソフトウェア・ハンドラに結果が渡される)。SSE2 によって SIMD 例外が発生した場合は、その例外が計算前型か計算後型かを問わず、結果は例外ハンドラに渡されない。マスクされていない計算後型の例外の場合、MXCSR 内の例外ステータス・フラグは、すべてのサブオペランド計算で発生した例外条件の論理和(OR)によって更新する(計算前型の例外条件の場合もフラグは更新するが、それらの例外はすべてマスクされていない)。

SSE2 の例外条件は FPU の例外条件と同様であるが、倍精度形式でサポートされないオペランドは存在しない(したがって、サポートされないオペランドが原因で無効操作例外が発生しない)。

SSE2 浮動小数点例外に優先する条件は、FPU 例外の場合と同様であり、SSE 浮動小数点例外についてすでに挙げた条件と同じである。

ただし、MXCSR 内のステータス・フラグの値をセットした後、それに対応する浮動小数点例外をアンマスクしても、ソフトウェア例外ハンドラは起動しない。この動作は、SSE と同じで

あり、FPU ステータス・ワードおよび制御ワードの場合とは異なる(この場合、FPU では、次の「待機型」命令で浮動小数点例外が処理される)。

FPU 命令で発生する浮動小数点例外と SSE2(および SSE 浮動小数点命令)で発生する浮動小数点例外の 2 つ目の重要な相違は、SIMD 命令では、マスクされていない例外が、次の浮動小数点命令を待たずに、ただちに報告される点である。

第 3 の相違点として、浮動小数点値をロード/ストアする SSE2 は、FPU ロード/ストア命令とは異なり、SSE と同様に数値例外を発生させない。この理由の 1 つは、SIMD ロード/ストアではデータ型変換が行われなため、オーバーフロー例外、アンダーフロー例外、および不正確結果例外が起こり得ないことである。

ただし、極小の結果が 0 にフラッシュされる(つまり、MXCSR 内で FZ ビットと UM ビットがセットされている)場合、不正確トラップがアンマスクされて(PM ビットがクリアされて)いれば、不正確結果例外に対してソフトウェア例外ハンドラを起動する。例外ハンドラに渡される結果は、正しい符号の 0 である。

最後に、一部の命令(以下で説明する COMISS と UCOMISS)は、処理の結果を EFLAGS レジスタに書き込む。マスクされていない浮動小数点例外(無効操作例外またはデノーマル・オペランド例外)を検出した場合は、EFLAGS の値は更新されない。

## 4.4 ソフトウェア例外処理

SSE2 のソフトウェア・ハンドラが実行する一般的な動作は、FPU 命令や SSE のソフトウェア・ハンドラの動作とよく似ている。ただし、SSE2 では、以前にステータス・フラグがセットされた浮動小数点例外をアンマスクしてもソフトウェア・ハンドラは起動しないため、MXCSR ステータス・フラグをクリアする必要はない。パックド・オペランドを操作する SSE2 で発生した、マスクされていない浮動小数点例外を処理する場合の主な問題は、SSE の場合と同様に、MXCSR 内のステータス・フラグからは、2 つのサブオペランドのうちどちらが例外を発生させたかが不明な点である。また、マスクされていない x87 例外とは異なり、浮動小数点トラップ(オーバーフロー、アンダーフロー、および不正確)が発生した場合は、結果はデスティネーション SIMD レジスタに返されない。例外を正しく処理するには(また、IEEE 規格[1]の推奨事項に適合するには)、ソフトウェアは、マスクされていない例外を発生させた命令をサブオペランドごとにエミュレートし、2 つの要素のうちどちらがその例外を発生させたかを特定する必要がある。例外に対応するオペランドの要素と、操作の結果(浮動小数点トラップの場合)が、ユーザ・ハンドラに渡される。例外を発生させた命令を含むアプリケーションの実行を続ける場合は、ユーザ・ハンドラが返した結果と、例外を発生させなかった残りのサブオペランド(存在する場合)についてエミュレートした結果を組み合わせ、割り込みをかけたアプリケーションの実行を再開する必要がある。

## 4.5 NaN の処理

FPU 命令および SSE の場合と同様に、浮動小数点値を操作する SSE2 のほとんどでは、QNaN が検出されても浮動小数点例外は発生しない(浮動小数点最大/最小命令と、一部の浮動小数点比較命令を除く)。SSE2 例外では、表 2 に示した FPU 命令の QNaN 結果の生成規則ではなく、表 3 に示した SSE 例外の規則が適用される。既に説明したように、NaN の処理は、一部の浮動小数点例外に優先する。

