intel®

IA-32 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：
基本アーキテクチャ

注記：
『IA-32 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』は、次の 4 巻から構成されています。

上巻：基本アーキテクチャ　（資料番号 253665-013J）
中巻 A：命令セット・リファレンス A-M　（資料番号 253666-013J）
中巻 B：命令セット・リファレンス N-Z　（資料番号 253667-013J）
下巻：システム・プログラミング・ガイド（資料番号 253668-013J）

設計する際は、これら 4 巻すべてを参照してください。

2004 年

【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コピーすることは禁じられています。

IA-32 アーキテクチャ・プロセッサ（インテル® Pentium® 4 プロセッサ、インテル® Pentium® III プロセッサなど）、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

ハイパー・スレッディング・テクノロジを利用するには、ハイパー・スレッディング・テクノロジに対応したインテル Pentium 4 プロセッサを搭載したコンピュータ・システム、および同技術に対応したチップセットと BIOS、OS が必要です。性能は、使用するハードウェアやソフトウェアによって異なります。HT テクノロジに対応したプロセッサの情報等、詳細については http://www.intel.co.jp/jp/info/hyperthreading/ を参照してください。

目 次

図目次

表目次

1

本書について

第 1 章
本書について

1

『IA-32 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 : 基本アーキテクチャ』（資料番号 253665-013J）は、IA-32 インテル® プロセッサ全般のアーキテクチャとプログラミング環境を説明している全巻のうちの 1 巻である。他の巻を次に示す。

- 『IA-32 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、中巻 A、B : 命令セット・リファレンス・マニュアル』（資料番号 253666-013J、253667-013J）
- 『IA-32 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、下巻 : システム・プログラミング・ガイド』（資料番号 253668-013J）

『IA-32 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』の「上巻 : 基本アーキテクチャ」は、IA-32 プロセッサの基本的なアーキテクチャとプログラミング環境について説明している。「中巻 A、B : 命令セット・リファレンス・マニュアル」は、プロセッサの命令セットとオペコードの構造について説明している。上巻と中巻は、既存のオペレーティング・システムやエグゼクティブの下で実行するプログラムを開発しているアプリケーション・プログラマを対象としている。「下巻 : システム・プログラミング・ガイド」は、IA-32 プロセッサのオペレーティング・システム・サポート環境と IA-32 プロセッサの互換性に関する情報について説明している。下巻が対象とするのは、オペレーティング・システムや BIOS の開発者である。

1.1. 本書の対象となる IA-32 プロセッサ

本書には、主に最近の IA-32 プロセッサに関する情報が記載されている。これには、インテル® Pentium® プロセッサ、P6 ファミリ・プロセッサ、インテル® Pentium® 4 プロセッサ、インテル® Pentium® M プロセッサ、インテル® Xeon™ プロセッサが含まれる。P6 ファミリ・プロセッサとは、P6 ファミリ・マイクロアーキテクチャに基づく IA-32 プロセッサである。P6 ファミリには、インテル® Pentium® Pro プロセッサ、インテル® Pentium® II プロセッサ、インテル® Pentium® III プロセッサが含まれる。インテル Pentium 4 プロセッサとインテル Xeon プロセッサは、Intel NetBurst® マイクロアーキテクチャに基づいている。

1.2. 『IA-32 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：基本アーキテクチャ』の概要

本書は、次の内容で構成されている。

第 1 章 － 本書について。『IA-32 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』の全 4 巻それぞれの内容を簡単に説明する。また、これらのマニュアルで使用されている表記法について説明すると共に、インテルのマニュアルやドキュメンテーションのなかでプログラマやハードウェア設計者に関係する関連資料を併記している。

第 2 章 － IA-32 インテル® アーキテクチャの概説。IA-32 アーキテクチャと、このアーキテクチャを基礎とするインテル® プロセッサのファミリについて概説する。また、これらのプロセッサに見られる共通の特徴や、IA-32 アーキテクチャの変遷について簡単に説明する。

第 3 章 － IA-32 基本実行環境。メモリ構成のモデルを概説すると共に、アプリケーション上で使用するレジスタセットについて説明する。

第 4 章 － データ型。プロセッサが認識するデータ型とアドレス指定モードについて説明する。実数、浮動小数点形式、浮動小数点例外の概要も示す。

第 5 章 － 命令セットの要約。すべての IA-32 アーキテクチャ命令の一覧を、テクノロジ・グループごとに分けて示す。各グループの命令は、機能的に関連のあるグループごとに記載されている。

第 6 章 － プロシージャ・コール、割り込み、例外。プロシージャ・スタックと、プロシージャ・コールの実行のメカニズム、割り込みと例外処理のメカニズムについて説明する。

第 7 章 － 汎用命令によるプログラミング。汎用レジスタおよびセグメント・レジスタ上で基本データ型を操作する、基本的なロード命令とストア命令、プログラム制御命令、算術命令、ストリング命令について説明する。プロテクト・モードで実行されるシステム命令についても説明する。

第 8 章 － x87 FPU によるプログラミング。x87 の浮動小数点ユニット（FPU）について説明し、浮動小数点レジスタとデータ型、浮動小数点命令セット、プロセッサの浮動小数点例外条件について説明する。

第 9 章 － インテル® MMX® テクノロジ・レジスタによるプログラミング。インテル MMX テクノロジについて説明する。これには、MMX テクノロジ・レジスタとデータ型、MMX 命令セットの概要についての説明が含まれる。

第10章 － ストリーミングSIMD拡張命令（SSE）によるプログラミング。SSEについて説明する。これには、XMMレジスタ、MXCSRレジスタ、パックド単精度浮動小数点データ型についての説明が含まれる。また、SSE命令セットの概要と、SSEにアクセスするコードを作成する際のガイドラインについても説明する。

第11章 － ストリーミングSIMD拡張命令2（SSE2）によるプログラミング。SSE2について説明する。これには、XMMレジスタ、パックド倍精度浮動小数点データ型についての説明が含まれる。また、SSE2の命令セットの概要と、SSE2にアクセスするコードを作成する際のガイドラインについても説明する。この章では、SSEとSSE2によって生成されるSIMD浮動小数点例外についても説明する。また、オペレーティング・システムとアプリケーション・コードにSSEとSSE2のサポート機能を組み込むための一般的なガイドラインを示す。

第12章 － ストリーミングSIMD拡張命令3（SSE3）によるプログラミング。SSE3について説明する。これには、SSE3の命令セットの概要と、SSE3にアクセスするコードを作成する際のガイドラインが含まれる。

第13章 － 入出力。I/Oポートのアドレス指定、I/O命令、I/O保護メカニズムなど、プロセッサのI/Oアーキテクチャについて説明する。

第14章 － プロセッサの識別と機能の判定。プロセッサが備えているCPUタイプおよび機能を判定する方法について説明する。

付録A － EFLAGSクロス・リファレンス。IA-32の命令がEFLAGSレジスタの各フラグに及ぼす影響を要約している。

付録B － EFLAGS条件コード。条件付きのジャンプ、移動、条件コード命令でのバイトセットにおいてEFLAGSレジスタの条件コードフラグ（OF、CF、ZF、SF、PF）がどのように使用されるかを説明する。

付録C － 浮動小数点例外の要約。x87 FPU浮動小数点、SSE、SSE2、SSE3浮動小数点命令で発生する例外を一覧で示す。

付録D － x87 FPU例外ハンドラを作成する際のガイドライン。FPU例外に対してMS-DOS* 互換の例外処理機能を設計し開発する方法について説明する。これには、ソフトウェアとハードウェアの要件、アセンブリ言語コードの例が含まれる。また、信頼性の高いFPU例外ハンドラを開発するための一般的な技法について説明する。

付録E － SIMD浮動小数点例外ハンドラを作成する際のガイドライン。SSE、SSE2、SSE3の浮動小数点命令で発生する例外と、これらの例外を処理する例外ハンドラを作成する際のガイドラインについて説明する。

1.3. 表記法

本書では、データ構造フォーマット、命令のシンボリック表現、16進数と2進数に対して特別な表記法を使用している。この表記法を理解しておけば、本書を理解しやすくなる。

1.3.1. ビット・オーダとバイト・オーダ

メモリ内のデータ構造図では、小さい方のアドレスが図の下の方に示され、上に行くほど大きくなる。ビット位置は、右から左に番号が付けられている。セットされたビットの数値は、2をビット位置を表す数で累乗した値に等しくなる。IA-32プロセッサは「リトル・エンディアン」マシンであり、ワードのバイトは最下位バイトから順に番号が付けられている。図1-1.にこれらの規則を示す。

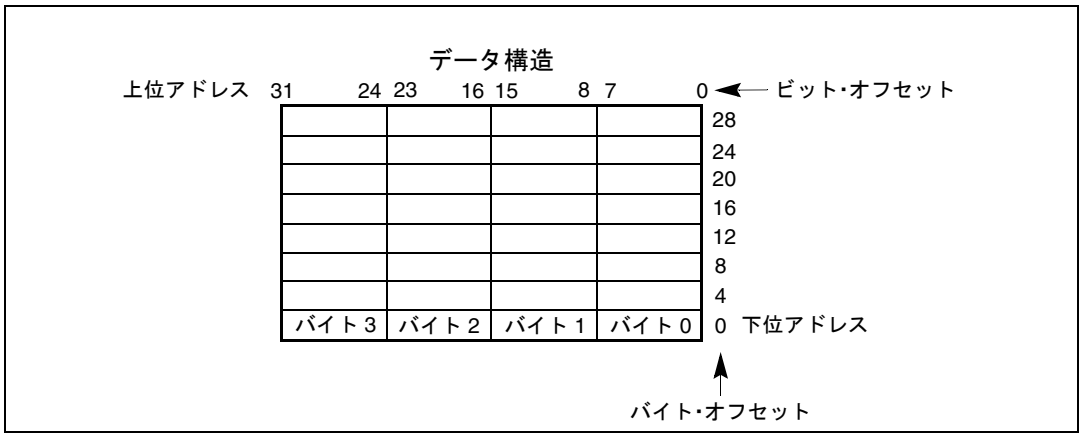


図 1-1. ビット・オーダとバイト・オーダ

1.3.2. 予約ビットとソフトウェア互換性

レジスタやメモリのレイアウトの説明で、特定のビットが「予約済み」と記されているときがある。ビットが予約済みとして記されている場合は、将来のプロセッサとの互換性を維持するため、これらのビットが将来的に何らかの機能を持つものとみなした上で、ソフトウェア上でこれらのビットを取り扱わなければならない。予約ビットの動作は、未定義としてだけではなく、予測不可能とみなさなければならない。予約ビットを処理する場合は、ソフトウェア上で、次に示すガイドラインに従わなければならない。

- 予約ビットを含むレジスタの値をテストするときは、予約ビットのステートに依存してはならない。テストする前に、予約ビットをマスクアウトする。

- メモリまたはレジスタに格納するときは、予約ビットのステートに依存してはならない。
- 予約ビットに書き込まれた情報が保存されるものとみなしてはならない。
- レジスタにロードするときは、マニュアル上で予約ビットに対して値を指定している場合には、その値を予約ビットにロードしなければならない。マニュアルになければ、同じレジスタから前に読まれた値を再ロードする。

注記

ソフトウェアを、IA-32 レジスタの予約ビットのステートに依存させることは絶対に避けること。予約ビットの値に依存すると、プロセッサが予約ビットを処理する方法が決定されていないにもかかわらず、その未決定の方法にソフトウェアが依存することになる。予約ビットの値に依存したプログラムを作成すると、将来のプロセッサとの互換性を損なう危険がある。

1.3.3. 命令オペランド

命令をシンボルで表現する場合は、IA-32 のアセンブリ言語のサブセットを使用する。このサブセットでは、命令は次の形式をとる。

```
label: mnemonic argument1, argument2, argument3
```

上記の形式では

- **label** は識別子で、後にコロンが続く。
- **mnemonic** は、同じ機能を持つ命令オペコードの予約名である。
- オペランド **argument1**、**argument2**、**argument3** はオプションである。オペコードに応じて、0 ～ 3 つのオペランドを使用する。オペランドを使用する場合、オペランドはリテラルかデータ項目の識別子のいずれかの形式をとる。オペランド識別子は、レジスタの予約名であるか、またはプログラムの別の箇所（例には示されていないことがある）で宣言されたデータ項目に割り当てられているものとみなされる。

演算命令や論理命令にオペランドが 2 つある場合は、右側のオペランドがソースであり、左側がデスティネーションになる。

例：

```
LOADREG: MOV EAX, SUBTOTAL
```

この例では、LOADREG はラベル、MOV はオペコードのニーモニック識別子、EAX はデスティネーション・オペランド、SUBTOTAL はソース・オペランドになる。アセンブリ言語によっては、ソースとデスティネーションの順序が逆になる場合がある。

1.3.4. 16 進数と 2 進数

16 をベースとする数（16 進数）は、末尾に文字 H を付けた 16 進数字の文字列で表す（例えば、F82EH）。16 進数字は、0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F のいずれかである。

ベースを 2 とする数（2 進数）は、1 と 0 の文字列で表し、場合によって末尾に文字 B を付ける（例えば、1010B）。「B」を付けるのは、数値のタイプに混乱が生じるような場合に限られる。

1.3.5. セグメント化アドレス指定

インテル® アーキテクチャ・プロセッサでは、バイトによるアドレス指定を採用している。つまり、メモリはバイトの連続として構成されアクセスされる。1 バイトをアクセスするのか複数バイトをアクセスするのかにかかわらず、そのバイトを格納しているメモリへのアクセスには、1 つのバイトアドレスを使用する。アドレス指定が可能なメモリの範囲を、**アドレス空間**と呼ぶ。

プロセッサは、セグメント化アドレス指定もサポートしている。これは、プログラムが**セグメント**と呼ばれる多数の独立したアドレス空間を持つ場合のアドレス指定の一形式である。例えば、プログラムはコード（命令）とスタックを別々のセグメントに保持できる。これにより、コードアドレスは常にコード空間を、スタックアドレスは常にスタック空間を参照することが可能になる。セグメント内のバイトアドレスを指定するには、次の表記法を使用する。

Segment-register:Byte-address

例えば、次のセグメント・アドレスは、DS レジスタがポイントするセグメント内のアドレス FF79H にあるバイトを指す。

```
DS:FF79H
```

また、次のセグメント・アドレスは、コード・セグメントの命令アドレスを指す。CS レジスタはコード・セグメントをポイントし、EIP レジスタは命令のアドレスを格納する。

```
CS:EIP
```

1.3.6. 例外

例外とは、命令がエラーを引き起こした場合に一般的に発生するイベントである。例えば、0で除算しようとすると例外が発生する。ただし、ブレークポイントのように、エラー以外の条件で発生する例外もある。例外によっては、エラーコードを提示するものもある。エラーコードによって、エラーに関する追加情報が示される。例外とエラーコードを示すために使用する表記例を次に示す。

```
#PF(fault code)
```

この例が示すのは、フォルトのタイプを指すエラーコードが報告される条件でのページ・フォルト例外である。ある種の条件では、エラーコードが発生する例外でも、正確なコードを報告できない場合がある。このような場合、一般保護例外の例が次に示すように、エラーコードは0になる。

```
#GP(0)
```

1.4. 参考文献

インテル・プロセッサに関連する資料の一覧は、以下のリンクに記載されている。

http://www.intel.co.jp/jp/developer/hardwaredesign/processors.htm（日本語）

http://developer.intel.com/design/processor/（英語）

このWebサイトに記載されている資料には、オンラインで表示できるものと注文できるものがある。入手可能な資料は、まずインテル・プロセッサ別に、次に資料のタイプ（アプリケーション・ノート、データシート、マニュアル、論文、仕様のアップデート）別に記載されている。

以下の資料も参照のこと。

- 特定のインテルIA-32プロセッサのデータシート
- 特定のインテルIA-32プロセッサの仕様のアップデート
- 『AP-485, Intel Processor Identification and the CPUID Instruction』（資料番号241618）
- 『AP-485、インテル® プロセッサの識別とCPUID命令』（資料番号241618J）
- 『IA-32 Intel® Architecture Optimization Reference Manual』（資料番号248966）
- 『IA-32 インテル® アーキテクチャ最適化 リファレンス・マニュアル』（資料番号248966-005J）

1.5. 参考 URL

- http://developer.intel.com/sites/developer/（英語）
- http://www.intel.co.jp/jp/developer/（日本語）

2

IA-32 インテル®
アーキテクチャの概説

第 2 章
IA-32 インテル®
アーキテクチャの概説

2

今日、コンピュータは、処理能力と普及率の飛躍的な向上によって、20 世紀後半のビジネスと社会を形成する最大の力の 1 つになった。技術、ビジネスなどの新しい分野の成長にも、コンピュータが重要な役割を果たしている。

IA-32 インテル® アーキテクチャは、これまで、コンピュータの進歩の最前線を切り開いてきた。今日では、全世界で使用されているコンピュータと総合的な処理能力から判断して、最も普及したコンピュータ・アーキテクチャと見なせる。

2.1. IA-32 アーキテクチャの変遷

本章では、インテル® 8086 プロセッサから最新のインテル® Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサまで、現在の IA-32 アーキテクチャに至る技術的発展の概要を説明する。歴史的データの詳細については、以下のリンクを参照のこと。

http://www.intel.co.jp/jp/personal/museum/index.htm

IA-32 アーキテクチャ・ファミリでは、1978 年にリリースされたプロセッサ向けのオブジェクト・コードが最新のプロセッサ上でも実行可能である。

2.1.1. 16 ビット・プロセッサとセグメンテーション（1978 年）

IA-32 アーキテクチャ・ファミリは、16 ビット・プロセッサである 8086 プロセッサと 8088 プロセッサから始まった。8086 プロセッサは、16 ビット・レジスタと 16 ビット外部データバスを持ち、また 20 ビットのアドレス指定により 1M バイトのアドレス空間を実現した。8088 プロセッサは、外部データバスが 8 ビットに縮小されていることを除けば、8086 プロセッサと同じである。

8086/8088 プロセッサでは、IA-32 アーキテクチャにセグメンテーションが導入された。セグメンテーションにより、16 ビット・セグメント・レジスタに、最大 64K バイトのメモリ・セグメントに対するポインタが追加された。8086/8088 プロセッサは、一度に 4 つのセグメント・レジスタを使用して、セグメント間の切り替えなしで、最大 256K バイトまでのアドレス指定が可能である。セグメント・レジスタ・ポインタと追加の 16 ビット・ポインタで構成される 20 ビット・アドレスによって、合計 1M バイトのアドレス範囲が利用できる。

2.1.2. インテル® 286 プロセッサ（1982 年）

IA-32 アーキテクチャに初めて保護モード操作を導入したのはインテル® 286 プロセッサである。保護モードは、セグメント・レジスタの内容を、ディスクリプタ・テーブルに対するセレクタもしくはポインタとして使用するものである。ディスクリプタは 24 ビットのベースアドレスを提供することで最大 16M バイトの物理メモリサイズを可能にすると共に、セグメントのスワッピングによる仮想メモリ・マネージメントや各種の保護メカニズムをサポートしていた。これらのメカニズムには以下のものが含まれる。

- セグメント・リミット・チェック
- 読み取り専用や実行専用のセグメント・オプション
- 4 つの特権レベル

2.1.3. Intel386™ プロセッサ（1985 年）

Intel386™ プロセッサは、IA-32 アーキテクチャ・ファミリの最初の 32 ビット・プロセッサである。このプロセッサでは、オペランドの保持とアドレス指定用に、32 ビット・レジスタが導入された。それぞれの 32 ビット Intel386 レジスタの下位半分は、上位互換性を得るため、前世代の 16 ビット・レジスタのいずれかのプロパティをそのまま受け継いだものになった。また、仮想 8086 モードにより、8086 プロセッサや 8088 プロセッサ用に開発されたプログラムを実行する際に高い効率をあげることが可能になった。

また、Intel386 プロセッサは以下の機能をサポートしている。

- 最大 4G バイトの物理メモリをサポートする 32 ビット・アドレス・バス
- セグメント・メモリ・モデルおよび「フラット」[1] メモリモデル
- 4K バイトの固定ページサイズによって仮想メモリ管理を実現するページング
- 並列ステージのサポート

2.1.4. Intel486™ プロセッサ（1989 年）

Intel486™ プロセッサは、Intel386™ プロセッサの命令デコードユニットと実行ユニットをパイプライン化された 5 ステージとすることで、さらに並列実行処理機能を改善したものである。各ステージは、異なる実行ステージにある最大 5 つの命令を他のステージと並列に処理する。

1. 任意のアドレス空間へのアクセスは、32 ビット・アドレス・コンポーネント 1 つだけで可能。

また、このプロセッサには以下の機能も追加されている。

- クロック当たりのスカラレートで実行可能な命令の割合を増やす、8K バイトのオンチップ第 1 レベル・キャッシュ
- 統合型 x87 FPU
- エネルギー節約をはじめとするシステム・マネージメント機能

2.1.5. インテル® Pentium® プロセッサ（1993 年）

インテル® Pentium® プロセッサの登場に際して、スーパースカラ性能を実現するため、2 番目の実行パイプラインが追加された（それぞれ u と v と呼ばれる 2 本のパイプラインにより、クロック当たり 2 命令を実行可能）。また、オンチップの第 1 レベル・キャッシュのサイズが倍増され、8K バイトがコードに、さらに 8K バイトがデータに割り当てられた。データ・キャッシュは MESI プロトコルを使用し、ライトバック・キャッシュの効率を改善すると共に、Intel486 プロセッサで採用しているライトスルー・キャッシュの効率も改善している。また、ループ命令における性能を改善するため、オンチップの分岐テーブルを持つ分岐予測が追加されている。

また、このプロセッサには以下の機能も追加されている。

- 仮想 8086 モードの効率を高め、4K バイト・ページと共に 4M バイト・ページを使用可能にする機能拡張
- 内部データ転送速度をアップする 128 ビットと 256 ビットの内部データパス
- 64 ビットに拡張されたバースト可能な外部データバス
- 複数のプロセッサを搭載したシステムをサポートする APIC
- 2 つのプロセッサ・システム間でスムーズな処理を実行する、デュアルプロセッサ・モード

インテル Pentium プロセッサ・ファミリにおける次の段階では、インテル® MMX® テクノロジが導入された（MMX テクノロジ対応インテル Pentium プロセッサ）。
インテル MMX テクノロジは、SIMD（Single Instruction, Multiple Data）実行モデルを使用して、64 ビット・レジスタ内のパックド整数データの並列処理を実行する。2.3. 節「SIMD 命令」を参照のこと。

2.1.6. P6 ファミリのプロセッサ（1995 ～ 1999 年）

P6 ファミリのプロセッサは、パフォーマンスの新しい基準を確立したスーパースケーラ・マイクロアーキテクチャに基づいている（2.2.1. 項「P6 ファミリ・マイクロアーキテクチャ」を参照）。P6 ファミリのマイクロアーキテクチャの開発目標の 1 つは、インテル® Pentium® プロセッサと同じ 0.6µm の 4 層メタル BICMOS 製造プロセスを使用して、インテル Pentium プロセッサの性能を大きく向上させることである。このファミリのメンバは以下のとおりである。

- インテル® Pentium® Pro プロセッサ
- インテル® Pentium® II プロセッサ
- インテル® Pentium® II Xeon™ プロセッサ
- インテル® Celeron® プロセッサ
- インテル® Pentium® III プロセッサ
- インテル® Pentium® III Xeon™ プロセッサ

インテル Pentium Pro プロセッサは、3 ウェイ・スーパースケーラ・アーキテクチャを持つ。このプロセッサは並列処理技法を利用し、1 クロック・サイクル当たり平均して 3 つの命令をデコード、ディスパッチ、完了（リタイヤ）できる。また、スーパースケーラ・アーキテクチャにはダイナミック・エグゼキューション（すなわち、マイクロデータ・フロー解析、アウトオブオーダー実行、高度な分岐予測、スペキュレーティブ・エグゼキューション）が導入された。このプロセッサはキャッシュによってさらに拡張されており、インテル Pentium プロセッサと同様の 2 つのオンチップ 8K バイト第 1 レベル（L1）キャッシュのほか、プロセッサと同じパッケージ内に 256K バイトの第 2 レベル（L2）キャッシュが追加された。

インテル Pentium II プロセッサでは、P6 ファミリ・プロセッサにインテルの MMX® テクノロジが追加され、新しいパッケージングといくつかのハードウェア的な拡張機能が採用された。プロセッサ・コアは、SECC（Single Edge Contact Cartridge）にパッケージされている。L1 データ・キャッシュと L1 命令キャッシュは、それぞれ 16K バイトに拡張された。L2 キャッシュのサイズは、256K バイト、512K バイト、1M バイトがサポートされている。L2 キャッシュは、「ハーフ・クロック・スピード」のバックサイド・バスによってプロセッサに接続される。また、AutoHALT、ストップグラント、スリープ、ディープスリープなどの各種の省電力状態がサポートされ、アイドル時間中の消費電力を軽減できる。

インテル Pentium II Xeon プロセッサは、前の世代のインテル・プロセッサのすぐれた特性を組み合わせた製品である。このプロセッサは、4 ウェイ、8 ウェイ（およびそれ以上）のスケーラビリティと、「フルクロック・スピード」のバックサイド・バス上で動作する 2M バイトの L2 キャッシュを備えている。

インテル Celeron プロセッサ・ファミリは、低価格 PC 市場向けの IA-32 アーキテクチャである。インテル Celeron プロセッサは、統合型の 128K バイト L2 キャッシュや、プラスチック・ピン・グリッド・アレイ（P.P.G.A）フォーム・ファクタなどの特徴を持ち、システムの設計コストの削減を可能にする。

インテル Pentium III プロセッサでは、IA-32 アーキテクチャにストリーミング SIMD 拡張命令（SSE）が導入された。SSE は、MMX テクノロジで導入された SIMD 実行モデルを拡張したものである。このプロセッサは、新しい 128 ビット・レジスタ・セットを搭載し、パックド単精度浮動小数点値の SIMD 演算を実行できる。2.3. 節「SIMD 命令」を参照のこと。

インテル Pentium III Xeon プロセッサは、フルスピードのオンダイ型アドバンスト・トランスファ・キャッシュを搭載し、IA-32 プロセッサの性能レベルを強化した製品である。

2.1.7. インテル® Pentium® 4 プロセッサ（2000 年）とハイパー・スレッディング・テクノロジ対応インテル® Pentium® 4 プロセッサ（2003 年）

高性能のインテル® Pentium® 4 プロセッサは、Intel NetBurst® マイクロアーキテクチャをベースにしている（2.2.2. 項「Intel NetBurst® マイクロアーキテクチャ」を参照）。このプロセッサでは、以下の主要な機能セットも導入された。

- ストリーミング SIMD 拡張命令 2（SSE2）。2.3. 節「SIMD 命令」を参照
- ストリーミング SIMD 拡張命令 3（SSE3）。2.3. 節「SIMD 命令」を参照

2.1.8. インテル® Xeon™ プロセッサ（2001 ～ 2003 年）

インテル® Xeon™ プロセッサも、Intel NetBurst® マイクロアーキテクチャをベースにしている（2.2.2. 項「Intel NetBurst® マイクロアーキテクチャ」を参照）。IA-32 プロセッサ中のこのグループは、1 つのファミリとして、マルチ・プロセッサのサーバシステムと高性能ワークステーション向けに設計されている。

インテル Xeon プロセッサ MP では、ハイパー・スレッディング（HT）テクノロジのサポートが開始された。2.3.1. 項「ハイパー・スレッディング・テクノロジ」を参照のこと。

2.1.9. インテル® Pentium® M プロセッサ（2003 年）

インテル® Pentium® M プロセッサは、前世代のインテル・モバイル・プロセッサのマイクロアーキテクチャを拡張した、高性能かつ低消費電力のモバイル・プロセッサである。このプロセッサには以下の機能が搭載されている。

- ダイナミック・エグゼキューションに対応したインテル・アーキテクチャをサポート
- カッパー・インターコネクトによるインテルの先進的な 0.13 ミクロン・プロセス・テクノロジを利用して製造された、高性能かつ低消費電力のコア
- オンダイの 1 次 32K バイト命令キャッシュと、32K バイトのライトバック・データ・キャッシュ
- アドバンスト・トランスファ・キャッシュ・アーキテクチャを採用した、オンダイの 1M バイト L2 キャッシュ
- 高度な分岐予測とデータ・プリフェッチ・ロジック
- MMX® テクノロジ、ストリーミング SIMD 命令、SSE2 命令セットをサポート
- 400MHz のソース・シンクロナス・プロセッサ・システムバス
- 拡張版 Intel SpeedStep® テクノロジによる省電力機能

2.2. 主な技術的進化の詳細

以下の各項では、IA-32 アーキテクチャの主な進化の詳細について説明する。

2.2.1. P6 ファミリ・マイクロアーキテクチャ

インテル® Pentium® Pro プロセッサでは、P6 プロセッサ・マイクロアーキテクチャと呼ばれる新しいマイクロアーキテクチャが導入された。その後、P6 プロセッサ・マイクロアーキテクチャは、アドバンスト・トランスファ・キャッシュと呼ばれるオンダイ L2 キャッシュによって拡張された。

マイクロアーキテクチャが、3 ウェイ・スーパースカラ方式のパイプライン・アーキテクチャである。3 ウェイ・スーパースカラとは、並列処理技法を使用するとプロセッサが 1 クロック・サイクルあたり平均して 3 つの命令をデコード、ディスパッチ、完了（リタイヤ）できることである。この高レベルの命令スループットを処理するために、P6 プロセッサ・ファミリではデカップリングされた 12 ステージのスーパーパイプラインを使用しており、これによって順序によらない（out-of-order）命令の実行をサポートしている。

図 2-1. は、アドバンスト・トランスファ・キャッシュによって拡張された、P6 プロセッサ・マイクロアーキテクチャのパイプラインの概念図を示している。

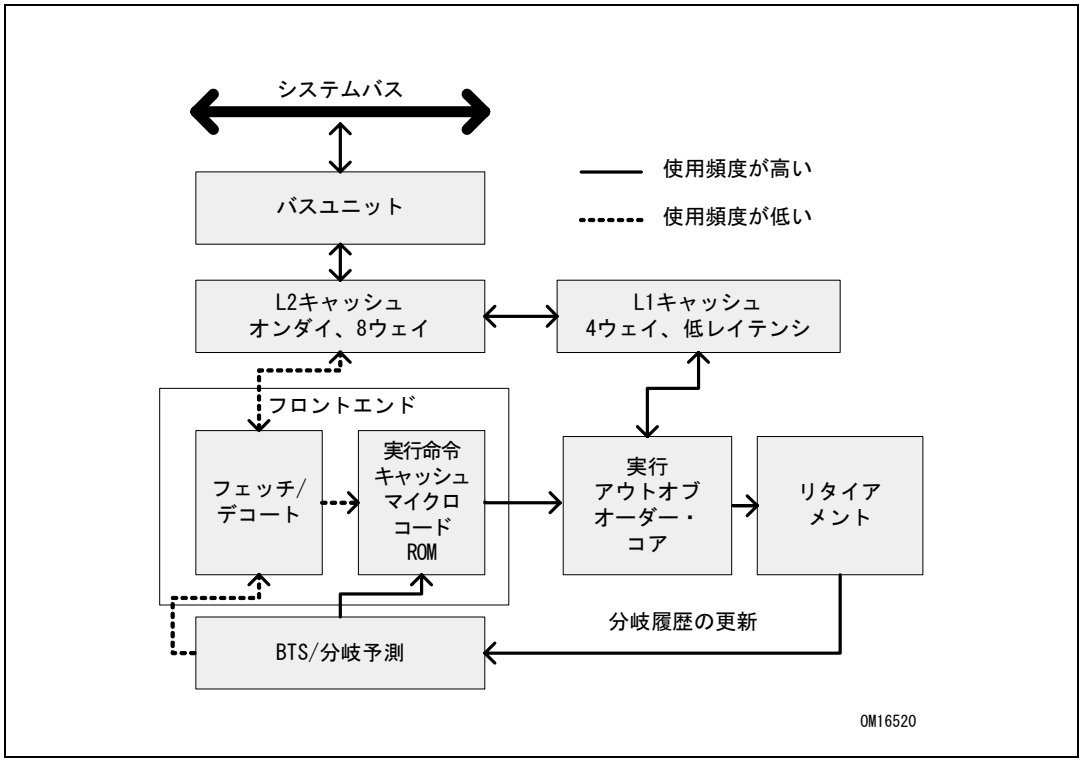


図 2-1. アドバンスト・トランスファ・キャッシュによって拡張された P6 プロセッサ・マイクロアーキテクチャ

命令とデータを確実に切れ目なく命令実行パイプラインに供給するために、P6 プロセッサのマイクロアーキテクチャには 2 つのレベルのキャッシュが内蔵されている。第 1 レベル・キャッシュは、8K バイトの命令キャッシュと 8K バイトのデータ・キャッシュで構成され、共にパイプラインに密にカップリングされている。第 2 レベル・キャッシュは 256K バイト、512k バイト、または 1M バイトのスタティック RAM を提供し、フル・クロック・レートで動作する 64 ビットのキャッシュ・バスを介してコア・プロセッサにカップリングされている。

P6 プロセッサのマイクロアーキテクチャの核となるのが、動的実行（dynamic execution）と呼ばれる革新的でアウトオブオーダー（out-of-order）な実行メカニズムである。この動的実行には、次の 3 つのデータ処理概念が取り入れられている。

- **高度な分岐予測**によって、命令パイプラインに切れ目が生じないように、プロセッサが分岐を超えて命令をデコードできる。P6 プロセッサ・ファミリでは、高度に最適化された分岐予測アルゴリズムを使用して、命令の方向を予測できる。

- **動的データフロー解析**では、プロセッサを経由するデータのフローをリアルタイムで解析し、依存関係を判定して、順序によらずに命令を実行することができるかどうかを検出する。アウトオブオーダー実行コア（out-of-order execution core）は、多数の命令をモニタし、処理の対象となっているデータの整合性を維持しながら、プロセッサが持つ複数の実行ユニットの最適な使用順序でこれらの命令を実行する。
- **スペキュレーティブ・エグゼキューション**は、プロセッサが、まだ解決されていない条件付き分岐の先にある命令を実行し、最終的に元の命令ストリームの順序でその結果を出力する機能である。推論による実行を可能にするために、P6 ファミリ・プロセッサのマイクロアーキテクチャでは、命令のディスパッチや実行を結果のコミットメントから切り離している。プロセッサのアウトオブオーダー実行コア（out-of-order execution core）は、データフロー解析を使用して命令プール内にあるすべての命令を実行し、その結果をテンポラリ・レジスタに格納する。次に、リタイヤユニットが命令プール内をリニヤに検索して、実行が完了した命令のうち、他の命令とのデータ依存関係がなく、未解決の分岐予測を持たない命令を探し出す。実行が完了したこれらの命令が見つかると、リタイヤユニットはこれらの命令の結果を、本来発行された順序でメモリや IA-32 アーキテクチャ・レジスタ（プロセッサが持つ 8 つの汎用レジスタと 8 つの x87 FPU データレジスタ）にコミットすると共に、命令を命令プールからリタイヤさせる。

2.2.2. Intel NetBurst® マイクロアーキテクチャ

Intel NetBurst® マイクロアーキテクチャには、次の機能がある。

- 高速実行エンジン
  - プロセッサの 2 倍の周波数で動作する算術論理ユニット（ALU）
  - プロセッサの 1/2 のクロック間隔で基本整数演算を実行
  - スループットの向上と実行レイテンシの削減
- ハイパー・パイプライン・テクノロジ
  - 深いパイプラインにより、デスクトップ PC およびサーバ用として業界トップレベルのクロック・レートを実現
  - 余裕のある周波数とスケーラビリティにより、将来もリーダーシップを維持
- 高度なダイナミック・エグゼキューション
  - 深いアウトオブオーダーのスペキュレーティブ実行エンジン
    - 最大 126 個の命令を段階的に処理
    - パイプライン内で最大 48 のロードと 24 のストアを処理[2]
  - 拡張された分岐予測機能