## 4.6 SSE2

SSE2 は主に浮動小数点(算術)命令で構成されるが、すべての MMX テクノロジ・データ型および 32 ビット IA-32 データ型のキャッシュを制御する命令(本書では説明しない。[2]を参照)も含まれる。パックド・オペランドの両方の要素を操作する命令には、サフィックス“PD”(“packed double precision”)が付く。パックド・オペランドの下位の要素だけを操作する命令には、サフィックス“SD”(“scalar double precision”)が付く。パックド命令がメモリからオペランドを読み出す場合は、暗黙的に 128 ビットの読み出しを実行する。スカラ命令がメモリからオペランドを読み出す場合は、暗黙的に 64 ビットの読み出しを実行する。浮動小数点数を操作する SSE2 を以下に示す。

### 1. データ移動命令

- MOVAPD/MOVUPD: move aligned/unaligned packed double precision floating-point; メモリと SIMD 浮動小数点レジスタの間または SIMD レジスタ同士の間で、128 ビットのデータを転送する。浮動小数点例外: なし
- MOVHPD/MOVLDP: move aligned, high/low packed double precision floating-point; メモリと SIMD 浮動小数点レジスタの上位/下位半分の間で、64 ビットのデータを転送する(下位/上位半分は変更されない)。浮動小数点例外: なし
- MOVMSKPD: move mask packed, double precision floating-point to r32; 2 つのパックド・オペランドの最上位ビットを、32 ビット IA-32 整数レジスタ r32 に転送する。浮動小数点例外: なし
- MOVSD: move scalar double precision floating-point; メモリまたは SIMD レジスタの下位 64 ビットと SIMD レジスタの下位 64 ビットの間で、64 ビットのデータを転送する。浮動小数点例外: なし

### 2. 算術命令

- ADDPD/ADDSD/SUBPD/SUBSD/MULPD/MULSD: add/subtract/multiply packed/scalar, double precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。浮動小数点例外: I, D, O, U, P
- DIVPD/DIVSD: divide packed/scalar, double precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。浮動小数点例外: I, Z, D, O, U, P
- SQRTPD/SQRTSD: square root packed/scalar, double precision floating-point; ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。デスティネーションは SIMD レジスタである。浮動小数点例外: I, D, P

### 3. 最大命令と最小命令

- MAXPD/MAXSD/MINPD/MINSD: maximum/minimum packed/scalar, double precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。浮動小数点例外: I, D(無効操作例外は、任意の NaN オペランドによって発生する)

### 4. 比較命令

- CMPPD/CMPSD: compare packed/scalar, double precision floating-point; 第 1 ソース・オペランドとデスティネーションは SIMD レジスタである。第 2 ソース・オペランドは SIMD レジスタまたはメモリ・ロケーションである。結果は 1(真)または 0(偽)の 64 ビット・マスクである。浮動小数点例外: I, D(無効操作例外は、lt, le, nlt, nle の比較では、任意の NaN オペランドによって発生する。それ以外の比較では、SNaN でのみ発生する)
- COMISD/UCOMISD: compare scalar double precision floating-point ordered/unordered and set EFLAGS; 第 1 ソース・オペランドとデスティネーションは、SIMD レジスタの下位半分である。第 2 ソース・オペランドは、SIMD レジスタの下位半分またはメモリ・ロケーションである。比較の結果は、EFLAGS レジスタの ZF、PF、および CF ビットに書き込まれる。浮動小数点例外: I, D(無効操作例外は、COMISD では任意の NaN オペランドによって発生する。UCOMISD では SNaN オペランドで発生する)

### 5. 変換命令

- CVTPD2PI: MXCSR で指定された丸めモードを使用して、SIMD レジスタまたはメモリ内のパックド倍精度浮動小数点数を、MMX テクノロジ・レジスタ内のパックド 32 ビット符号付き整数に変換する。
- CVTSD2SI: MXCSR で指定された丸めモードを使用して、SIMD レジスタの下位半分またはメモリ内の 1 つのパックド倍精度浮動小数点数を、32 ビット IA-32 整数レジスタ内の 32 ビット符号付き整数に変換する。
- CVTTPD2PI: ゼロ方向への丸めを使用して、SIMD レジスタまたはメモリ内のパックド倍精度浮動小数点数を、MMX テクノロジ・レジスタ内のパックド 32 ビット符号付き整数に変換する。
- CVTTSD2SI: ゼロ方向への丸めを使用して、SIMD レジスタの下位半分またはメモリ内の 1 つのパックド倍精度浮動小数点数を、32 ビット IA-32 整数レジスタ内の 32 ビット符号付き整数に変換する。
- CVTPI2PD: MMX テクノロジ・レジスタまたはメモリ内の下位 2 つの 32 ビット符号付き整数を、SIMD レジスタ内の 2 つの倍精度浮動小数点数に変換する。
- CVTSD2SD: 32 ビット IA-32 整数レジスタまたはメモリ内の 1 つの 32 ビット符号付き整数を、SIMD レジスタの下位半分の倍精度浮動小数点数に変換する。
- CVTPD2DQ/CVTTPD2DQ: SIMD レジスタまたはメモリ内の 2 つの倍精度浮動小数点数を、SIMD レジスタの下位半分の 2 つの 32 ビット整数に変換する。CVTPD2DQ は