- パイプライン段数の増加による分岐の予測ミスのペナルティを軽減
- 高度な分岐予測アルゴリズム
- 4K エントリの分岐ターゲット配列

- 新しいキャッシュ・サブシステム
  - 1 次キャッシュ
    - 高度な実行トレース・キャッシュにより、デコード済みの命令を格納
    - 実行トレース・キャッシュにより、メイン実行ループからデコーダのレイテンシを除去
    - 実行トレース・キャッシュにより、プログラムの実行フローのパスを 1 つのラインに統合
    - レイテンシが小さいデータ・キャッシュ
  - 2 次キャッシュ
    - フルスピードのユニファイド 8 ウェイ 2 次オンダイ・アドバンスト・トランスファ・キャッシュ
    - プロセッサの周波数と共に帯域幅とパフォーマンスを向上
- Intel NetBurst マイクロアーキテクチャ・システム・バスに対する高性能クワッドポンプ型バス・インターフェイス
  - クワッドポンプ型のスケーラブルなバスクロックにより、実効速度を最大 4 倍に向上
  - 最大 3.2 ～ 6.4GB/ 秒の帯域幅を実現
- スーパースケーラ構造により並列処理が可能
- ハードウェア・レジスタを拡張してレジスタ名変更機能を追加し、レジスタ名空間の制限を解消
- 64 バイトのキャッシュ・ライン・サイズ（最大 2 ライン / セクタのデータを転送）

2. 90nm プロセスの Intel NetBurst マイクロアーキテクチャに基づく IA-32 プロセッサは、24 個のストアを処理できる。

図 2-2. は、Intel NetBurst マイクロアーキテクチャの概要を示している。このマイクロアーキテクチャ・パイプラインは、次の 3 つの部分で構成される。(1) フロントエンド・パイプ・ライン、(2) アウトオブオーダー実行コア、(3) リタイアメント・ユニット。

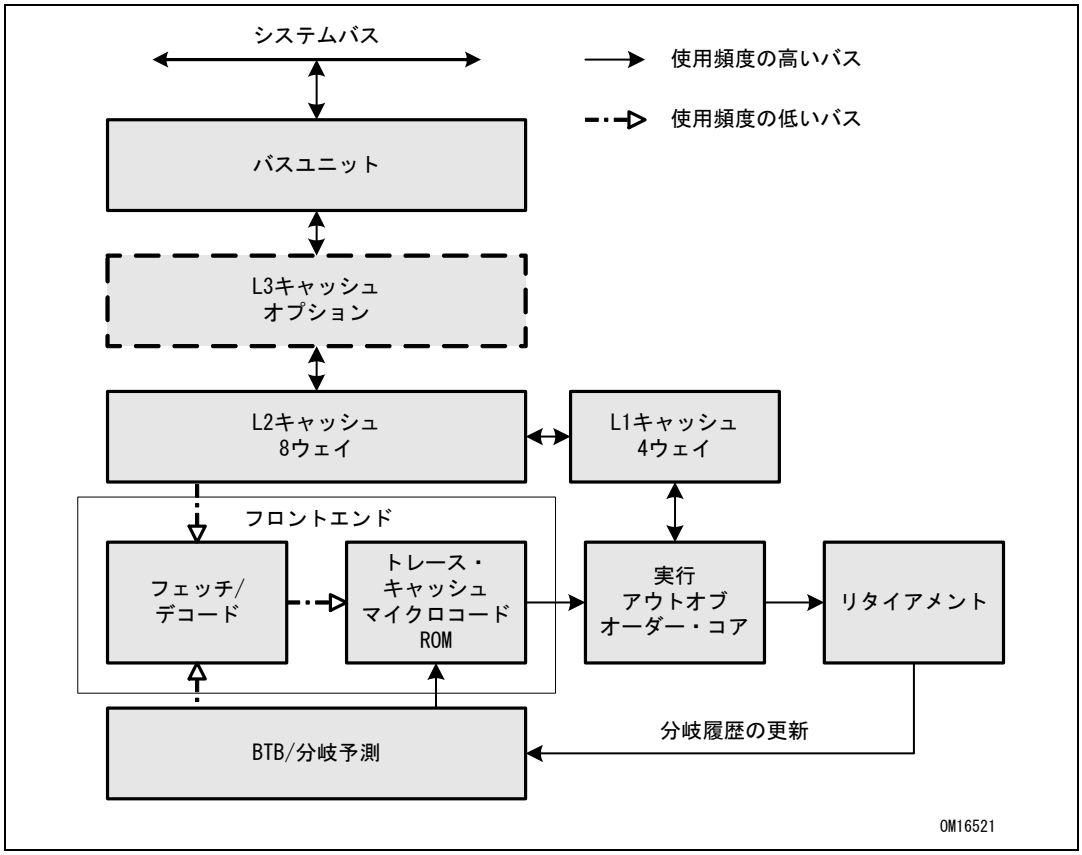


図 2-2. Intel NetBurst® マイクロアーキテクチャ

2.2.2.1. フロントエンド・パイプライン

フロントエンドは、命令をプログラムの順序でアウトオブオーダー・コアに供給する部分である。フロントエンドは、以下の機能を行う。

- 実行されそうな命令をプリフェッチする
- まだプリフェッチされていない命令をフェッチする
- IA-32 命令をデコードし、マイクロオペレーションに変換する
- 複雑な命令と特殊目的コード用のマイクロコードを生成する
- デコードされた命令を実行トレース・キャッシュから取り出す
- 積極的に分岐予測を実行する

パイプラインは、パイプライン型高速マイクロプロセッサの一般的な問題に対処するように設計されている。特に、次の2つの問題は、遅延の主な原因となる。

- ターゲットからフェッチされた命令のデコードに時間がかかる。
- 分岐または分岐ターゲットがキャッシュ・ラインの中間にあるために、デコード帯域幅が浪費される。

パイプラインのトレース・キャッシュの操作によって、これらの問題に対処できる。命令は、トランスレーション・エンジン（フェッチ/デコード・ロジックの一部）によって絶えずフェッチされてデコードされ、トレースと呼ばれる一連の μops に変換される。常に、複数のトレース（プリフェチされた分岐で表される）が、トレース・キャッシュに格納される。アクティブ分岐に後続する命令が、トレース・キャッシュ内で検索される。見つかった命令もプリフェチされた分岐内の最初の命令である場合は、メモリ階層からの命令のフェッチとデコードは中止され、そのプリフェチされた分岐が命令の新しいソースになる（図2-2. を参照）。

トレース・キャッシュとトランスレーション・エンジンは、協調する分岐予測ハードウェアを持つ。分岐ターゲットは、分岐ターゲット・バッファ（BTB）を使用して、リニアアドレスに基づいて予測され、できるだけ速やかにフェッチされる。

2.2.2.2. アウトオブオーダー実行コア

アウトオブオーダー実行コアが命令をアウトオブオーダーで実行できる機能は、並列処理を可能にする主要な要素である。この機能により、ある μops の処理が遅れる場合、プロセッサは命令の順序を変更して、他の μops を先に処理できる。プロセッサは、複数のバッファを使用して、μops の流れを円滑にする。

実行コアは、並列実行向けに設計されている。このコアは、1サイクル当たり最大6つの μops をディスパッチできる（この値は、トレース・キャッシュとリタイアメント・セクションの μops 帯域幅を超えていることに注意する）。ほとんどのパイプラインは、1サイクルごとに新しい μops の実行を開始できるため、各パイプラインで複数の命令を一度に段階的に処理できる。多くの算術論理ユニット（ALU）命令は、1サイクル当たり2つの μop を開始できる。多くの浮動小数点命令は、2サイクルごとに1つの μop を開始できる。

2.2.2.3. リタイヤ

リタイアメント・ユニットは、実行された μop の結果をアウトオブオーダー実行コアから受け取り、元のプログラムの順序にしたがってアーキテクチャ上の状態が更新されるように、それらの結果を処理する。

μops が完了し、結果が書き込まれた時点で、その μops はリタイヤされる。1サイクル当たり最大3つの μops をリタイヤさせることができる。リオーダー・バッファ

（ROB）は、完了した μops をバッファに入れる、アーキテクチャ・ステートを順序どおりに更新する、例外の順序を管理するなどの機能を持つ、プロセッサ内のユニットである。また、リタイアメント部は、分岐を監視し、更新された分岐ターゲット情報を BTB に送信する。次に、BTB は不要になったプリフェチされたトレースをパージする。

2.3. SIMD 命令

インテル® Pentium® II プロセッサ・ファミリおよびインテル® MMX® テクノロジ対応 インテル® Pentium® プロセッサ・ファミリ以降、4 つの拡張命令が IA-32 アーキテクチャに導入され、IA-32 プロセッサは SIMD（Single Instruction, Multiple Data）演算を実行できるようになった。この拡張命令とは、MMX テクノロジ、SSE、SSE2、SSE3 である。それぞれが提供する一連の命令は、64 ビット MMX レジスタまたは 128 ビット XMM レジスタ内のパックド整数やパックド浮動小数点のデータ要素に対して SIMD 演算を実行する。図 2-3. に、各種の SIMD 拡張命令（MMX テクノロジ、SSE、SSE2、SSE3）、処理対象のデータ型、データ型を MMX レジスタおよび XMM レジスタにパックする方法の概要を示す。

インテル MMX テクノロジは、インテル Pentium II プロセッサ・ファミリおよび MMX テクノロジ対応インテル Pentium プロセッサ・ファミリに導入された。MMX 命令は、MMX レジスタ内のパックドバイト、パックドワード、パックド・ダブルワードの整数に対して SIMD 演算を実行する。この命令は、SIMD 処理向けの整数配列および整数データのストリームを扱うアプリケーションに有効である。

SSE は、インテル® Pentium® III プロセッサ・ファミリに導入された。この命令は、XMM レジスタ内のパックド単精度浮動小数点値と、MMX レジスタ内のパックド整数を処理する。一部の SSE では、ステート管理、キャッシュ制御、メモリの順序づけ操作を実行する。それ以外の SSE は、単精度浮動小数点データ要素の配列を処理するアプリケーション（3D ジオメトリ、3D レンダリング、ビデオ・エンコード / デコード）が対象である。

SSE2 は、インテル® Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサに導入された。この命令は、XMM レジスタ内のパックド倍精度浮動小数点値と、MMX および XMM レジスタ内のパックド整数を処理する。SSE2 整数命令は、128 ビットの新しい SIMD 整数演算を追加し、既存の 64 ビット SIMD 整数演算を 128 ビットの XMM 機能に拡張することによって、IA-32 SIMD 演算を強化している。また、新たなキャッシュ制御とメモリの順序づけ操作も追加された。

SSE3 は、HT テクノロジ対応の インテル Pentium 4 プロセッサ（90nm プロセス・テクノロジがベース）に導入された。SSE3 では、SSE テクノロジ、SSE2 テクノロジ、x87-FP 演算機能の性能を高める 13 個の命令が追加されている。

以下も参照のこと。

- 5.4. 節「MMX® 命令」および第 9 章「インテル® MMX® テクノロジによるプログラミング」
- 5.5. 節「SSE」および第 10 章「ストリーミング SIMD 拡張命令（SSE）によるプログラミング」
- 5.6. 節「SSE2」および第 11 章「ストリーミング SIMD 拡張命令 2（SSE2）によるプログラミング」
- 5.7. 節「SSE3」および第 12 章「ストリーミング SIMD 拡張命令 3 （SSE3）によるプログラミング」

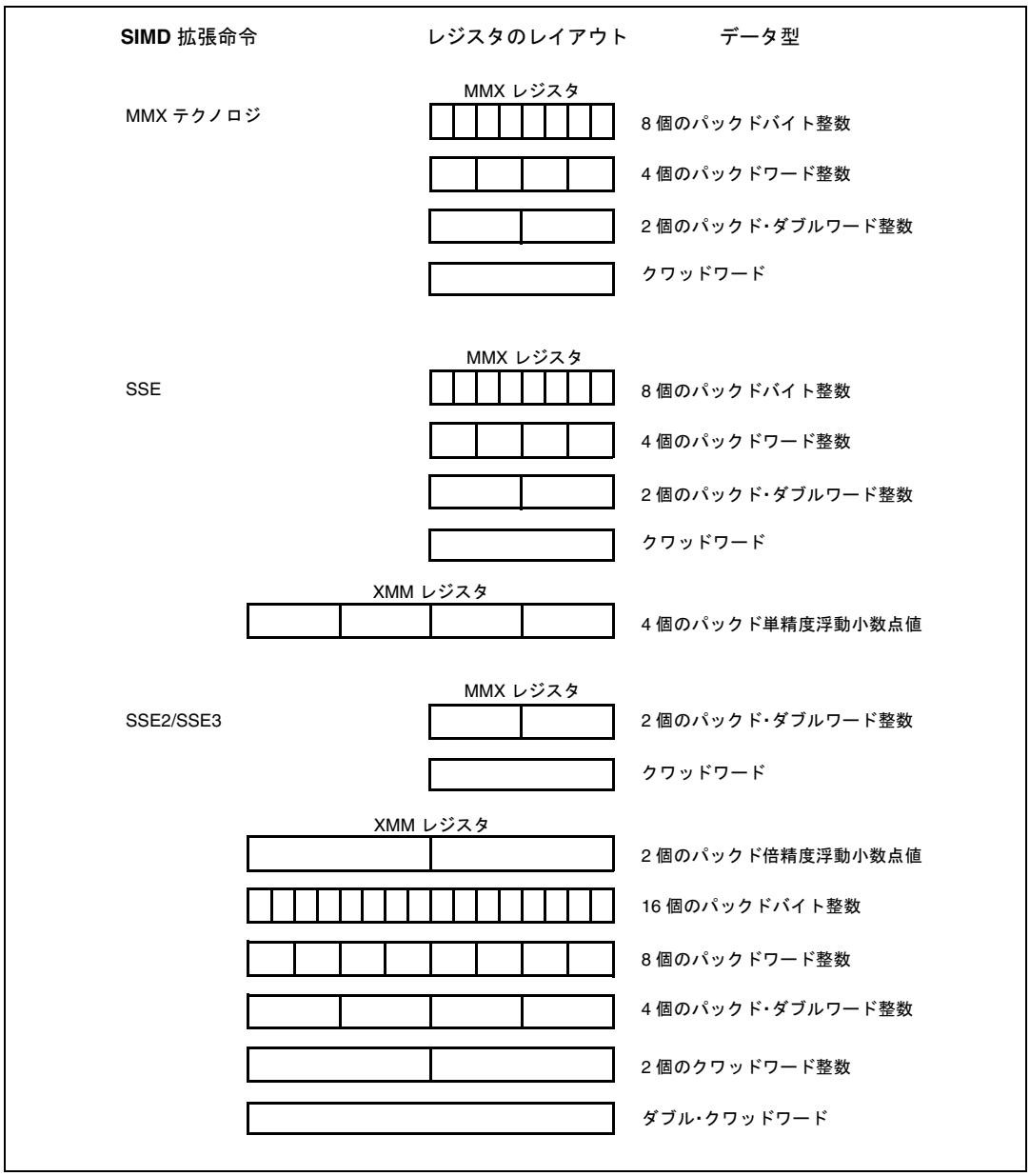


図 2-3. SIMD 拡張命令、レジスタのレイアウト、データ型

2.3.1. ハイパー・スレッディング・テクノロジ

ハイパー・スレッディング（HT）テクノロジは、マルチスレッドのオペレーティング・システムおよびアプリケーション・コードや、マルチタスク環境におけるシングルスレッド・アプリケーションを実行する際の IA-32 プロセッサの性能を向上するために開発された。このテクノロジを利用すると、単一の物理プロセッサ上で複数の異なるコード・ストリーム（スレッド）を同時に実行できる。

アーキテクチャ面で見ると、HT テクノロジに対応した IA-32 プロセッサは複数の論理プロセッサからなり、それぞれが個別の IA-32 アーキテクチャ・ステートを持っている。各論理プロセッサは、IA-32 データレジスタ、セグメント・レジスタ、コントロール・レジスタ、デバッグレジスタで構成され、MSR の大半も含まれている。さらにそれぞれが、個別の Advanced Programmable Interrupt Controller（APIC）を備えている。

図 2-4. では、HT テクノロジ対応プロセッサ（論理プロセッサを 2 つ搭載）と、従来のデュアルプロセッサ・システムを比較している。

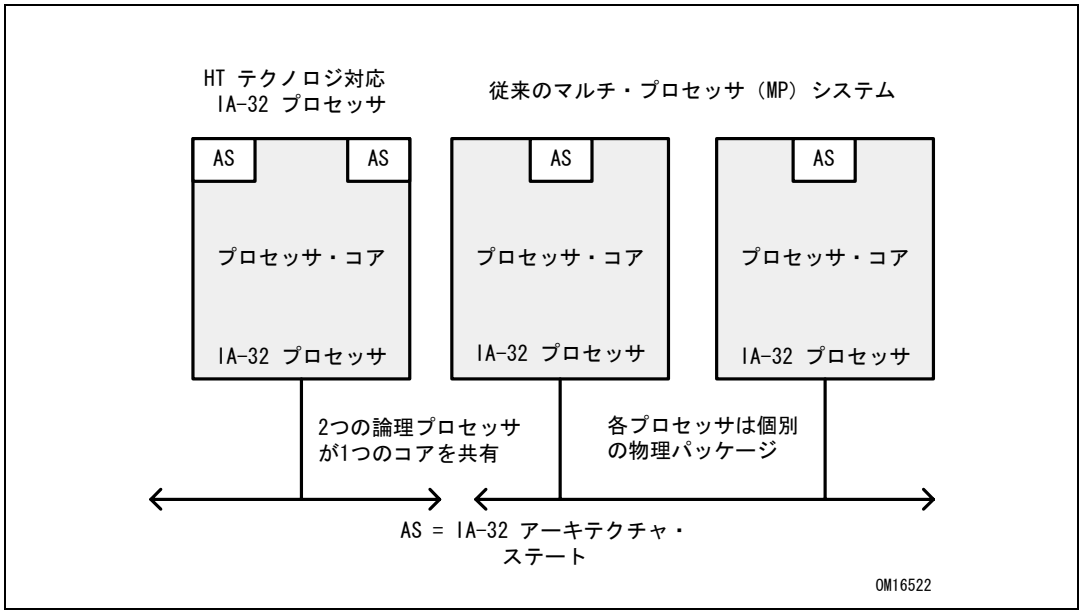


図 2-4. HT テクノロジ対応 IA-32 プロセッサと従来のデュアルプロセッサ・システムとの比較

個別の物理 IA-32 プロセッサを複数搭載した従来の MP システム構成と異なり、HT テクノロジ対応 IA-32 プロセッサ内の論理プロセッサは、物理プロセッサのコアのリソースを共有する。このリソースには、実行エンジンやシステムバス・インターフェイスも含まれる。電源投入および初期化後、各論理プロセッサに対して別々に、指定されたスレッドの実行、割り込み、停止を命令できる。

HT テクノロジは、単一のチップ上に複数の論理プロセッサを設ければ、先進的なオペレーティング・システムや高性能アプリケーションにおけるプロセスレベルおよびスレッドレベルの並列処理を活用している。この構成では、各物理プロセッサ上で同時に複数のスレッド[3] を実行できる。各論理プロセッサはアプリケーション・スレッドの命令実行時に、プロセッサ・コア内のリソースを使用する。コアは、アウトオブオーダー命令スケジューリングによってクロックサイクルごとの実行ユニット使用率を最大限に高めながら、各スレッドを同時に実行する。

2.3.1.1. 導入時の注意事項

すべての HT テクノロジ構成は、以下の要素を必要とする。

- HT テクノロジに対応したプロセッサ
- HT テクノロジに対応したチップセットおよび BIOS
- 最適化されたオペレーティング・システム

詳細は、http://www.intel.co.jp/jp/info/hyperthreading/ を参照のこと。

ファームウェア（BIOS）レベルでは、HT テクノロジ対応プロセッサ内の論理プロセッサを初期化するための基本的な手順は、従来の DP プラットフォームや MP プラットフォーム[4] と同じである。『Multiprocessor Specification, Version 1.4』で説明されている、MP システムの物理プロセッサに電源を投入し初期化するためのメカニズムが、HT テクノロジ対応プロセッサ内の論理プロセッサにも適用される。

従来の DP プラットフォームまたは MP プラットフォーム上で運用するように設計されたオペレーティング・システムは、CPUID を利用して、HT テクノロジ対応 IA-32 プロセッサの有無と、同プロセッサが持つ論理プロセッサの数を判断する。

従来のオペレーティング・システムやアプリケーション・コードも HT テクノロジ対応プロセッサ上で正しく動作するが、最大限のメリットを得るには、コードを一部修正することが推奨される。修正方法については、『IA-32 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、下巻』の第 7 章「マルチ・プロセッサ管理」の「必要なオペレーティング・システムのサポート」の項を参照のこと。

3. 本書では以後、「プロセス」および「スレッド」の総称として「スレッド」を用いる。
4. MP 初期化アルゴリズムに対する比較的簡単な修正が必要。

2.4. Moore の法則と IA-32 プロセッサの各世代

1960 年代半ばに、Gordon Moore（インテルの創立者で名誉会長）は、「今後数年間にわたって、CPU チップ 1 個当たりのトランジスタ数は 18 カ月ごとに 2 倍に増えるだろう」と予想した。「Moore の法則」として知られるこの予測は、その後 35 年間にわたって有効であった。

インテル・アーキテクチャ・プロセッサの処理能力と複雑さ（プロセッサ 1 個当たりのトランジスタ数にほぼ該当する）は、ほぼ Moore の法則にしたがって成長してきた。各世代の IA-32 プロセッサは、新しいプロセス技術と新設計のマイクロアーキテクチャを利用して、それ以前の世代のプロセッサより大幅に高い動作周波数とパフォーマンス・レベルを達成してきた。

表 2-1. に、高度なトランスファ・キャッシュを備えた インテル® Pentium® 4 プロセッサ、インテル® Xeon™ プロセッサ、インテル® Xeon™ プロセッサ MP、インテル® Pentium® III プロセッサ、インテル® Pentium® III Xeon™ プロセッサの主な特徴を示す。表 2-2. に、オンダイ L2 キャッシュを搭載しない過去の世代の IA-32 プロセッサの主な特徴を示す。

表 2-1. 最近の IA-32 プロセッサの主な特徴

インテル・プロセッサ	導入年	マイクロアーキテクチャ	導入時のクロック周波数	ダイ上のトランジスタ数	レジスタサイズ[1]	システムバスの帯域幅	最大外部アドレス空間	オンダイ・キャッシュ[2]
インテル Pentium 4 プロセッサ	2000	Intel NetBurst マイクロアーキテクチャ	1.50 GHz	42 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/秒	64 GB	12Kμop 実行トレース・キャッシュ、8KB L1、256KB L2
インテル Xeon プロセッサ	2001	Intel NetBurst マイクロアーキテクチャ	1.70 GHz	42 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/秒	64 GB	12Kμop トレース・キャッシュ、8-KB L1、256-KB L2
インテル Xeon プロセッサ	2002	Intel NetBurst マイクロアーキテクチャ、ハイパー・スレッディング・テクノロジ	2.20 GHz	55 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/秒	64 GB	12Kμop トレース・キャッシュ、8-KB L1、512-KB L2
インテル Xeon プロセッサ MP	2002	Intel NetBurst マイクロアーキテクチャ、ハイパー・スレッディング・テクノロジ	1.60 GHz	108 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/秒	64 GB	12Kμop トレース・キャッシュ、8-KB L1; 256-KB L2; 1-MB L3

表 2-1. 最近の IA-32 プロセッサの主な特徴

インテル・プロセッサ	導入年	マイクロアーキテクチャ	導入時のクロック周波数	ダイ上のトランジスタ数	レジスタサイズ[1]	システムバスの帯域幅	最大外部アドレス空間	オンダイ・キャッシュ[2]
ハイパー・スレッディング・テクノロジ インテル Pentium 4 プロセッサ	2002	Intel NetBurst マイクロアーキテクチャ、ハイパー・スレッディング・テクノロジ	3.06 GHz	55 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	4.2 GB/秒	64 GB	12Kμop 実行トレース・キャッシュ、8KB L1、512-KB L2
90 nm プロセス・ベースのハイパー・スレッディング・テクノロジ インテル Pentium 4 プロセッサ	2003	Intel NetBurst マイクロアーキテクチャ、ハイパー・スレッディング・テクノロジ	3.40 GHz	125 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	6.4 GB/秒	64 GB	12Kμop 実行トレース・キャッシュ、16KB L1、1MB L2
インテル Pentium M プロセッサ	2003	インテル Pentium M プロセッサ	1.60 GHz	77 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/秒	64 GB	L1: 64KB L2: 1MB

注 :

1. レジスタサイズと外部データ・バス・サイズの単位はビットである。
2. 1 次キャッシュは L1、2 次キャッシュは L2 で示す。L1 のサイズは、適用できる 1 次データ・キャッシュと命令キャッシュを含むが、トレース・キャッシュは含まない。

表 2-2. IA-32 プロセッサの過去の世代の主な特徴

インテル・プロセッサ	導入年	導入時の最大クロック周波数	ダイ上のトランジスタ数	レジスタサイズ[1]	外部データ・バス・サイズ[2]	最大外部アドレス空間	キャッシュ
8086	1978	8 MHz	29 K	16 GP	16	1 MB	None
インテル 286 プロセッサ	1982	12.5 MHz	134 K	16 GP	16	16 MB	Note 3
Intel386 DX プロセッサ	1985	20 MHz	275 K	32 GP	32	4 GB	Note 3
Intel486 DX プロセッサ	1989	25 MHz	1.2 M	32 GP 80 FPU	32	4 GB	L1: 8KB
インテル Pentium プロセッサ	1993	60 MHz	3.1 M	32 GP 80 FPU	64	4 GB	L1: 16KB
インテル Pentium Pro プロセッサ	1995	200 MHz	5.5 M	32 GP 80 FPU	64	64 GB	L1: 16KB L2: 256KB または 512KB
インテル Pentium II プロセッサ	1997	266 MHz	7 M	32 GP 80 FPU 64 MMX	64	64 GB	L1: 32KB L2: 256KB または 512KB

表 2-2. IA-32 プロセッサの過去の世代の主な特徴

インテル・プロセッサ	導入年	導入時の最大クロック周波数	ダイ上のトランジスタ数	レジスタサイズ[1]	外部データ・バス・サイズ[2]	最大外部アドレス空間	キャッシュ
インテル Pentium III プロセッサ	1999	500 MHz	8.2 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	L1: 132KB L2: 512KB
インテル Pentium III プロセッサ、インテル Pentium III Xeon プロセッサ	1999	700 MHz	28 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	L1: 32KB L2: 256KB

注：

1. レジスタサイズと外部データ・バス・サイズの単位はビットである。ただし、すべてのプロセッサ上で、各 32 ビット汎用（GP）レジスタは、8 ビットまたは 16 ビット・データ・レジスタとしてアドレス指定可能である。
2. 各プロセッサには、外部データバスの 2 ～ 4 倍の幅の内部データパスがある。

3

IA-32 基本実行環境

第 3 章
IA-32 基本実行環境

3

本章では、アセンブリ言語プログラマの視点から、IA-32 プロセッサの基本実行環境について説明する。さらに、プロセッサが命令を実行する方法や、データを格納し操作する方法についても説明する。本章で説明する実行環境に含まれるのは、メモリ（アドレス空間）、汎用データレジスタ、セグメント・レジスタ、EFLAGS レジスタ、命令ポインタレジスタである。

3.1. 動作モード

IA-32 アーキテクチャは、保護モード、実アドレスモード、システム管理モードの 3 種類の動作モードをサポートする。動作モードによって、どの命令やアーキテクチャ上の機能が使用できるかが決まる。

- **保護モード。**このモードは、プロセッサ本来の動作ステートである。このモードでは、すべての命令とアーキテクチャ上の機能が使用可能であり、最高の処理能力と機能が得られる。すべての新規アプリケーションやオペレーティング・システムに対しては、このモードを推奨する。保護モードの数々の機能の 1 つとして、「実アドレスモード」の 8086 ソフトウェアを保護されたマルチタスク環境で直接実行できる。この機能は、実際にはプロセッサのモードではないが、**仮想 8086** モードと呼ばれる。仮想 8086 モードは、実際には任意のタスクに対してイネーブルにできる保護モードの属性である。
- **実アドレスモード。**このモードは、インテル® 8086 プロセッサのプログラミング環境にいくつかの拡張機能（保護モードとシステム管理モードとの間の切り替えなど）を提供する。プロセッサは、電源投入やリセットの直後には実アドレスモードになる。
- **システム管理モード（SMM）。**このモードは、オペレーティング・システムやエグゼクティブに、電源管理やシステム・セキュリティなどのプラットフォーム固有の機能をインプリメントするための透過的な機構を提供する。プロセッサは、外部 SMM 割り込みピン（SMI#）がアクティブになるか、アドバンスド・プログラマブル割り込みコントローラ（APIC）から SMI を受け取った時点で SMM に移行する。SMM になると、プロセッサは現在実行されているプログラムあるいはタスクの基本的なコンテキストをセーブしてから、個々のアドレス空間に切り替える。これ以降、SMM 固有コードを透過的に実行できる。SMM から戻ると、プロセッサはシステム管理割り込みが発生する前のプロセッサ・ステートに戻される。SMM は、

Intel386™ SL プロセッサおよび Intel486™ SL プロセッサで導入され、インテル® Pentium® プロセッサ・ファミリで IA-32 の標準機能になった。

基本実行環境は、本章の以降の各節で説明しているように、これらの動作モードそれぞれにおいて同じである。

3.2. 基本実行環境の概要

IA-32 プロセッサ上で動作するプログラムやタスクには、命令の実行や、コード、データ、ならびにステート情報を格納するためのリソースが与えられる。これらのリソース（以下に簡単に説明する。図 3-1. を参照）は、IA-32 プロセッサの基本実行環境を構成する。この基本実行環境は、プロセッサ上で実行されるアプリケーション・プログラムとオペレーティング・システムまたはエグゼクティブによって共同で使用される。

- **アドレス空間**：IA-32 プロセッサ上で実行されるタスクまたはプログラムは、最大 4G バイト（2^{32} バイト）のリニアアドレス空間と、最大 64G バイト（2^{36} バイト）の物理アドレス空間をアドレス指定することができる。4G バイトを超えるアドレス空間のアドレス指定についての詳細は、3.3.3. 項「拡張された物理アドレス指定」を参照のこと。
- **基本プログラム実行レジスタ**：8 個の汎用レジスタ、6 個のセグメント・レジスタ、EFLAGS レジスタ、および EIP（命令ポインタ）レジスタが、一連の汎用命令を実行するための基本実行環境を構成する。これらの命令は、バイト整数、ワード整数、ダブルワード整数の整数演算、プログラム・フロー制御、ビット・ストリングとバイト・ストリングの処理、メモリのアドレス指定を実行する。これらのレジスタについての詳細は、3.4. 節「基本プログラム実行レジスタ」を参照のこと。
- **x87 FPU レジスタ**：8 個の x87 FPU データレジスタ、x87 FPU コントロール・レジスタ、ステータス・レジスタ、x87 FPU 命令ポインタレジスタ、x87 FPU オペランド（データ）ポインタ・レジスタ、x87 FPU タグレジスタ、x87 FPU オペコード・レジスタが、単精度 / 倍精度 / 拡張倍精度の浮動小数点値、ワード / ダブルワード / クワッドワード整数、2 進化 10 進数（BCD）の演算用の実行環境となる。これらのレジスタについての詳細は、8.1. 節「x87 FPU の実行環境」を参照のこと。
- **MMX® テクノロジ・レジスタ**：8 個の MMX テクノロジ・レジスタは、64 ビットのパックドバイト / ワード / ダブルワード整数の SIMD（Single Instruction, Multiple Data）演算の実行をサポートする。これらのレジスタについての詳細は、9.2. 節「MMX® テクノロジのプログラミング環境」を参照のこと。
- **XMM レジスタ**：8 個の XMM データレジスタと MXCSR レジスタは、128 ビットのパックド単精度 / 倍精度浮動小数点値の SIMD 演算と、128 ビットのパックドバイト / ワード / ダブルワード / クワッドワード整数の SIMD 演算をサポートする。これらのレジスタについての詳細は、10.2. 節「SSE のプログラミング環境」を参照のこと。

- **スタック**：プロシージャまたはサブルーチンの呼び出しと、プロシージャまたはサブルーチン間でのパラメータの受け渡しをサポートするために、実行環境にスタックとスタック管理リソースが含まれている。スタック（図 3-1. には示されていない）はメモリ内に置かれる。スタックの構造についての詳細は、6.2. 節「スタック」を参照のこと。

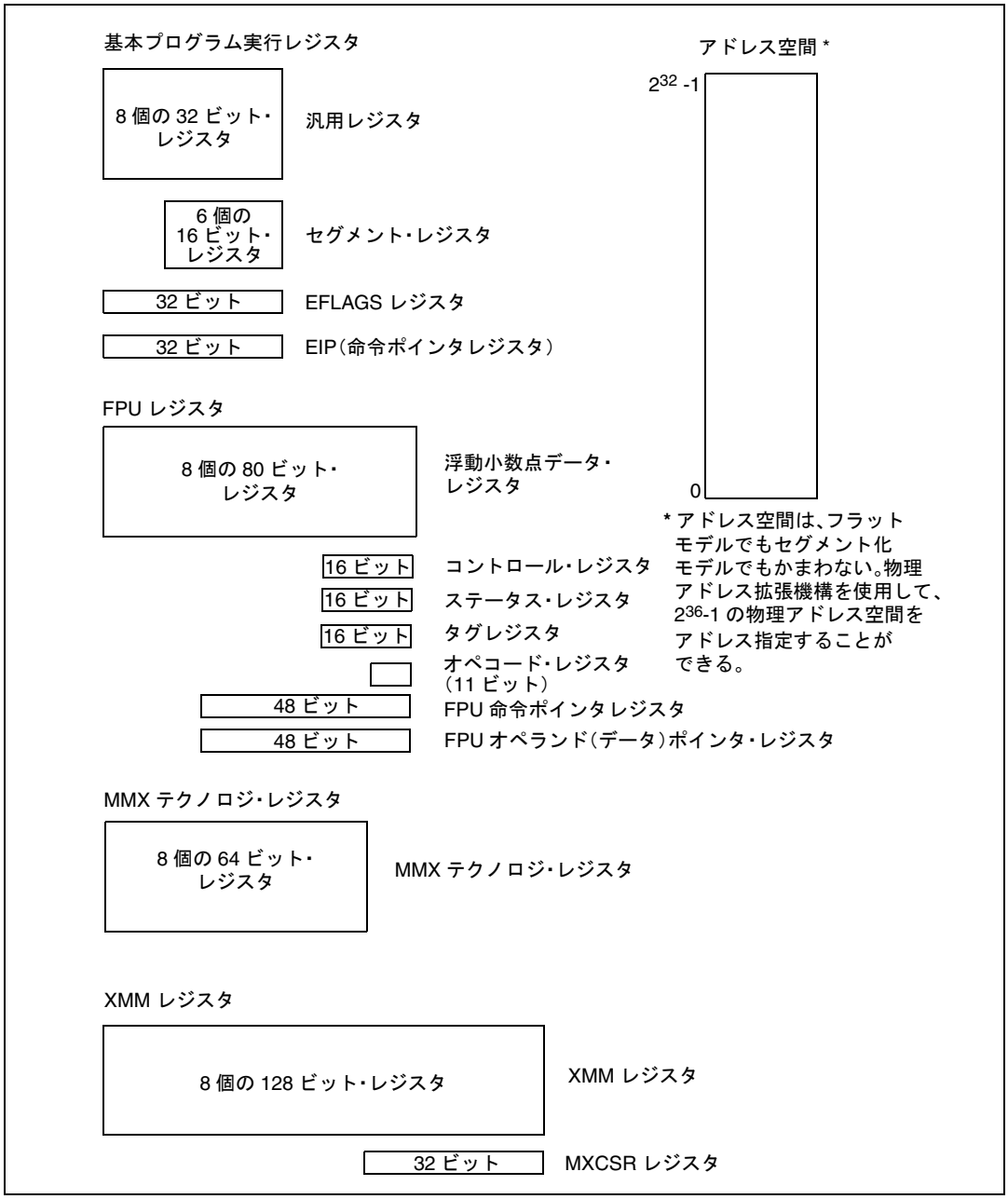


図 3-1. IA-32 の基本実行環境

IA-32 アーキテクチャは、基本実行環境のリソース以外に、システムレベル・アーキテクチャの一部として、次のようなシステムリソースを備えている。これらのリソースは、オペレーティング・システムとシステム開発ソフトウェアを広範囲にわたってサポートする。I/O ポート以外のシステムリソースについての詳細は、『IA-32 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、下巻 : システム・プログラミング・ガイド』を参照のこと。