MXCSR で指定された丸めモードを使用し、CVTTPD2DQ はゼロ方向への丸めを使用する。

- CVTDQ2PD: SIMD レジスタの下位半分またはメモリ内の 2 つの 32 ビット符号付き整数を、SIMD レジスタ内の 2 つのパックド倍精度浮動小数点数に変換する。
- CVTPS2PD: SIMD レジスタの下位半分またはメモリ内の 2 つの単精度浮動小数点数を、SIMD レジスタ内の 2 つの倍精度浮動小数点数に変換する。
- CVTSS2SD: SIMD レジスタの最下位の位置またはメモリ内の単精度浮動小数点数を、SIMD レジスタの下位半分の倍精度浮動小数点数に変換する。
- CVTPD2PS: SIMD レジスタまたはメモリ内の 2 つの倍精度浮動小数点数を、SIMD レジスタの下位半分の 2 つの単精度浮動小数点数に変換する。
- CVTSD2SS: SIMD レジスタの下位半分またはメモリ内の倍精度浮動小数点数を、SIMD レジスタの最下位の位置の単精度浮動小数点数に変換する。
- CVTPS2DQ/CVTTPS2DQ: SIMD レジスタまたはメモリ内の 4 つの単精度浮動小数点数を、SIMD レジスタ内の 4 つの 32 ビット符号付き整数に変換する。CVTPS2DQ は MXCSR で指定された丸めモードを使用し、CVTTPS2DQ はゼロ方向への丸めを使用する。
- CVTDQ2PS: SIMD レジスタまたはメモリ内の 4 つの 32 ビット符号付き整数を、SIMD レジスタ内の 4 つの単精度浮動小数点数に変換する。

#### 6. 論理命令(ビット単位)

- ANDPD/ANDNPD/ORPD/XORPD: packed logical AND, AND-NOT, OR, XOR; 浮動小数点例外: なし

#### 7. SSE2 のステート管理命令: これらの命令は、SSE について定義されたステート管理命令 (FXSAVE, FXRSTOR, STMXCSR, LDMXCSR) と同じである。

その他の SSE2 には、整数データを操作する命令や、SIMD レジスタ同士の間または SIMD レジスタとメモリ・ロケーションの間で(デスティネーションは常に SIMD レジスタである)、倍精度浮動小数点数に対応する 64 ビット・フィールドをシャッフルする命令がある。最後に、キャッシュ制御命令は、SSE について定義されたキャッシュ制御命令と同じである。

## 4.7 SSE2 を使用したアプリケーションの開発

SSE2 は、すべての IA-32 実行モード(プロテクト・モード、実アドレス・モード、および仮想 8086 モード)から利用できる。アプリケーションは、SSE2 を使用する前に、プロセッサとオペレーティング・システムが SSE2 をサポートしているのを確認しなければならない。

- エミュレーションが無効にされている: CR0.EM(ビット 2) = 0
- プロセッサが SSE2 をサポートしている: CPUID.WNI=1
- プロセッサが FXSAVE/FXRSTOR をサポートしている: CPUID.FXSR(EDX ビット 24)=1

- OS がコンテキスト・スイッチ時の SIMD FP ステートをサポートしている:  
CR4.OSFXSR(ビット 9)=1

さらに、追加された非浮動小数点 SIMD 命令のサポートの有無を確認するためのチェックが必要である(詳細は[2]を参照)。また、SIMD 浮動小数点例外をアンマスクする場合は、アプリケーションは、オペレーティング・システムがアンマスクされた SSE2 例外をサポートしているのを確認しなければならない。

OS がアンマスクされた SIMD 例外をサポートしている: CR4.OSXMMEXCPT(ビット 10)=1

## 4.8 例

### 例 13: SSE2 を使用した $1.0 / (\text{sqrt}(a) - 1.0)$ の例

この例は、例 12 の SSE を使用した簡単な計算  $1.0 / (\text{sqrt}(a) - 1.0)$  の倍精度版である。a の 2 つの値のうち 1 つは、1.0 に非常に近い値である ( $a = 1.0...010...0 \cdot 2^0 = 1 + 2^{-23}$ )。正確な結果は次のようになる。

$$R = (1 + 2^{-25} - 2^{-50} + 2^{-74} - 2^{-97} + \dots) \cdot 2^{24}$$

正確な結果を使用して、結果の相対誤差を求める。倍精度オペランド  $a = 1.0...010...0$  は、計算の始めに XMM1 の下位半分に格納される。

```
#include <stdio.h>
void main () {
    char *mem;
    unsigned int *uimem;
    mem = (char *)(((int)malloc (144) + 16) & ~0x0f); // 16-byte aligned
    // printf ("mem = %x\n\n", (int)mem);
    uimem = (unsigned int *)mem;
    // load x[i] in XMM1, i = 0,1
    uimem[1] = 0x40000000; uimem[0] = 0x00000000;
    // 2.0 (in uimem[1], uimem[0])
    uimem[3] = 0x3ff00000; uimem[2] = 0x20000000;
    // 1.0 + 2^-23 (in uimem[3], uimem[2])
    __asm {
        mov eax, DWORD PTR mem;
        movaps XMM1, [eax];
    }
    // load y[i] = 1.0 in XMM2, i = 0,1
    uimem[1] = 0x3ff00000; uimem[0] = 0x00000000; // 1.0
    uimem[3] = 0x3ff00000; uimem[2] = 0x00000000; // 1.0
    __asm {
        mov eax, DWORD PTR mem;
        movaps XMM2, [eax];
    }
    // calculate 1.0 / (sqrt (x[i]) - 1.0), i = 0,1
    __asm {
        // calculate sqrt (x[i]) in XMM1, i = 0,1
        sqrtpd XMM1, XMM1;
        // calculate sqrt (x[i]) - 1.0 in XMM1, i = 0,1
        subpd XMM1, XMM2;
        // calculate 1.0 / (sqrt (x[i]) - 1.0) in XMM2, i = 0,1
        divpd XMM2, XMM1;
    }
}
```

```

    // store result in memory
    mov eax, DWORD PTR mem;
    movaps [eax], XMM2;
}
printf ("res = %8.8x%8.8x %8.8x%8.8x = %f %f\n",
        uimem[1], uimem[0], uimem[3], uimem[2],
        *(double *)&uimem[0], *(double *)&uimem[2]);
}

```

出力は次のようになる。

```
res = 4003504f333f9de5 4170000008000004 = 2.414214 16777216.500000
```

この出力は、(uimem[3]、uimem[2]の)a =  $1 + 2^{-23}$  については、計算された R の値は  $R^* = (1 + 2^{-25} + 2^{-50}) \cdot 2^{24}$  になることを示している。相対誤差は次のようになる。

$$\varepsilon = |(R - R^*) / R| = (2^{-49} - 2^{-74} + 2^{-97} - \dots) / (1 + 2^{-25} - 2^{-50} + 2^{-74} - 2^{-97} + \dots) \approx 2^{-49}$$

この相対誤差は、例 12 に示した同じ計算の単精度版よりはるかに小さい(例 12 では、計算された結果はプラス無限大であった)。同様に、拡張倍精度計算(本書では説明しない)では、結果の相対誤差は倍精度計算に対して約 1.6 倍向上する( $\varepsilon \approx 5 \cdot 2^{-52}$ )。

## 5 相違点のまとめ

表 4 に、IA-32 FPU 命令、SSE、および SSE2 を使用して浮動小数点計算を実行する場合の主要な相違点を示す。

表 4: IA-32 FPU 命令、SSE、および SSE2 を使用した計算の相違点

FPU	SSE	SSE2
FPU 命令は、いつでも使用できる。	SSE は、プロセッサと OS のサポートのチェック後にのみ使用できる。	SSE2 は、プロセッサと OS のサポートのチェック後にのみ使用できる。
FPU 例外は、いつでもアンマスクできる。	プロセッサと OS が SSE 例外をサポートしている場合、浮動小数点例外は、OS のサポートのチェック後にのみアンマスクできる。	プロセッサと OS が SSE2 例外をサポートしている場合、浮動小数点例外は、OS のサポートのチェック後にのみアンマスクできる。
スカラ命令	4 ウェイ SIMD 命令	2 ウェイ SIMD 命令
浮動小数点形式: 単精度、倍精度、IA-32 スタック単精度、IA-32 スタック倍精度、および拡張倍精度	浮動小数点形式: 単精度	浮動小数点形式: 倍精度
サポートされないオペランドがある(無効操作浮動小数点例外を発生させる)。	サポートされないオペランドはない。	サポートされないオペランドはない。
8 つの 80 ビット・スタック・レジスタ・セット	8 つの 128 ビット・リニア・レジスタ・セット(SSE2 と同じ)	8 つの 128 ビット・リニア・レジスタ・セット(SSE と同じ)
アライメントの合ったメモリ・アクセスとアライメントの合っていないメモリ・アクセスの両方に、独自のロード/ストア命令を使用。	アライメントの合ったメモリ・アクセスとアライメントの合っていないメモリ・アクセスに、別々のロード/ストア命令を使用。	アライメントの合ったメモリ・アクセスとアライメントの合っていないメモリ・アクセスに、別々のロード/ストア命令を使用。
スタック・オーバーフローまたはアンダーフロー例外が発生する。	スタック・モデルはない。	スタック・モデルはない。
別々の FPU 制御ワードとステータス・ワード	統合型制御/ステータス・ワード MXCSR(SSE2 と同じ)	統合型制御/ステータス・ワード MXCSR(SSE と同じ)