- **I/O ポート** : IA-32 アーキテクチャは、入力 / 出力（I/O）ポートとの間のデータ転送をサポートしている。本巻の第 13 章「入出力」を参照のこと。
- **コントロール・レジスタ** : 5 個のコントロール・レジスタ（CR0 ～ CR5）は、プロセッサの動作モードと、現在実行中のタスクの特性を指定する。『IA-32 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、下巻』の「コントロール・レジスタ」の項を参照のこと。
- **メモリ管理レジスタ** : GDTR、IDTR、タスクレジスタ、LDTR は、プロテクト・モードのメモリ管理に使用されるデータ構造の位置を指定する。『IA-32 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、下巻』の第 2 章の「メモリ管理レジスタ」の項を参照のこと。
- **デバッグレジスタ** : デバッグレジスタ（DR0 ～ DR7）は、プロセッサのデバッグ動作の監視機能を制御する。『IA-32 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、下巻』の第 15 章の「デバッグレジスタ」の項を参照のこと。
- **メモリタイプ範囲レジスタ（MTRR）** : MTRR を使用して、メモリタイプをメモリの領域に割り当てることができる。『IA-32 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、下巻』の第 10 章の「メモリタイプ範囲レジスタ [MTRR]」の項を参照のこと。
- **マシン固有レジスタ（MSR）** : プロセッサは、プロセッサのパフォーマンスの制御とレポートに使用される各種のマシン固有レジスタを搭載している。事実上すべての MSR は、システム関連機能を処理するためにあり、アプリケーション・プログラムは MSR にアクセスできない。ただし、タイムスタンプ・カウンタは例外である。MSR については、『IA-32 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、下巻』の付録 B「モデル固有レジスタ（MSR）」を参照のこと。
- **マシン・チェック・レジスタ** : マシン・チェック・レジスタは、ハードウェア（マシン）エラーの検出と報告に使用される、一連のコントロール・レジスタ、ステータス・レジスタ、エラー報告 MSR で構成される。『IA-32 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、下巻』の第 14 章の「マシンチェック MSR」の項を参照のこと。

- **パフォーマンス監視カウンタ**：パフォーマンス監視カウンタは、監視対象となるプロセッサ・パフォーマンス・イベントの中に含まれるものである。『IA-32 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、下巻』の第 15 章の「パフォーマンス監視の概要」の項を参照のこと。

本章の後半では、メモリの構成とアドレス空間、基本プログラム実行レジスタ、アドレス指定モードについて説明する。図 3-1. に記載されている、その他のプログラム実行リソースについては、本巻の以下の章を参照のこと。

- x87 FPU レジスタ ― 第 8 章「x87 FPU による プログラミング」を参照。
- MMX テクノロジ・レジスタ ― 第 9 章「インテル® MMX® テクノロジによるプログラミング」を参照。
- XMM レジスタ ― 第 10 章「ストリーミング SIMD 拡張命令（SSE）によるプログラミング」、第 11 章「ストリーミング SIMD 拡張命令 2（SSE2）によるプログラミング」、第 12 章「ストリーミング SIMD 拡張命令 3 （SSE3）によるプログラミング」を参照。
- スタックの実装とプロシージャ・コール ― 第 6 章「プロシージャ・コール、割り込み、例外」を参照。

3.3. メモリの構成

プロセッサがそのバス上でアドレス指定するメモリは、**物理メモリ**と呼ばれる。物理メモリは、8 ビットのバイト・シーケンスとして構成される。それぞれのバイトには、**物理アドレス**と呼ばれる一意のアドレスが割り当てられる。**物理アドレス空間**は、0 ～ 2^{36}-1（64G バイト）の範囲をとる。

IA-32 プロセッサ上で動作するよう設計されたオペレーティング・システムやエグゼクティブでは、プロセッサのメモリ管理機能を使用してメモリにアクセスする。これらの機能には、効率よく、しかも高い信頼性でメモリを管理をするためのセグメンテーションやページングなどが含まれる。メモリ管理については、『IA-32 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、下巻』の第 3 章「保護モードにおけるメモリ・マネージメント」で詳しく説明している。これ以降の各項では、メモリ管理を利用してメモリをアドレス指定する際の基本的な方法について説明する。

プロセッサのメモリ管理機能を使用する場合は、プログラムで物理メモリに対して直接にアドレス指定することはない。代わりに、プログラムは、3 つのメモリモデル（フラット、セグメント化、実アドレスモード）のいずれかを使用してメモリにアクセスする。

フラット・メモリ・モデル（図 3-2. を参照）では、メモリは、プログラムの視点からは、**リニアアドレス空間**と呼ばれる単一の連続したアドレス空間のように見える。コード（すなわちプログラムの命令）、データ、プロシージャ・スタックはすべて、このアドレス空間に格納される。リニアアドレス空間では、バイトによるアドレス指定が可能であり、アドレスは 0 ～ $2^{32} - 1$ の範囲で連続している。リニアアドレス空間内の任意のバイトに対するアドレスは、**リニアアドレス**と呼ばれる。

セグメント化メモリモデルでは、メモリは、プログラムの視点からは、**セグメント**と呼ばれる独立したアドレス空間のグループのように見える。このモデルを使用する場合、コード、データ、およびスタックは、一般的には独立したセグメントに格納される。セグメント内のバイトをアドレス指定するには、プログラムによって、セグメント・セレクタとオフセットで構成される**論理アドレス**を発行しなければならない。（論理アドレスは、far **ポインタ**と呼ばれる。）**セグメント・セレクタ**で、アクセスの対象となるセグメントを識別し、オフセットでそのセグメントのアドレス空間にあるバイトを識別する。IA-32 プロセッサ上で動作するプログラムでは、異なるサイズとタイプのセグメントを最大 16,383 までアドレス指定できる。また、各セグメントのサイズは、最大 2^{32} バイトまでである。

内部的には、システムに対して定義されたセグメントはすべて、プロセッサのリニアアドレス空間にマッピングされる。したがって、プロセッサがメモリにアクセスするときに、プロセッサはそれぞれの論理アドレスをリニアアドレスに変換する。この変換は、アプリケーション・プログラムからは透過である。

セグメント化メモリを使用する最大の理由は、プログラムやシステムの信頼性を向上させることにある。例えば、プログラムのスタックを別個のセグメントに配置すると、スタックが大きくなってコード空間やデータ空間にまで入り込み、命令やデータが上書きされてしまうのを防止できる。オペレーティング・システムやエグゼクティブのコード、データ、スタックを別々のセグメントに配置すれば、アプリケーション・プログラムとの間で相互に保護もできる。

フラットまたはセグメント化メモリモデルでは、リニアアドレス空間が、直接またはページングを使用して、プロセッサの物理アドレス空間にマッピングされる。直接マッピングを使用する場合（ページング無効）は、各リニアアドレスは物理アドレスに 1 対 1 で対応する（つまり、リニアアドレスは、変換されずにプロセッサのアドレスラインに送られる）。IA-32 アーキテクチャのページ機構を使用する場合（ページング有効）は、リニアアドレス空間はページに分割され、各ページが仮想メモリにマッピングされる。

仮想メモリのページは、必要に応じて物理メモリにマッピングされる。オペレーティング・システムまたはエグゼクティブがページングを使用する際は、このページング機構はアプリケーション・プログラムからは透過的である。つまり、アプリケーション・プログラムは、リニアアドレス空間だけを認識する。

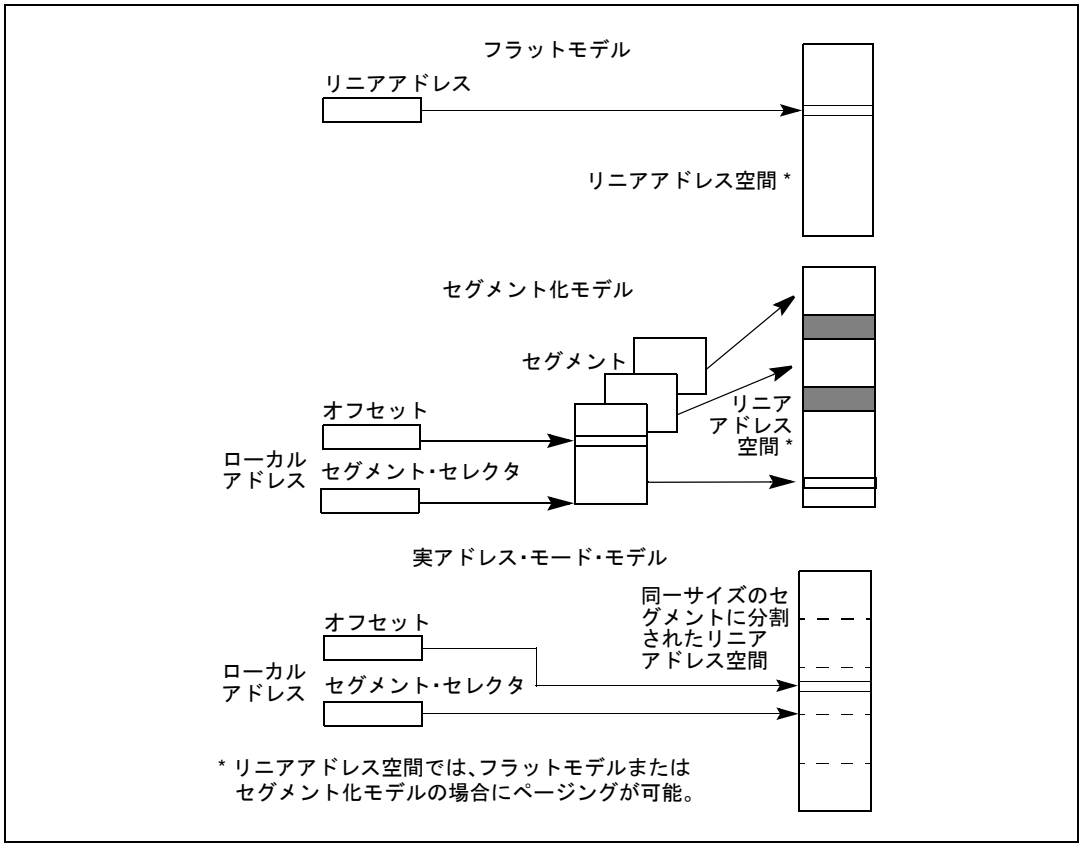


図 3-2. 3 つのメモリ管理モデル

実アドレス・モード・メモリ・モデルでは、インテル® 8086 プロセッサのメモリモデルが使用される。このメモリモデルは、インテル 8086 プロセッサ上で動作するように開発された既存のプログラムとの互換性を維持するために、IA-32 アーキテクチャ上でサポートされている。実アドレスモードでは、セグメント化メモリの特定のインプリメンテーションを使用して、プログラムやオペレーティング・システムあるいはエグゼクティブ用のリニアアドレス空間は、それぞれ最大 64K バイトのサイズのセグメント配列で構成される。実アドレスモードにおけるリニアアドレス空間の最大サイズは、2^{20} バイトである。このメモリモデルの詳細については、『IA-32 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、下巻』の第 16 章「8086 エミュレーション」を参照のこと。

3.3.1. 動作モード対メモリモデル

IA-32 プロセッサ用のコードを開発するときは、プロセッサがコードを実行する際の動作モードやメモリモデルをプログラマは理解しておかなければならない。動作モードとメモリモデルとの関係は、次のようになる。

- **保護モード。**保護モードでは、プロセッサは前節で説明した任意のメモリモデルを使用できる。（実アドレスモードのメモリモデルは、通常は、プロセッサが仮想 8086 モードにある場合にのみ使用する。）いずれのメモリモデルを使用するかは、オペレーティング・システムやエグゼクティブの設計によって決まる。マルチタスクがインプリメントされている場合は、個々のタスクで異なるメモリモデルを使用できる。
- **実アドレスモード。**実アドレスモードでは、プロセッサは実アドレスモードのメモリモデルしかサポートしない。
- **システム管理モード（SMM）。**SMM では、プロセッサはシステム管理モード RAM（SMRAM）と呼ばれる独立したアドレス空間に切り替える。このアドレス区間内のバイトをアドレス指定する際に使用されるメモリモデルは、実アドレス・モード・モデルと同じである。SMM で使用されるメモリモデルの詳細については、『IA-32 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、下巻』の第 13 章「システム管理モード（SMM）」を参照のこと。

3.3.2. 32 ビットと 16 ビットのアドレスサイズとオペランド・サイズ

高度な IA-32 プロセッサは、32 ビットあるいは 16 ビットのアドレスサイズとオペランド・サイズを設定できる。32 ビットのアドレスサイズとオペランド・サイズを使用する場合は、最大のリニアアドレスまたはセグメント・オフセットは FFFFFFFFH（2^{32}-1）になり、オペランド・サイズは一般的には 8 ビットか 32 ビットになる。16 ビットのアドレスサイズとオペランド・サイズを使用する場合は、最大のリニアアドレスまたはセグメント・オフセットは FFFFH（2^{16}-1）になり、オペランド・サイズは一般的には 8 ビットか 16 ビットになる。

32 ビットのアドレス指定を使用する場合は、論理アドレス（すなわち far ポインタ）は 16 ビットのセグメント・セレクタと 32 ビットのオフセットで構成される。一方、16 ビットのアドレス指定を使用する場合は、論理アドレスは 16 ビットのセグメント・セレクタと 16 ビットのオフセットで構成される。

命令プリフィックスを使用すれば、プログラム内でデフォルトのアドレスサイズやオペランド・サイズを一時的にオーバーライドすることが可能である。

保護モードで動作する場合は、デフォルトのアドレスサイズとオペランド・サイズは、現在実行されているコード・セグメントのセグメント・ディスクリプタによって定義される。セグメント・ディスクリプタは、アプリケーション・コードからは通常見る

ことができないシステム・データ構造の 1 つである。アセンブラの指示語を使用すれば、プログラムに対してデフォルトのアドレス指定サイズとオペランド・サイズを選択することができる。この後、アセンブラや他のツールによって、コード・セグメントのセグメント・ディスクリプタが正しくセットアップされる。

実アドレスモードで動作する場合は、デフォルトのアドレス指定サイズとオペランド・サイズは、16 ビットになる。実アドレスモードでは、アドレスサイズをオーバーライドすることで 32 ビットのアドレス指定が可能になるが、32 ビットにおいても、使用できる最大アドレスは 000FFFFFH（2^{20}-1）である。

3.3.3. 拡張された物理アドレス指定

IA-32 アーキテクチャは、P6 ファミリ・プロセッサ以来、最大 64G バイト（2^{36} バイト）の物理メモリのアドレス指定をサポートしている。プログラムまたはタスクは、このアドレス空間内の位置を直接アドレス指定することはできない。代わりに、プログラムまたはタスクは、最大 4G バイトのリニアアドレス空間を個別にアドレス指定する。このリニアアドレス空間が、プロセッサの仮想メモリ管理機構によって、より大きな 64G バイトの物理アドレス空間にマッピングされる。プログラムは、セグメント・レジスタ内のセグメント・セレクタを変更すれば、この 64G バイトの物理アドレス空間内でリニアアドレス空間を切り替えられる。

拡張された物理アドレス指定を使用するには、プロセッサがプロテクト・モードで動作し、オペレーティング・システムが仮想メモリ管理システムを提供する必要がある。このアドレス指定機構についての詳細は、『IA-32 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、下巻』の第 3 章の「物理アドレス拡張」を参照のこと。

3.4. 基本プログラム実行レジスタ

プロセッサには、汎用システムやアプリケーションのプログラミングで使用するために、16 個のレジスタの基本プログラム実行が用意されている。図 3-3. に示すように、これらのレジスタは次のグループに分類できる。

- **汎用データ レジスタ。**これら 8 つのレジスタは、オペランドやポインタを格納するのに使用できる。
- **セグメント・レジスタ。**これらのレジスタは、最大 6 つのセグメント・セレクタを保持できる。
- **EFLAGS（プログラム・ステータス / コントロール）レジスタ**：EFLAGS レジスタは、実行中のプログラムのステータスを示す。また、プロセッサを限定的に（アプリケーション・プログラム・レベルで）制御できる。
- **EIP（命令ポインタ）レジスタ**：EIP レジスタは、次に実行される命令を指す 32 ビット・ポインタを格納する。

3.4.1. 汎用レジスタ

プロセッサには、8 つの 32 ビット汎用レジスタ（EAX、EBX、ECX、EDX、ESI、EDI、EBP、および ESP）が搭載されており、次の項目を保持する。

- 論理演算と算術演算用のオペランド
- アドレス計算用のオペランド
- メモリポインタ

これらのレジスタはいずれも、オペランド、結果、ポインタの汎用記憶領域として使用できるが、ESP レジスタを参照する場合は注意が必要である。ESP レジスタは、スタックポインタを保持するためのもので、原則としてこれ以外の用途に使用することは避けなければならない。

命令の多くは、オペランドを保持するために特定のレジスタを割り当てる。例えば、ストリング命令は ECX、ESI、EDI の各レジスタの内容をオペランドとして使用する。また、セグメント化メモリモデルを使用する場合、命令によっては特定のレジスタのポインタを特定のセグメントと関連づけるものがある。例えば、一部の命令では、EBX レジスタのポインタは DS セグメント内のメモリ・ロケーションを指しているものとみなす。

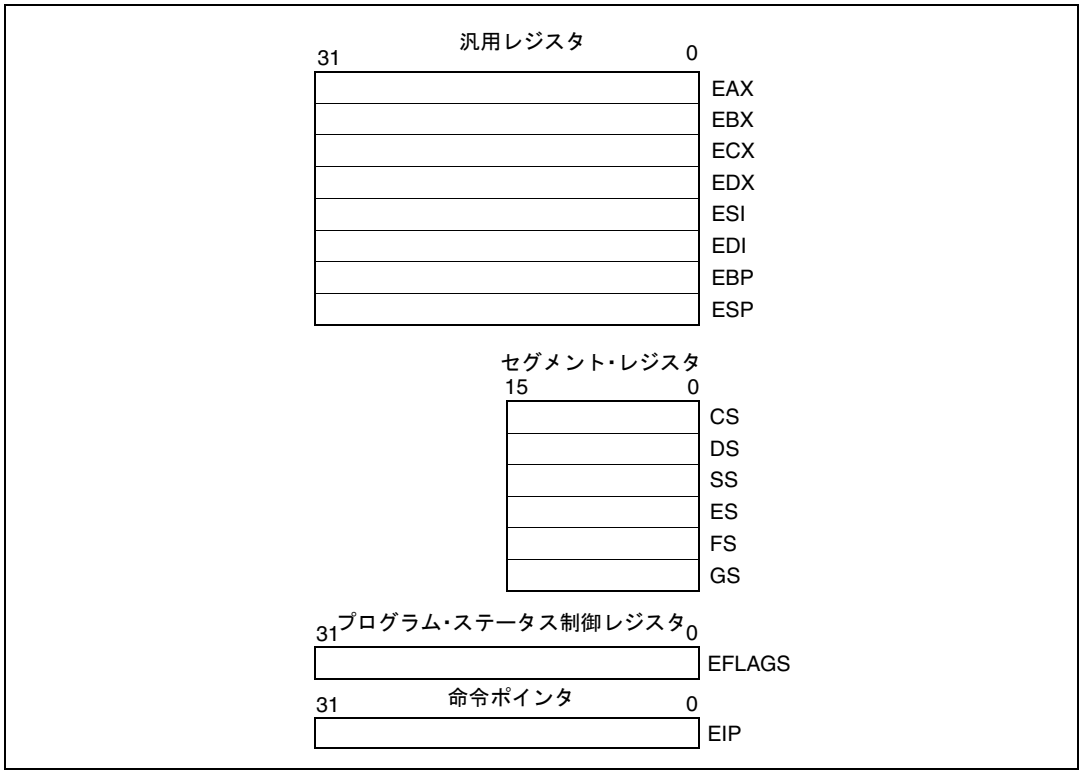


図 3-3. 汎用システムおよびアプリケーション・プログラミング・レジスタ

命令による汎用レジスタの特殊な使用法については、本書の第 5 章「命令セットの要約」で説明している。『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」も参照のこと。これらの特殊な使用法としては、次のものがある。