続く

表 4: IA-32 FPU 命令、SSE、および SSE2 を使用した計算の相違点(続き)

FPU	SSE	SSE2
ゼロ・フラッシュ・モードはない。	ゼロ・フラッシュ・モードを使用できる。	ゼロ・フラッシュ・モードを使用できる。
ステータス・フラグは、フラグをセットした操作を個別に特定する。	ステータス・フラグは、フラグをセットした操作を個別に特定するとは限らない。フラグの値は、最大4つのサブオペレーションの結果の論理和(OR)になる。	ステータス・フラグは、フラグをセットした操作を個別に特定するとは限らない。フラグの値は、最大2つのサブオペレーションの結果の論理和(OR)になる。
割り込みベクタ 2 またはベクタ 16 を使用して、ソフトウェア浮動小数点例外ハンドラを起動する。	割り込みベクタ 19 を使用して、ソフトウェア浮動小数点例外ハンドラを起動する。	割り込みベクタ 19 を使用して、ソフトウェア浮動小数点例外ハンドラを起動する。
浮動小数点ロード/ストアによって、浮動小数点例外が報告される場合がある。	浮動小数点ロード/ストアでは、浮動小数点例外は報告されない。	浮動小数点ロード/ストアでは、浮動小数点例外は報告されない。
マスクされていない浮動小数点例外は遅延し、次の「待機型」浮動小数点命令で報告される。	マスクされていない浮動小数点例外は、その例外の原因となった命令でただちに報告される。	マスクされていない浮動小数点例外は、その例外の原因となった命令でただちに報告される。
マスクされていないフォルト(I、D、Z)の場合、入力オペランドが例外ハンドラに渡される。	マスクされていないフォルト(I、D、Z)の場合、入力オペランドが例外ハンドラに渡される。パックド・オペレーションのうち例外を発生させなかったサブオペレーションを処理するために、エミュレーションを実行する必要がある。	マスクされていないフォルト(I、D、Z)の場合、入力オペランドが例外ハンドラに渡される。パックド・オペレーションのうち例外を発生させなかったサブオペレーションを処理するために、エミュレーションを実行する必要がある。
マスクされていないトラップ(O、U、P)の場合、結果(スケールされている可能性がある)が例外ハンドラに渡される。	マスクされていないトラップ(O、U、P)の場合、入力オペランドだけが例外ハンドラに渡される。各サブオペレーションの結果を計算するために、ソフトウェア・エミュレーションが必要である。	マスクされていないトラップ(O、U、P)の場合、入力オペランドだけが例外ハンドラに渡される。各サブオペレーションの結果を計算するために、ソフトウェア・エミュレーションが必要である。

続く

表 4: IA-32 FPU 命令、SSE、および SSE2 を使用した計算の相違点(続き)

FPU	SSE	SSE2
ステータス・フラグをセットした後で、それに対応する浮動小数点例外をアンマスクすると、例外が発生する。	ステータス・フラグをセットした後で、それに対応する浮動小数点例外をアンマスクしても、例外は発生しない。	ステータス・フラグをセットした後で、それに対応する浮動小数点例外をアンマスクしても、例外は発生しない。
FPU 命令セットに超越関数が含まれている。	超越関数はない。	超越関数はない。
ハードウェアは、IEEE 規格 754-1985 の必要条件に 100% 適合している。また、高精度のオペランドから低精度の結果を生成できる(高精度のメリットが向上する)点を除いて、すべての推奨事項に適合している。	ハードウェアは、単精度計算に関する IEEE 規格 754-1985 の必要条件に 100% 適合している。マスクされていない浮動小数点例外の処理(IEEE 754 の標準推奨事項)には、ソフトウェア・エミュレーションが必要である。	ハードウェアは、倍精度計算に関する IEEE 規格 754-1985 の必要条件に 100% 適合している。マスクされていない浮動小数点例外の処理(IEEE 754 の標準推奨事項)には、ソフトウェア・エミュレーションが必要である。
FPU 命令は、(場合によっては)SSE および SSE2 とは異なる方法で NaN 入力を処理する。	SSE と SSE2 は、同じ方法で NaN 入力を処理するが、(場合によっては)FPU 命令とは処理方法が異なる。	SSE2 と SSE は、同じ方法で NaN 入力を処理するが、(場合によっては)FPU 命令とは処理方法が異なる。

最後の例は、FPU、SSE、または SSE2 を使用して浮動小数点計算を実行する場合の予想される精度の違いを示している。

#### 例 14: FPU、SSE、および SSE2 を使用した計算の精度の比較

次の簡単な式を計算する。この計算は、(任意の浮動小数点形式で表現可能な)正確な結果が得られる。

$$(((1 / ((1 / 10) / (1 / 3))) + 3 / 10) / 11) * (1 / (1 / 99) + 11)) * 39 = 1417$$

最初に SSE を使用して左辺の式を計算する(パックド・オペランドが使用されるため、計算は実際には 4 回実行される)。

```
#include <stdio.h>
void main () {
    float res[4], *pres = res,
          a1[4] = {1.0, 1.0, 1.0, 1.0}, *pa1 = a1,
          a3[4] = {3.0, 3.0, 3.0, 3.0}, *pa3 = a3,
          a10[4] = {10.0, 10.0, 10.0, 10.0}, *pa10 = a10,
          a11[4] = {11.0, 11.0, 11.0, 11.0}, *pa11 = a11,
          a39[4] = {39.0, 39.0, 39.0, 39.0}, *pa39 = a39,
          a99[4] = {99.0, 99.0, 99.0, 99.0}, *pa99 = a99;
    __asm {
        mov eax, DWORD PTR pa1
        movups XMM5, [eax] // 1 in xmm5
    }
}
```

```

movaps XMM1, XMM5 // 1 in xmm1
mov eax, DWORD PTR pa10
movups XMM2, [eax] // 10 in xmm2
divps XMM1, XMM2 // 1/10 in xmm1
movaps XMM2, XMM5 // 1 in xmm2
mov eax, DWORD PTR pa3
movups XMM3, [eax] // 3 in xmm3
divps XMM2, XMM3 // 1/3 in xmm2
divps XMM1, XMM2 // 3/10 in xmm1
movaps XMM2, XMM5 // 1 in xmm2
divps XMM2, XMM1 // 10/3 in xmm2
mov eax, DWORD PTR pa10
movups XMM1, [eax] // 10 in xmm1
divps XMM3, XMM1 // 3/10 in xmm3
addps XMM2, XMM3 // 109/30 in xmm2
mov eax, DWORD PTR pa11
movups XMM1, [eax] // 11 in xmm1
divps XMM2, XMM1 // 109/330 in xmm2
mov eax, DWORD PTR pa99
movups XMM3, [eax] // 99 in xmm3
movups XMM4, XMM5 // 1 in xmm4
divps XMM4, XMM3 // 1/99 in xmm4
divps XMM5, XMM4 // 99 in xmm5
addps XMM1, XMM5 // 110 in xmm1
mulps XMM1, XMM2 // 109/3 in xmm1
mov eax, DWORD PTR pa39
movups XMM2, [eax] // 39 in xmm2
mulps XMM1, XMM2 // 1417 in xmm1
mov eax, DWORD PTR pres;
movups [eax], XMM1;
}
printf ("res = \n\t%8.8x %8.8x %8.8x %8.8x = \n\t%f %f %f %f\n",
        *(unsigned int *)&res[0], *(unsigned int *)&res[1],
        *(unsigned int *)&res[2], *(unsigned int *)&res[3],
        res[0], res[1], res[2], res[3]);
}

```

出力は、純粋な IEEE 単精度計算に対応する。

```

res = 44b12001 44b12001 44b12001 44b12001 =
      1417.000122 1417.000122 1417.000122 1417.000122

```

この出力は、結果が 1417 より 1ulp だけ大きいことを示している。

$$\text{res} = 1417 + 1 \text{ ulp} = 1417 + 2^{10-23} = 1417 + 2^{-13}$$

絶対誤差  $e = +2^{-13}$  に対応する相対誤差は、次のとおりである。

$$\varepsilon_1 = 8.6147 \cdot 10^{-8}$$

FPU 命令を使用して、制御ワード内で精度を 24 ビットに設定して、FPU スタック上でこの計算を実行した場合は、これと同じ結果が得られる。この簡単な計算では、オーバーフロー状態やアンダーフロー状態は発生しないため、純粋な IEEE 単精度モデルをエミュレートするために、中間結果を一度メモリにストアして再びロードする必要はない。