- EAX － オペランドと結果データ用のアキュムレータ。
- EBX － DS セグメント内のデータに対するポインタ。
- ECX － ストリング操作およびループ操作用のカウンタ。
- EDX － I/O ポインタ。
- ESI － DS レジスタがポイントするセグメント内のデータに対するポインタ；ストリング操作ではソースポインタ。
- EDI － ES　レジスタがポイントするセグメント内のデータ（またはデスティネーション）に対するポインタ；ストリング操作ではデスティネーション・ポインタ。
- ESP －（SS セグメント内の）スタックポインタ。

- EBP －（SSセグメント内の）スタック上のデータに対するポインタ。

図3-4.に示すように、汎用レジスタの下位16ビットは、8086やインテル® 286プロセッサのレジスタセットに直接マッピングされ、それぞれAX、BX、CX、DX、BP、SP、SI、DIの名前で参照できる。EAX、EBX、ECX、EDXの各レジスタの下位2バイトはそれぞれ、AH、BH、CH、DH（上位バイト）とAL、BL、CL、DL（下位バイト）の名前で参照できる。

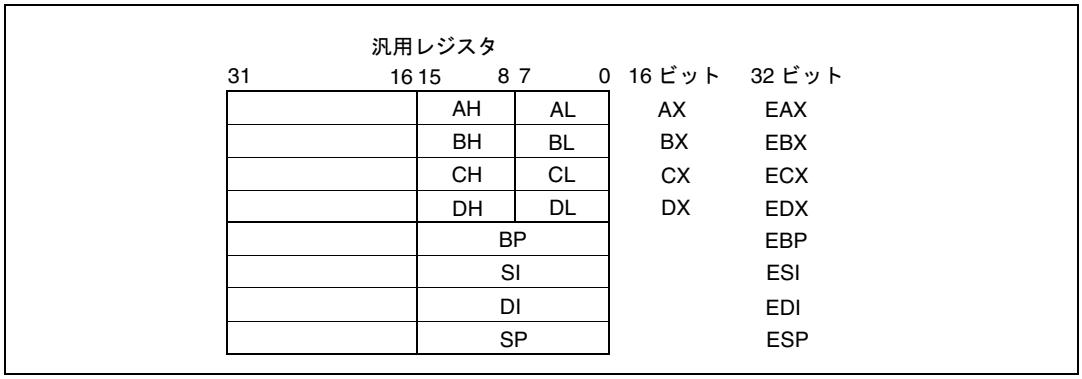


図 3-4. 汎用レジスタの代替名

3.4.2. セグメント・レジスタ

セグメント・レジスタ（CS、DS、SS、ES、FS、GS）は、16ビットのセグメント・セレクタを保持する。セグメント・セレクタは、メモリ内のセグメントを識別する特殊なポインタである。メモリ内の特定のセグメントにアクセスするには、そのセグメントに対するセグメント・セレクタが対応するセグメント・レジスタ内になければならない。

アプリケーション・コードを開発する際は、一般的には、ユーザがまずアセンブラの指示語とシンボルを使用してセグメント・セレクタを作成する。これらの指示語やシンボルに関連付けられた実際のセグメント・セレクタ値は、この後でアセンブラや他のツールによって生成される。システムコードを開発している場合は、ユーザがセグメント・セレクタを直接作成しなければならない場合もある。セグメント・セレクタのデータ構造についての詳細な説明は、『IA-32 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、下巻』の第3章「保護モードにおけるメモリ・マネージメント」を参照のこと。

セグメント・レジスタがどのように使用されるかは、オペレーティング・システムやエグゼクティブが使用しているメモリ管理モデルのタイプによって異なる。フラットな（セグメント化されていない）メモリモデルを使用する場合は、セグメント・レジスタには、オーバーラップするセグメントをポイントするセグメント・セレクタがロードされる。このオーバーラップする各セグメントは、リニアアドレス空間のアド

レス0から始まる（図3-5.を参照）。プログラム用のリニアアドレス空間は、オーバーラップするこれらのセグメントによって構成される。一般的には、コード用に1つ、データとスタック用に1つ、合計2つのオーバーラップするセグメントが定義される。CSセグメント・レジスタはコード・セグメントをポイントし、それ以外のセグメント・レジスタはデータとスタックのセグメントをポイントする。

セグメント化メモリモデルを使用する場合は、一般的にはリニアアドレス空間内の異なるセグメントをポイントできるよう、それぞれのセグメント・レジスタには異なるセグメント・セレクタがロードされる（図3-6.を参照）。これにより、プログラムが任意の時点でリニアアドレス空間にあるセグメントを最大6つまでアクセスできる。どのセグメント・レジスタもポイントしないセグメントにアクセスする場合は、プログラムでまずアクセスの対象となるセグメントのセグメント・セレクタをセグメント・レジスタにロードしなければならない。

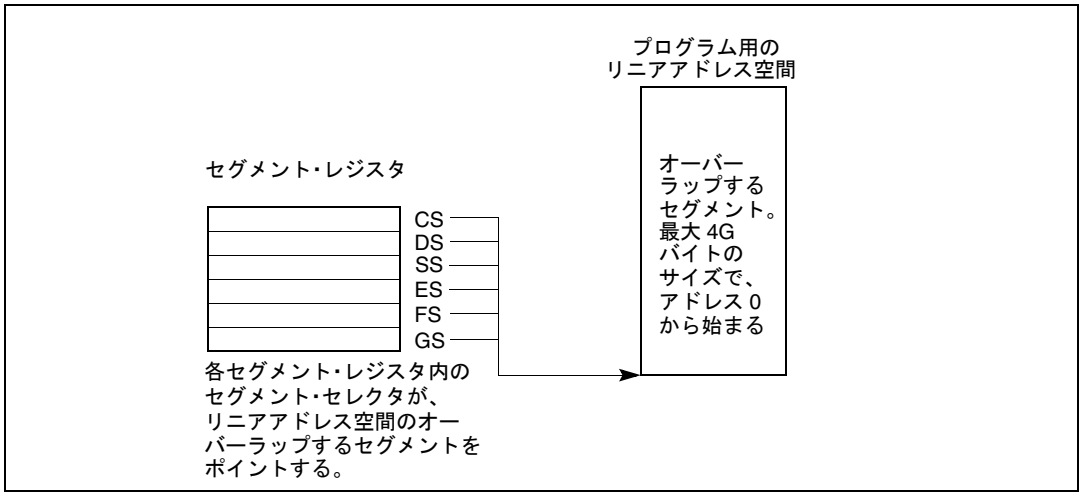


図 3-5. フラット・メモリ・モデルでのセグメント・レジスタの使用法

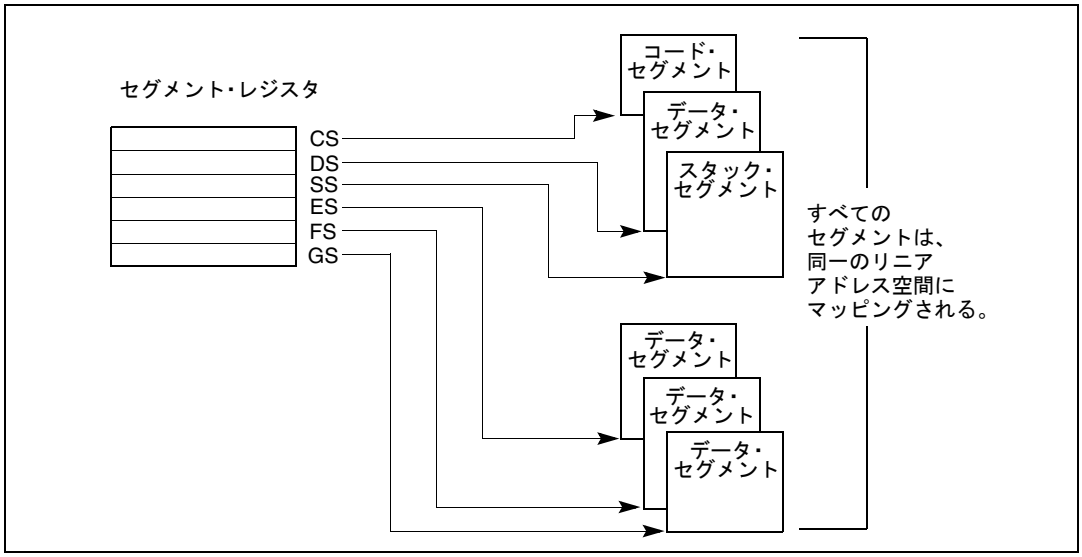


図 3-6. セグメント化メモリモデルでのセグメント・レジスタの使用法

セグメント・レジスタはそれぞれ、コード、データ、またはスタックのいずれかの記憶領域に対応付けられる。例えば、CS レジスタには、実行中の命令を格納する**コード・セグメント**用のセグメント・セレクタが格納される。プロセッサは、CS レジスタ内のセグメント・セレクタと EIP レジスタの内容で構成される論理アドレスを使用して、このコード・セグメントから命令をフェッチする。EIP レジスタには、次に実行される命令のコード・セグメント内のオフセットが格納される。CS レジスタには、アプリケーション・プログラムから明示的にロードできない。CS レジスタには、プログラム制御を変更する命令あるいは内部プロセッサ操作（プロシージャ・コール、割り込み処理、タスクスイッチなど）によって、暗黙的にロードされる。

DS、ES、FS、GS の各レジスタは、4 つの**データ・セグメント**をポイントする。4 つのデータ・セグメントを使用できることで、異なるタイプのデータ構造に効率よく、確実にアクセスできる。例えば、モジュールのデータ構造用に 1 つ、上位レベルにあるモジュールからエクスポートされるデータ用に 1 つ、動的に生成されるデータ構造用に 1 つ、他のプログラムと共用するデータ用に 1 つ、合計 4 つの独立したデータ・セグメントを作成できる。これ以外のデータ・セグメントにアクセスする場合は、アプリケーション・プログラムによって、それらのセグメント用のセグメント・セレクタを、必要に応じて DS、ES、FS、GS の各レジスタにロードしなければならない。

SS レジスタには、**スタック・セグメント**（現在実行中のプログラム、タスク、またはハンドラ用のプロシージャ・スタックを格納する）用のセグメント・セレクタが格納される。すべてのスタック操作において、SS レジスタを使用してスタック・セグメントを探し出す。CS レジスタとは異なり、SS レジスタには明示的にロードできるので、

アプリケーション・プログラム上でこのレジスタを介して複数のスタックをセットアップし、それらを交互に切り替えられる。

実アドレスモードでセグメント・レジスタを使用する方法については、3.3. 節「メモリの構成」を参照のこと。

CS、DS、SS、ES の 4 つのセグメント・レジスタは、インテル 8086 やインテル 286 プロセッサのセグメント・レジスタと同じである。また、FS と GS の両レジスタは、Intel386 ファミリに属するプロセッサから IA-32 アーキテクチャに導入された。

3.4.3. EFLAGS レジスタ

32 ビットの EFLAGS レジスタには、1 群のステータス・フラグ、1 つの制御フラグ、1 群のシステムフラグが格納される。このレジスタの各フラグの定義を、図 3-7. に示す。RESET ピンか INIT ピンをアサートしてプロセッサを初期化すると、EFLAGS レジスタのステートは 00000002H になる。このレジスタのビット 1、3、5、15、22 ～ 31 は予約済みであるため、ソフトウェア上でこれらのビットを使用したりビットのステートに依存することは避けなければならない。

EFLAGS レジスタのフラグの一部は、特殊な命令（次項以降で説明）を使用すれば、直接変更できる。レジスタ全体のチェックや変更を直接行う命令はない。LAHF、SAHF、PUSHF、PUSHFD、POPF、POPFD などの命令を使用すれば、プロシージャ・スタックあるいは EAX レジスタとの間でフラグのグループを移動できる。EFLAGS レジスタの内容をプロシージャ・スタックあるいは EAX レジスタに転送した後は、プロセッサのビット操作命令（BT、BTS、BTR、BTC）を使用してフラグのチェックや変更を実行できる。

プロセッサのマルチタスク機能を使用してタスクがサスペンドされる場合は、プロセッサは EFLAGS レジスタのステートを、サスペンドされるタスク用のタスク・ステート・セグメント（TSS）に自動的にセーブする。プロセッサは、自身を新規タスクに結合する際に、EFLAGS レジスタに、新規タスクの TSS からのデータをロードする。

割り込みまたは例外ハンドラ・プロシージャへのコールがかけられると、プロセッサは EFLAGS レジスタのステートを、プロシージャ・スタック上に自動的にセーブする。割り込みまたは例外がタスクスイッチで処理される場合は、EFLAGS レジスタのステートは、サスペンドされているタスク用の TSS にセーブされる。

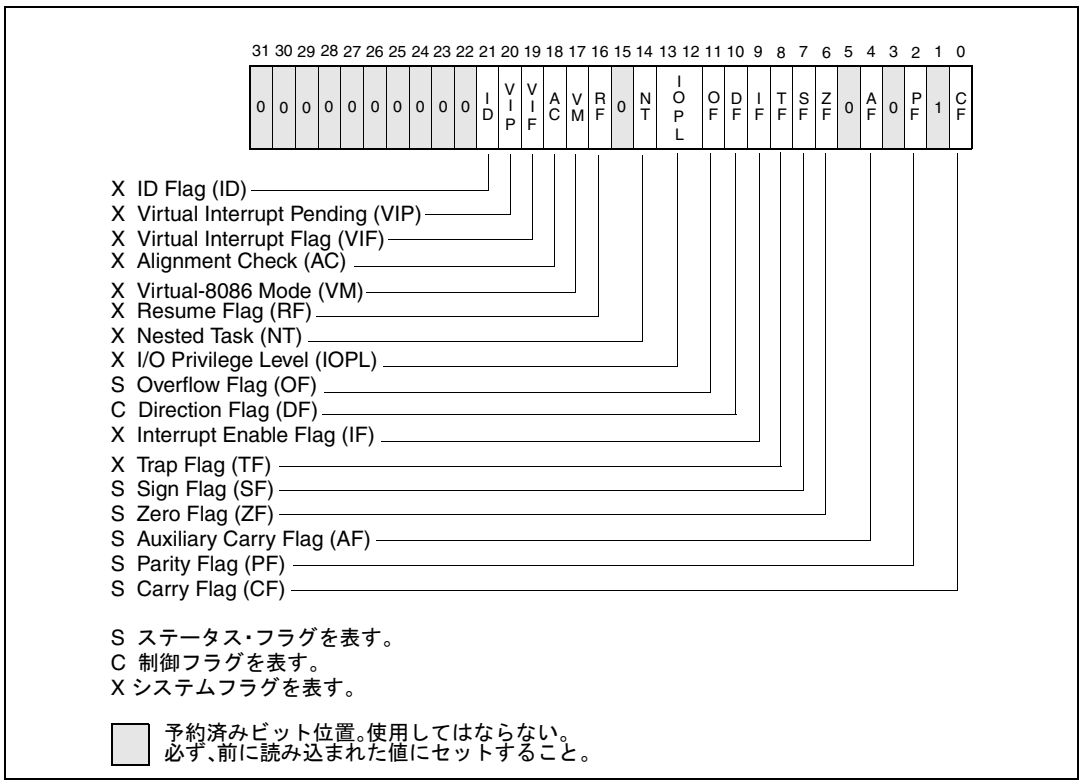


図 3-7. EFLAGS レジスタ

IA-32 アーキテクチャの進展にあわせて数々のフラグが EFLAGS レジスタに追加されてきたが、既存のフラグの機能や位置は、IA-32 プロセッサのどのファミリにおいても同じである。このため、コード上であるファミリに属する IA-32 プロセッサ向けにこれらのフラグにアクセスしたりフラグを変更するように書かれていても、それ以降のファミリに属するプロセッサ上でも問題なく動作する。

3.4.3.1. ステータス・フラグ

EFLAGS レジスタのステータス・フラグ（ビット 0、2、4、6、7、11）は、ADD、SUB、MUL、DIV などの算術命令の結果を示す。ステータス・フラグの機能を次に示す。

CF（ビット 0）　Carry Flag（キャリーフラグ）。算術演算では、結果の最上位ビットでキャリーまたはボローが生じた場合にセットされ、生じなかった場合にはクリアされる。このフラグは、符号なし整数演算でのオーバーフロー状態を示す。このフラグはまた、多倍精度演算においても使用される。

PF（ビット 2）　Parity flag（パリティフラグ）。結果の最下位バイトに値 1 のビットが偶数個含まれている場合にセットされ、奇数個の場合にはクリアされる。

AF（ビット 4） Adjust flag（調整フラグ）。算術演算では、結果のビット 3 にキャリーまたはボローが生じた場合にセットされ、生じなかった場合にはクリアされる。このフラグは、2 進化 10 進（BCD）演算で使用される。

ZF（ビット 6） Zero flag（ゼロフラグ）。結果がゼロの場合にセットされ、ゼロでない場合にクリアされる。

SF（ビット 7） Sign flag（符号フラグ）。符号付き整数の符号ビットである結果の最上位ビットと同じ値にセットされる。（0 は正の値を、1 は負の値を示す。）

OF（ビット 11） Overflow flag（オーバーフロー・フラグ）。整数の演算結果が大きすぎる正の数であるか、小さすぎる負の数で、デスティネーション・オペランドに収まらない場合（符号ビットは除く）にセットされ、そうでない場合にクリアされる。このフラグは、符号付き整数（2 の補数）演算でのオーバーフロー状態を示す。

これらのステータス・フラグのなかで、STC、CLC、CMC の各命令を使用して直接変更できるのは、CF フラグだけである。CF フラグに指定のビットをコピーするには、ビット命令（BT、BTS、BTR、BTC）を使用する。

ステータス・フラグを使用すれば、単一の算術演算で 3 つの異なるデータ型（符号なし整数、符号付き整数、BCD 整数）に対して結果を生成できる。算術演算の結果が符号なし整数として処理される場合は、CF フラグが範囲外状態（キャリーあるいはボロー）を示す。符号付き整数（2 の補数）として処理される場合は、OF フラグがキャリーあるいはボローを示す。BCD 数として処理される場合は、AF フラグがキャリーあるいはボローを示す。SF フラグは、符号付き整数の符号を示す。ZF フラグは、符号付き整数または符号なし整数でのゼロを示す。

整数に対して多倍精度演算を実行する場合は、CF フラグが ADC（Add with Carry）命令や SBB（Subtract with Borrow）命令と共に使用され、キャリーあるいはボローを計算間で伝達する。

J*cc*（Jump on Condition Code *cc*）、SET*cc*（Byte Set on Condition *cc*）、LOOP*cc*、CMOV*cc*（Conditional Move）などの条件付き命令では、1 つ、または複数のステータス・フラグを条件コードとして使用し、分岐、セットバイト、エンドループなどの条件をテストする。

3.4.3.2. DF フラグ

DF（Direction Flag（方向フラグ）、EFLAGS レジスタのビット 10 にある）は、ストリング命令（MOVS、CMPS、SCAS、LODS、および STOS）を制御する。DF フラグがセットされると、ストリング命令は自動的にデクリメントされる（ストリングを上位アドレスから下位アドレスに向かって処理する）。DF フラグがクリアされると、ストリング命令は自動的にインクリメントされる（ストリングを下位アドレスから上位アドレスに向かって処理する）。

DF フラグは、STD 命令を使ってセットし、CLD 命令を使ってクリアする。

3.4.4. システムフラグと IOPL フィールド

EFLAGS レジスタのシステムフラグと IOPL フィールドを使って、オペレーティング・システムやエグゼクティブの動作を制御する。**これらは、アプリケーション・プログラム上で変更してはならない。**これらのステータス・フラグは、それぞれ次の機能を持つ。

IF（ビット 9）　Interrupt Enable Flag（**割り込み可能フラグ**）。マスク可能な割り込みリクエストに対するプロセッサの応答を制御する。これがセットされると、プロセッサはマスク可能割り込みに応答する。クリアされると、マスク可能割り込みは無効になる。

TF（ビット 8）　Trap flag（**トラップフラグ**）。これがセットされると、デバッグにおいてシングル・ステップ・モードがイネーブルになる。クリアされると、シングル・ステップ・モードがディスエーブルになる。

IOPL（ビット 12 、13）I/O privilege level field（**I/O 特権レベル・フィールド**）。現在実行されているプログラムあるいはタスクの I/O 特権レベルを示す。現在実行されているプログラムあるいはタスクの現行特権レベル（CPL）は、I/O アドレス空間をアクセスするための I/O 特権レベルに等しいか、小さくなければならない。このフィールドは、CPL=0 で動作している場合に限り、POPF 命令と IRET 命令を使って変更できる。

NT（ビット 14）　Nested Task Flag（**ネスト・タスク・フラグ**）。割り込まれたタスクやコールされたタスクのチェーン結合を制御する。現在のタスクが直前に実行されたタスクにリンクされている場合にセットされ、現在のタスクが別のタスクにリンクされていない場合はクリアされる。

RF（ビット 16）　Resume flag（**再開フラグ**）。デバッグ例外に対するプロセッサの応答を制御する。

VM（ビット 17）　Virtual-8086 mode flag（**仮想 8086 モードフラグ**）。これがセットされると、仮想 8086 モードがイネーブルになる。クリアされると、仮想 8086 モードのセマンティクスなしで保護モードに戻る。

AC（ビット 18）　Alignment check flag（**アライメント・チェック・フラグ**）。このフラグと CR0 レジスタの AM ビットがセットされると、メモリ参照においてアライメント・チェックがイネーブルになる。このフラグと AM ビットのいずれかまたは両方がクリアされると、アライメント・チェックはディスエーブルになる。

VIF（ビット 19）　Virtual interrupt flag（**仮想割り込みフラグ**）。IF フラグの仮想イメージ。VIP フラグと共に使用する。（このフラグと VIP フラグを使用するには、制御レジスタ CR4 の VME フラグをセットすることにより、仮想モード拡張をイネーブルにしなければならない。）

VIP（ビット 20）　Virtual interrupt pending flag（**仮想割り込み保留フラグ**）。ペンディング状態の割り込みがある場合にセット、またはペンディング状態の

割り込みがない場合にクリアされる。（このフラグは、ソフトウェアによってセットしクリアする。プロセッサは、読み取るだけである。）このフラグは、VIF フラグと共に使用される。

ID（ビット 21）　Identification flag（識別フラグ）。プログラムがこのフラグをセットするかクリアできれば、CPUID 命令がサポートされることを表す。

こられのフラグの詳細な説明については、『IA-32 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、下巻』の第 3 章「保護モードにおけるメモリ・マネージメント」を参照のこと。

3.5. 命令ポインタ

命令ポインタ（EIP）レジスタは、（現在のコード・セグメント内にある）次に実行される命令用のオフセットを格納する。このオフセットは、直線的コードでは 1 つの命令境界から次の命令境界へと順番に進められる。一方、JMP、J*cc*、CALL、RET、IRET などの命令の実行時には、多くの命令分前方または後方に進められる。

EIP レジスタに対しては、ソフトウェアから直接アクセスはできない。このレジスタは、制御転送命令（JMP、J*cc*、CALL、RET など）、割り込み、例外などによって暗黙的に制御される。EIP レジスタを読み取る唯一の方法として、まず CALL 命令を実行し、次にプロシージャ・スタックからリターン命令ポインタの値を読み取る。また、プロシージャ・スタック上のリターン命令ポインタの値を変更し、リターン命令（RET あるいは IRET）を実行すれば、EIP レジスタに間接的にロードできる。6.2.4.2. 項「リターン命令ポインタ」を参照のこと。

すべての IA-32 プロセッサは、命令をプリフェッチする。この命令のプリフェッチにより、命令のロード時にバスから読み取られた命令アドレスは EIP レジスタ内の値とは一致しないことになる。プロセッサの世代が異なればプリフェッチの機構も異なるが、プログラム・フローを指示するという EIP レジスタの機能は、IA-32 プロセッサ上で動作するように開発されたソフトウェアすべてと完全な互換性がある。

3.6. オペランド・サイズ属性とアドレスサイズ属性

プロセッサが保護モードで動作している場合、すべてのコード・セグメントはデフォルトのオペランド・サイズ属性とアドレスサイズ属性を持っている。これらの属性を選択するには、コード・セグメントに対するセグメント・ディスクリプタ内のD（デフォルト・サイズ）フラグを使用する（『IA-32 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、下巻』の第3章「保護モードにおけるメモリ・マネージメント」を参照）。Dフラグがセットされているときには、32ビットのオペランド・サイズ属性とアドレスサイズ属性が選択される。フラグがクリアされているときには、16ビットのオペランド・サイズ属性とアドレスサイズ属性が選択される。プロセッサが、実アドレスモード、仮想8086モード、またはSMMで動作している場合は、デフォルトのオペランド・サイズ属性とアドレスサイズ属性は常に16ビットになる。

オペランド・サイズ属性によって、オペランドのサイズを選択する。16ビットのオペランド・サイズ属性が有効になっている場合は、オペランドは一般的には8ビットか16ビットのいずれかである。32ビットのオペランド・サイズ属性が有効になっている場合は、オペランドは一般的には8ビットか32ビットのいずれかである。

アドレスサイズ属性によって、メモリをアドレス指定する際に使用されるアドレスのサイズ（16ビットか32ビット）を選択する。16ビットのアドレスサイズ属性が有効になっている場合は、セグメント・オフセットとディスプレースメントは16ビットになる。このため、セグメントのサイズは64Kバイトまでに制限される。一方、32ビットのアドレスサイズ属性が有効になっている場合は、セグメント・オフセットとディスプレースメントは32ビットになり、4Gバイトまでのアドレス指定が可能になる。

特定の命令に対するデフォルトのオペランド・サイズ属性やアドレスサイズ属性は、命令にオペランド・サイズ・プリフィックスやアドレス・サイズ・プリフィックスを追加すると、オーバーライドできる。『IA-32 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、下巻』の第2章にある「命令プリフィックス」を参照のこと。このプリフィックスの効果は、対象の命令に対してのみ有効になる。

表3-1.に、Dフラグ、オペランド・サイズ・プリフィックス、アドレス・サイズ・プリフィックスの各設定によって得られる有効なオペランド・サイズとアドレスサイズを示す（保護モードで動作する場合）。

表 3-1. 有効なオペランド・サイズ属性とアドレスサイズ属性

コード・セグメント・ディスクリプタ内の D フラグ	0	0	0	0	1	1	1	1
オペランド・サイズ・プリフィックス 66H	×	×	○	○	×	×	○	○
アドレス・サイズ・プリフィックス 67H	×	○	×	○	×	○	×	○
有効なオペランド・サイズ	16	16	32	32	32	32	16	16
有効なアドレスサイズ	16	32	16	32	32	16	32	16

注：

○：この命令プリフィックスはある。

×：この命令プリフィックスはない。

3.7. オペランドのアドレス指定

IA-32 のマシン語命令には、オペランドがないもの、オペランドが 1 つのもの、複数のオペランドをとるものがある。オペランドには、明示的に指定するものと、暗黙的に決まるものがある。ソース・オペランドのデータは、次のいずれかに配置できる。

- 命令自体（即値オペランド）
- レジスタ
- メモリ・ロケーション
- I/O ポート

命令がデータをデスティネーション・オペランドに返す場合、以下のいずれかに返すことができる。

- レジスタ
- メモリ・ロケーション
- I/O ポート

3.7.1. 即値オペランド

一部の命令では、命令そのものにエンコーディングされているデータをソース・オペランドとして使用する。これらのオペランドを即値オペランド（または、単に即値）と呼ぶ。例えば、次の ADD 命令では、即値 14 を EAX レジスタの内容に加算する。

```
ADD EAX, 14
```

（DIV 命令と IDIV 命令を除く）すべての算術命令では、ソース・オペランドとして即値を使用できる。即値オペランドとして許可される最大値は命令によって異なるが、符号なしダブルワード整数の最大値（2^{32}）を超えることはできない。

3.7.2. レジスタ・オペランド

ソース・オペランドとデスティネーション・オペランドは、実行される命令に応じて、次に挙げるレジスタのいずれかになる。

- 32 ビット汎用レジスタ（EAX、EBX、ECX、EDX、ESI、EDI、ESP、または EBP）
- 16 ビット汎用レジスタ（AX、BX、CX、DX、SI、DI、SP、または BP）
- 8 ビット汎用レジスタ（AH、BH、CH、DH、AL、BL、CL、または DL）
- セグメント・レジスタ（CS、DS、SS、ES、FS、GS）
- EFLAGS レジスタ
- x87 FPU レジスタ（ST0 ～ ST7、ステータス・ワード、制御ワード、タグワード、データ・オペランド・ポインタ、命令ポインタ）
- MMX レジスタ（MM0 ～ MM7）
- XMM レジスタ（XMM0 ～ XXM7）、MXCSR レジスタ
- コントロール・レジスタ（CR0、CR2、CR3、CR4）、システム・テーブル・ポインタ・レジスタ（GDTR、LDTR、IDTR、タスクレジスタ）
- デバッグレジスタ（DR0、DR1、DR2、DR3、DR6、DR7）
- MSR レジスタ

ある命令（DIV 命令や MUL 命令など）は、1 対の 32 ビット・レジスタに格納されるクワッドワード・オペランドを使用する。レジスタの対は、コロンで区切って表す。例えば、レジスタペア EDX:EAX では、クワッドワード・オペランドの上位ビットが EDX に、下位ビットが EAX に格納される。

また、EFLAGS レジスタの内容をロードあるいはストアしたり、EFLAGS レジスタの個々のフラグをセットあるいはクリアするための命令（PUSHFD 命令や POPFD 命令など）が用意されている。命令によっては、EFLAGS レジスタ内のステータス・フラグのステートを、分岐などの結果判定操作を行う際の条件コードとして使用するものもある（J*cc* 命令など）。

プロセッサには、メモリ管理、割り込みや例外の処理、タスク管理、プロセッサ管理、デバッグ操作などを制御するための一連のシステムレジスタがある。これらのシステムレジスタのなかには、システム命令を使用すれば、アプリケーション・プログラム、オペレーティング・システム、あるいはエグゼクティブからアクセスできるものもあ

る。システム命令でシステムレジスタにアクセスする場合は、通常は、レジスタがその命令の暗黙のオペランドになる。

3.7.3. メモリ・オペランド

メモリ内のソース・オペランドとデスティネーション・オペランドは、セグメント・セレクタとオフセットによって参照される（図 3-8. を参照）。セグメント・セレクタで、オペランドが格納されているセグメントを指定する。オフセット（セグメントの先頭からオペランドの最初のバイトまでのバイト数）で、オペランドのリニアアドレスもしくは実効アドレスを指定する。

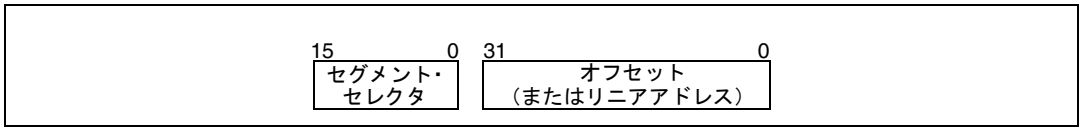


図 3-8. メモリ・オペランドのアドレス

3.7.3.1. セグメント・セレクタの指定

セグメント・セレクタは、暗黙的にも明示的にも指定できる。セグメント・セレクタを指定する最も一般的な方法は、セグメント・セレクタをセグメント・レジスタにまずロードし、実行しようとしている操作の種類に応じて、プロセッサにレジスタを暗黙的に選択させるものである。プロセッサは、表 3-2. に示す規則にしたがって、セグメントを自動的に選択する。

メモリに対してデータをストアあるいはロードするときは、デフォルトの DS セグメントをオーバーライドして他のセグメントにアクセスできる。アセンブラでは、セグメントのオーバーライドは一般的にコロン（:）演算子で処理される。例えば、次の MOV 命令では、EAX レジスタから、ES レジスタがポイントするセグメントに値を移動する。そのセグメントに対するオフセットは、EBX レジスタに格納されている。

```
MOV ES:[EBX], EAX;
```

表 3-2. デフォルトのセグメント選択規則

参照のタイプ	使用される レジスタ	使用される セグメント	デフォルトの選択規則
命令	CS	コード・セグメント	すべての命令フェッチ
スタック	SS	スタック・セグメント	すべてのスタックのプッシュとポップ。ベースレジスタとして ESP あるいは EBP レジスタを使用するすべてのメモリ参照。
ローカル データ	DS	データ・セグメント	すべてのデータ参照。スタックに関連する場合や、ストリング・デスティネーションを除く。
デスティネーション・ストリング	ES	ES レジスタでポイントされるデータ・セグメント	ストリング命令のデスティネーション

マシンレベルでは、セグメントのオーバーライドはセグメント・オーバーライド・プリフィックスで指定する。このプリフィックスは 1 バイトで、命令の先頭に置く。ただし、次のデフォルトのセグメント選択はオーバーライドできない。

- 命令フェッチは、コード・セグメントから実行しなければならない。
- ストリング命令内のデスティネーション・ストリングは、ES レジスタがポイントするデータ・セグメント内に格納しなければならない。
- プッシュ操作とポップ操作では、常に SS セグメントを参照しなければならない。

命令によっては、セグメント・セレクタを明示的に指定しなければならないものがある。このような場合は、16 ビットのセグメント・セレクタをメモリ・ロケーションか 16 ビット・レジスタ内に配置できる。例えば、次の MOV 命令では、レジスタ BX に配置されたセグメント・セレクタをセグメント・レジスタ DS に転送する。

```
MOV DS, BX
```

セグメント・セレクタは、メモリ内の 48 ビットの far ポインタの一部として明示的に指定することもできる。この場合、メモリ内の最初のダブルワードにオフセットが入り、次のワードにセグメント・セレクタが入る。

3.7.3.2. オフセットの指定

メモリアドレスのオフセット部分は、スタティックな値（**ディスプレースメント**と呼ばれる）として直接に指定するか、あるいは次に挙げる要素の 1 つ以上で構成されるアドレス計算によって指定する。

- ディスプレースメント － 8 ビット、16 ビット、または 32 ビットの値
- ベース － 汎用レジスタの値
- インデックス － 汎用レジスタの値
- スケール係数 － 2、4、または 8 の値。これとインデックス値を掛け合わせる

これらの要素を組み合わせて得られるオフセットは、**実効アドレス**と呼ばれる。これらの要素それぞれは、スケール係数を除き、正の値か負（2 の補数）の値をとることができる。図 3-9. に、これらの要素を組み合わせて、選択したセグメント内の実効アドレスを算出する方法をすべて示す。

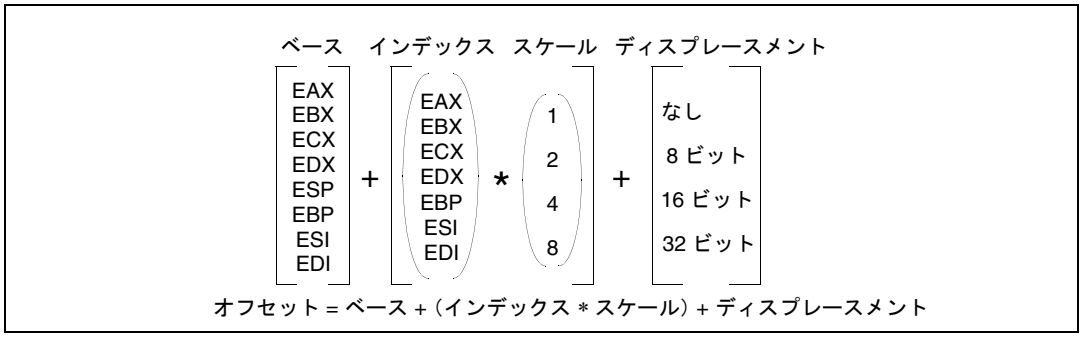


図 3-9. オフセット（または実効アドレス）の計算

汎用レジスタをベースやインデックスの要素として使用する場合は、次の制限に従わなければならない。

- ESP レジスタは、インデックス・レジスタとして使用できない。
- ESP レジスタあるいは EBP レジスタをベースとして使用する場合は、SS セグメントがデフォルト・セグメントになる。これ以外の場合は、DS セグメントがデフォルト・セグメントになる。