次に、SSE2 を使用して、(1417.0 の結果が得られるはずの)同じ式を計算する(パックド・オペランドが使用されるため、計算は 2 回実行される)。

```

#include <stdio.h>
void main () {
    double res[2], *pres = res,
        a1[2] = {1.0, 1.0}, *pa1 = a1,
        a3[2] = {3.0, 3.0}, *pa3 = a3,
        a10[2] = {10.0, 10.0}, *pa10 = a10,
        a11[2] = {11.0, 11.0}, *pa11 = a11,
        a39[2] = {39.0, 39.0}, *pa39 = a39,
        a99[2] = {99.0, 99.0}, *pa99 = a99;
    unsigned int *uint;
    uint = (unsigned int *)res;
    __asm {
        mov eax, DWORD PTR pa1
        movupd XMM5, [eax] // 1 in xmm5
        movapd XMM1, XMM5 // 1 in xmm1
        mov eax, DWORD PTR pa10
        movupd XMM2, [eax] // 10 in xmm2
        divpd XMM1, XMM2 // 1/10 in xmm1
        movapd XMM2, XMM5 // 1 in xmm2
        mov eax, DWORD PTR pa3
        movupd XMM3, [eax] // 3 in xmm3
        divpd XMM2, XMM3 // 1/3 in xmm2
        divpd XMM1, XMM2 // 3/10 in xmm1
        movapd XMM2, XMM5 // 1 in xmm2
        divpd XMM2, XMM1 // 10/3 in xmm2
        mov eax, DWORD PTR pa10
        movupd XMM1, [eax] // 10 in xmm1
        divpd XMM3, XMM1 // 3/10 in xmm3
        addpd XMM2, XMM3 // 109/30 in xmm2
        mov eax, DWORD PTR pa11
        movupd XMM1, [eax] // 11 in xmm1
        divpd XMM2, XMM1 // 109/330 in xmm2
        mov eax, DWORD PTR pa99
        movupd XMM3, [eax] // 99 in xmm3
        movupd XMM4, XMM5 // 1 in xmm4
        divpd XMM4, XMM3 // 1/99 in xmm4
        divpd XMM5, XMM4 // 99 in xmm5
        addpd XMM1, XMM5 // 110 in xmm1
        mulpd XMM1, XMM2 // 109/3 in xmm1
        mov eax, DWORD PTR pa39
        movupd XMM2, [eax] // 39 in xmm2
        mulpd XMM1, XMM2 // 1417 in xmm1
        mov eax, DWORD PTR pres;
        movupd [eax], XMM1;
    }
    printf ("res = \n\t%8.8x%8.8x %8.8x%8.8x = \n\t%f %f\n",
        uint[3], uint[2], uint[1], uint[0], res[1], res[0]);
}

```

出力は、純粋な IEEE 倍精度計算に対応する。

```
res = 409623fffffffffe 409623fffffffffe = 1417.000000 1417.000000
```

この出力は、結果が 1417 より 2ulp 小さいことを示している。

$$\text{res} = 1417 - 2 \text{ulp} = 1417 - 2 \cdot 2^{10-52} = 1417 - 2^{-41}$$

絶対誤差  $e = -2^{-41}$  に対応する相対誤差は、次のとおりである。

$$\varepsilon_2 = 3.2092 \cdot 10^{-16}$$

この相対誤差は、単精度の場合の相対誤差( $\varepsilon_1 = 8.6147 \cdot 10^{-8}$ )よりはるかに小さい。

FPU 命令を使用して、制御ワード内で精度を 53 ビットに設定して、FPU スタック上でこの計算を実行した場合は、これと同じ結果が得られる。この簡単な計算では、オーバーフロー状態やアンダーフロー状態は発生しないため、純粋な IEEE 倍精度モデルをエミュレートするために、中間結果を一度メモリにストアして再びロードする必要はない。

最後に、FPU 命令を使用して同じ式を計算する。中間計算には、拡張倍精度形式を使用する (FPU 制御ワード内で PC=11 に設定する)。

```
#include <stdio.h>
void
main () {
    float a3 = 3., a10 = 10., a11 = 11., a39 = 39., a99 = 99.;
    char *pa3, *pa10, *pa11, *pa39, *pa99;
        // pointers to single precision numbers
    unsigned short t[5], *pt; // 10-byte (80-bit) result
    unsigned short cw, *pcw; // control word and pointer to it
    float res; // result, used just to print the decimal value
    char *pres;
    pa3 = (char *)&a3;
    pa10 = (char *)&a10;
    pa11 = (char *)&a11;
    pa39 = (char *)&a39;
    pa99 = (char *)&a99;
    pt = t;
    pres = (char *)&res;
    pcw = &cw;
    // set control word
    cw = 0x033f; // round to nearest, 64 bits, exceptions disabled
        // (double-extended precision)
    // cw = 0x023f; // (use for pure IEEE double precision)
        // round to nearest, 53 bits, exceptions disabled
    // cw = 0x003f; // (use for pure IEEE single precision)
        // round to nearest, 24 bits, exceptions disabled
    __asm {
        mov eax, DWORD PTR pcw
        fldcw [eax]
        // compute E = 1417.0
        fldl // 1 in st(0)
        mov eax, DWORD PTR pa10
        fdiv DWORD PTR [eax] // 1/10 in st(0)
        fldl // 1 in st(0), 1/10 in st(1)
        mov eax, DWORD PTR pa3
        fdiv DWORD PTR [eax] // 1/3 in st(0), 1/10 in st(1)
        fdivp st(1), st(0) // 3/10 in st(0)
        fldl // 1 in st(0), 3/10 in st(1)
        fxch // 3/10 in st(0), 1 in st(1)
```

```

    fdivp st(1), st(0) // 10/3 in st(0)
    mov eax, DWORD PTR pa3
    fld DWORD PTR [eax] // 3 in st(0), 10/3 in st(1)
    mov eax, DWORD PTR pa10
    fdiv DWORD PTR [eax] // 3/10 in st(0), 10/3 in st(1)
    faddp st(1), st(0) // 109/30 in st(0)
    mov eax, DWORD PTR pa11
    fdiv DWORD PTR [eax] // 109/330 in st(0)
    fld1 // 1 in st(0), 109/330 in st(1)
    mov eax, DWORD PTR pa99
    fdiv DWORD PTR [eax] // 1/99 in st(0), 109/330 in st(1)
    fld1 // 1 in st(0), 1/99 in st(1), 109/330 in st(2)
    fxch // 1/99 in st(0), 1 in st(1), 109/330 in st(2)
    fdivp st(1), st(0) // 99 in st(0), 109/330 in st(1)
    mov eax, DWORD PTR pa11
    fadd DWORD PTR [eax] // 110 in st(0), 109/330 in st(1)
    fmulp st(1), st(0) // 109/3 in st(0)
    mov eax, DWORD PTR pa39
    fmul DWORD PTR [eax] // 1417 in st(0)
    mov eax, DWORD PTR pres
    fst DWORD PTR [eax] // res from the FPU stack to memory, pop st(0)
    mov eax, DWORD PTR pt
    fstp TBYTE PTR [eax] // res from the FPU stack to memory, pop st(0)
}
printf ("res = %4.4x%4.4x%4.4x%4.4x%4.4x\n",
        t[4], t[3], t[2], t[1], t[0]); // t = 1417.0
printf ("res = %6.6f\n", res);
}

```

出力は、純粋な IEEE 拡張倍精度計算に対応する。

res = 4009b1200000000000001

res = 1417.000000

この出力は、結果が 1417 より 1ulp 大きいことを示している。

$$\text{res} = 1417 + 1 \text{ ulp} = 1417 + 2^{10-63} = 1417 - 2^{-53}$$

絶対誤差  $e = -2^{-53}$  に対応する相対誤差は、次のとおりである。

$$\varepsilon_3 = 1.0842 \cdot 10^{-19}$$

したがって、単精度、倍精度、および拡張倍精度計算の相対誤差は、それぞれ次のようになる。

$$\varepsilon_1 = 8.6147 \cdot 10^{-8} > \varepsilon_2 = 3.2092 \cdot 10^{-16} > \varepsilon_3 = 1.0842 \cdot 10^{-19}$$

数千以上の計算ステップを実行した場合、3種類の相対誤差の比は維持されるが、各相対誤差の絶対値ははだいに大きくなる。アルゴリズムが詳しくわかれば、テストまたは理論的な数値分析によって、最も良い計算モデルを選択できる。

## 6 結論

FPU 命令は、さまざまな精度、三角関数、対数関数、または指数関数の呼び出し、整数計算、BCD 計算(財務会計アプリケーションに効果的)、および浮動小数点計算を組み合わせた計算など、複雑な浮動小数点計算のための多くの操作を可能にする。しかし、多数のデータ・アイテムに対して同じ浮動小数点計算を実行する場合は、SIMD モデルを使用すれば、パフォーマンスが大きく向上する。このとき、単精度形式の有効範囲で十分な場合は、SSE が最適である。倍精度形式の有効範囲が必要な場合は、SSE2 の使用を推奨する。

IA-32 FPU アーキテクチャ、SSE、および SSE2 は、いずれも浮動小数点計算に関する IEEE の必要条件に適合する。ただし、IEEE 規格の推奨事項に適合するには、ソフトウェアによる拡張が必要である(たとえば、ユーザのソフトウェア浮動小数点例外ハンドラを起動する前に、例外を発生させた SSE または SSE2 のソフトウェア・エミュレーションを実行する必要がある)。IA-32 浮動小数点アーキテクチャには、IEEE 規格のすべての推奨事項をサポートする以外に多数の機能が追加され、その一部は事実上の業界標準になっている。たとえば、ゼロ・フラッシュ・モードによって、精度が多少低下する代わりにパフォーマンスを改善できる。また、もう 1 つの例として、低精度のオペランドから高精度の結果を生成できる。全体として、インテル・アーキテクチャは、浮動小数点計算に関して、非常にすぐれた精度、パフォーマンス、および柔軟性を実現している。