ベース、インデックス、ディスプレースメントの各要素は任意の組み合わせで使用でき、また、これらのうちのどれがヌルであってもよい。スケール係数は、インデックスを使用する場合にのみ使用する。いずれの組み合わせも、プログラマが高級言語やアセンブリ言語において一般的に使用するデータ構造に対して使用できる。

次の各項では、アドレス要素の一般的な組み合わせによるアドレス指定モードを示す。

ディスプレースメント

ディスプレースメントは、単体ではオペランドに対する直接的な（すなわち計算されない）オフセットを表す。ディスプレースメントは命令内にエンコーディングされるため、この形式のアドレスを絶対アドレスもしくは静的アドレスと呼ぶことがある。ディスプレースメントは、一般的に、静的に割り当てられたスカラ・オペランドにアクセスする場合に使用される。

ベース

ベースは、単体ではオペランドに対する間接的なオフセットを表す。ベースレジスタ内の値は変更が可能なため、ベースは変数やデータ構造の動的記憶領域に使用される。

ベース + ディスプレースメント

ベースレジスタとディスプレースメントの組み合わせは、次の 2 つの目的に使用できる。

- 要素サイズが 2 バイト、4 バイト、8 バイトでない場合に、配列に対するインデックスとして使用する。ディスプレースメント要素は、配列の先頭までの静的オフセットをエンコーディングする。ベースレジスタには、配列内の特定の要素までのオフセットを決めるための計算結果が入る。
- レコードのフィールドにアクセスするために使用する。ベースレジスタには、レコードの先頭のアドレスが入り、ディスプレースメントにはフィールドまでの静的オフセットが入る。

この組み合わせの特殊かつ重要なケースの 1 つに、プロシージャ起動レコード内にあるパラメータへのアクセスを挙げることができる。プロシージャ起動レコードは、プロシージャに移行した時点で作成されるスタックフレームである。このケースでは、ベースレジスタとして EBP レジスタを選択するのが最適である。これは、EBP レジスタがスタック・セグメントを自動的に選択するためである。この方法により、この一般的な機能をコンパクトなエンコーディングで実現できる。

（インデックス * スケール）+ ディスプレースメント

このアドレスモードは、要素のサイズが 2 バイト、4 バイト、または 8 バイトの場合に、静的配列を効率よくインデックスできる。ディスプレースメントには配列の先頭が入り、インデックス・レジスタには必要な配列要素の添字が入る。プロセッサは、スケール係数を適用させて、この添字をインデックスに自動的に変換する。

ベース + インデックス + ディスプレースメント

2 つのレジスタを一緒に使用すると、2 次元配列（ディスプレースメントには、配列の先頭のアドレスが入る）か、レコード配列のインスタンスの 1 つをサポートできる（ディスプレースメントには、レコード内の対象フィールドまでのオフセットが入る）。

ベース + （インデックス * スケール） + ディスプレースメント

すべてのアドレス指定要素を組み合わせて使用すると、配列要素のサイズが 2 バイト、4 バイト、または 8 バイトのいずれかの場合に、2 次元配列を効率よくインデックスできる。

3.7.3.3. アセンブラとコンパイラのアドレス指定モード

マシン・コード・レベルでは、ディスプレースメント、ベースレジスタ、インデックス・レジスタ、スケール係数のなかから選択した組み合わせが命令のなかにエンコーディングされる。いずれのアセンブラにおいても、プログラマがこれらのアドレス指定要素を任意に組み合わせてオペランドをアドレス指定できる。高級言語（HLL − High Level Language）のコンパイラでは、プログラマが定義した HLL 構造をベースに、これらの要素を適当に組み合わせて選択される。

3.7.4. I/O ポートのアドレス指定

プロセッサは、最大 65,536 個の 8 ビット I/O ポートが格納された I/O アドレス空間をサポートする。I/O アドレス空間には、16 ビットや 32 ビットのポートも定義できる。I/O ポートは、即値オペランドまたは DX レジスタ内の値を使用してアドレス指定できる。I/O ポートのアドレス指定の詳細については、第 13 章「入出力」を参照のこと。

4

データ型

第 4 章
データ型

4

本章では、IA-32 アーキテクチャのデータ型の定義について説明する。本章の最後の節では、x87 FPU、SSE、SSE2、SSE3 で用いられる実数および浮動小数点の概念について説明する。

4.1. 基本データ型

IA-32 アーキテクチャの基本データ型は、バイト、ワード、ダブルワード、クワッドワード、ダブル・クワッドワードである（図 4-1. 参照）。1 バイトは 8 ビット、1 ワードは 2 バイト（16 ビット）、1 ダブルワードは 4 バイト（32 ビット）、1 クワッドワードは 8 バイト（64 ビット）、1 ダブル・クワッドワードは 16 バイト（128 ビット）にそれぞれ相当する。IA-32 アーキテクチャ命令の一部は、追加のオペランド・タイプの指定なしに、これらの基本データ型を操作する。

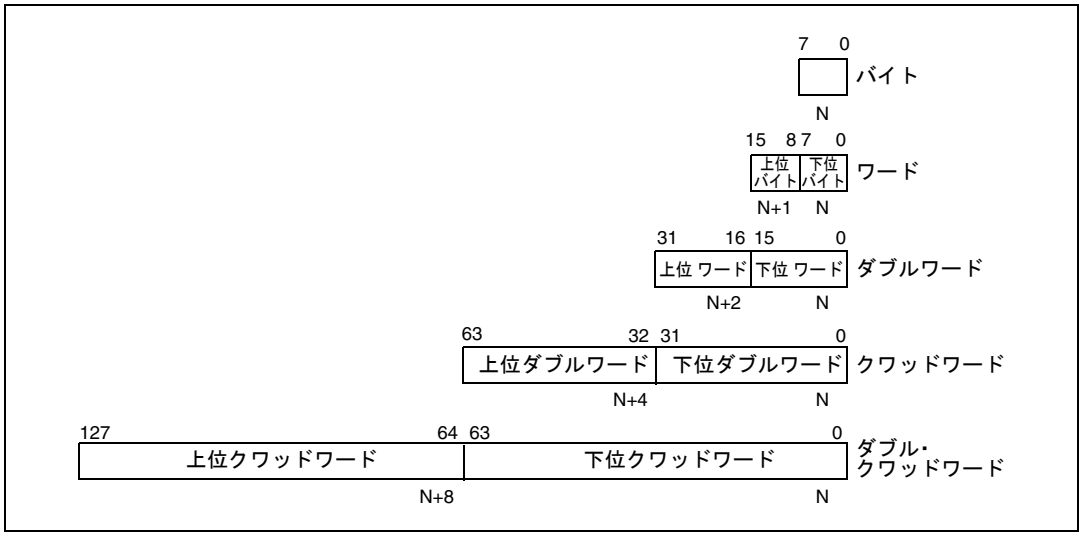


図 4-1. 基本データ型

クワッドワード・データ型は、Intel486™プロセッサでIA-32アーキテクチャに導入された。ダブル・クワッドワード・データ型は、SSEと共に、インテル® Pentium® IIIプロセッサで導入された。

図4-2.に、メモリ内のオペランドとして参照される場合の基本データ型それぞれのバイト配列を示す。それぞれのデータ型の下位バイト（ビット0～7）が、メモリ内の最下位アドレスを占有し、そのアドレスがオペランドのアドレスになる。

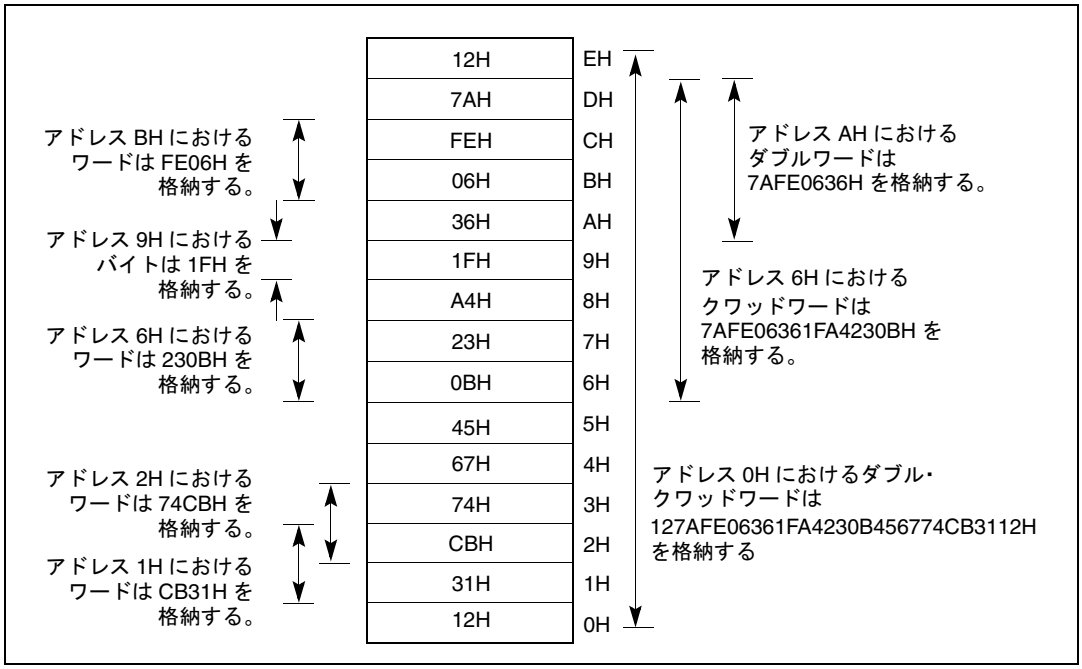


図 4-2. メモリ内のバイト、ワード、ダブルワード、およびクワッドワード、
およびダブル・クワッドワード

4.1.1. ワード、ダブルワード、クワッドワード、ダブル・クワッドワードのアライメント

ワード、ダブルワード、クワッドワードは、メモリ内では自然境界にアライメントを合わせる必要はない。（ワード、ダブルワード、クワッドワードの自然境界はそれぞれ、偶数のアドレス、4 で割り切れるアドレス、8 で割り切れるアドレスになる。）ただし、プログラムの処理能力を向上させるためには、データ構造（特にスタック）においては可能な限りこれらの自然境界にアライメントを合わせなければならない。その理由は、アライメントが合っていないメモリにアクセスを 1 回行おうとすると、プロセッサは実際には 2 回のメモリアクセスを行う必要があるが、アライメントが合っているメモリにアクセスする場合は、1 回のメモリアクセスで済む。4 バイト境界にまたがるワード・オペランドとダブルワード・オペランド、あるいは 8 バイト境界にまたがるクワッドワード・オペランドは、アライメントが合っていないものと見なされ、アクセスには 2 回の別個のメモリ・バス・サイクルが必要になる。

ダブル・クワッドワードを操作する命令のいくつかを実行する場合は、メモリ・オペランドのアライメントが自然境界に合っていなければならない。アライメントが合っていないオペランドを指定した場合、これらの命令を実行すると、一般保護例外（#GP）が発生する。ダブル・クワッドワードの自然境界とは、16 で割り切れる任意のアドレスである。ダブル・クワッドワードを操作する命令の中には、（一般保護例外を発生することなく）アライメントの合っていないオペランドにアクセスできるものもあるが、メモリ内のアライメントの合っていないデータにアクセスすると、追加のメモリ・バス・サイクルが発生する。

4.2. 数値のデータ型

バイト、ワード、ダブルワードは、IA-32 アーキテクチャの基本データ型であるが、命令によっては、数値データ型（符号付き整数や符号なし整数、浮動小数点数など）に対する演算を可能にするため、これら以外のデータ型をサポートするものもある（図 4-3. を参照）。

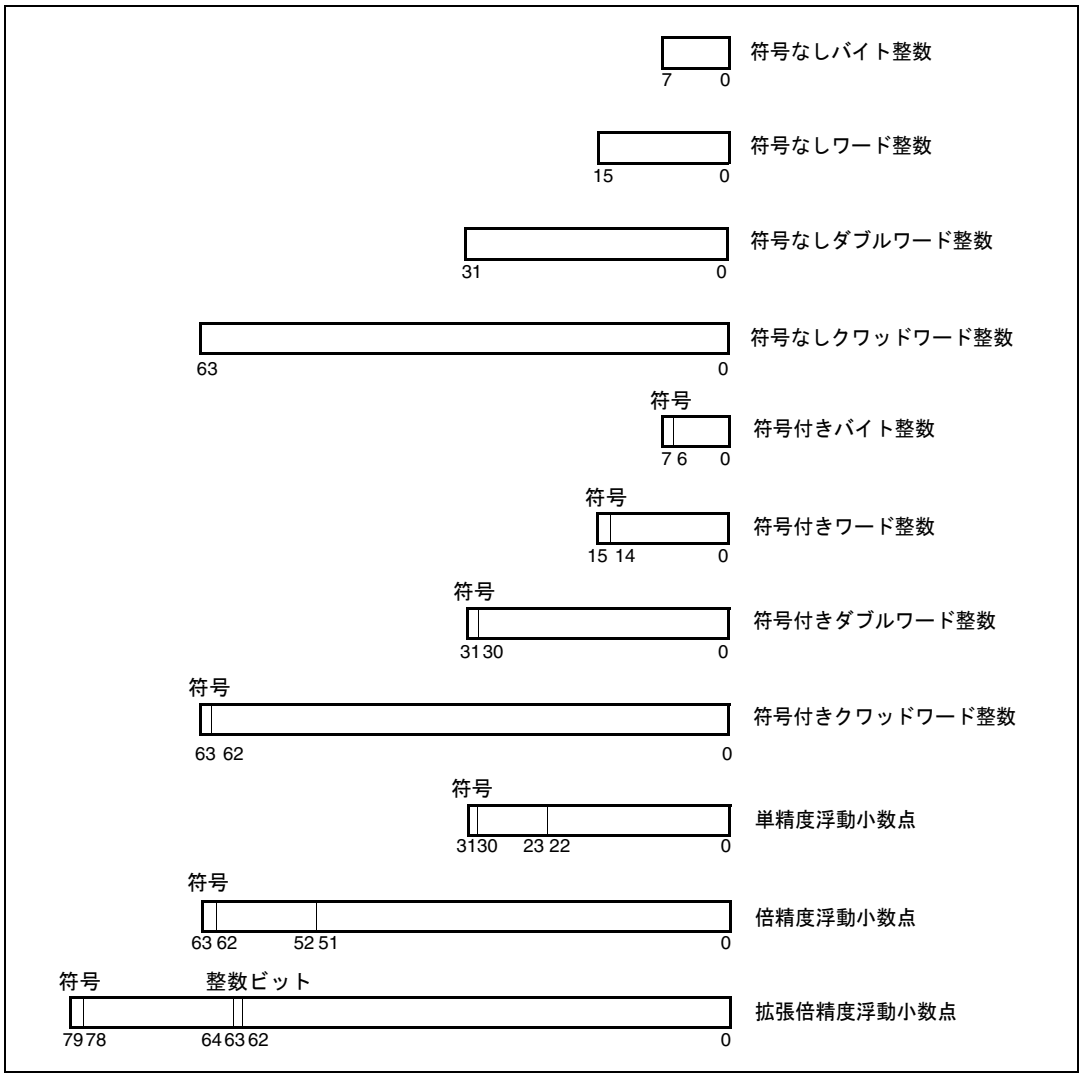


図 4-3. 数値のデータ型

4.2.1. 整数

IA-32アーキテクチャは、2種類の整数（符号なし整数と符号付き整数）を定義している。符号なし整数は、0～正の最大数の範囲の通常の2進値で、選択したオペランド・サイズでエンコードできる。符号付き整数は、正と負の両方の整数値を表現できる、2の補数の2進値である。

一部の整数命令（ADD、SUB、PADDB、PSUBB命令など）は、符号なし整数と符号付き整数のオペランドを操作できる。その他の整数命令（IMUL、MUL、IDIV、DIV、FIADD、FISUBなど）は、いずれかのタイプの整数だけを操作する。

以下の各節では、2種類の整数のエンコーディングと範囲について説明する。

4.2.1.1. 符号なし整数

符号なし整数は、1バイト、1ワード、1ダブルワード、またはクワッドワードに格納される符号なし2進数である。符号なし整数値の範囲は、符号なしバイト整数では0～255、符号なしワード整数では0～65,535、符号なしダブルワード整数では0～2^{32}-1、符号なしクワッドワード整数では0～2^{64}-1になる。符号なし整数は、**序数**とも呼ばれる。

4.2.1.2. 符号付き整数

符号付き整数は、1バイト、1ワード、1ダブルワード、または1クワッドワードに格納される符号付き2進数である。符号付き整数のすべての演算において、2の補数表現が使用されているものと見なされる。符号ビットは、バイト整数ではビット7に、ワード整数ではビット15に、ダブルワード整数ではビット31、クワッドワード整数ではビット63に配置される（符号付き整数のエンコーディングについては表4-1.を参照）。

表 4-1. 符号付き整数のエンコーディング

クラス		2 の補数のエンコーディング	
		符号	
正	最大	0	11..11
		.	.
		.	.
	最小	0	00..01
0		0	00..00
負	最小	1	11..11
		.	.
		.	.
	最大	1	00..00
整数不定値		1	00..00
		符号付きバイト整数 符号付きワード整数 符号付きダブルワード整数 符号付きクワッドワード整数	← 7 ビット → ← 15 ビット → ← 31 ビット → ← 63 ビット →

符号ビットは、負の整数に対してセットされ、正の整数とゼロに対してはクリアされる。整数値の範囲は、バイト整数では -128 ～ +127、ワード整数では -32,768 ～ +32,767、ダブルワード整数では -2^{31} ～ $+2^{31}-1$、クワッドワード整数では -2^{63} ～ $+2^{63}$ になる。

整数値をメモリに格納するとき、ワード整数は連続する 2 バイトに格納され、ダブルワード整数は連続する 4 バイトに格納され、クワッドワード整数は連続する 8 バイトに格納される。

整数不定値は、x87 FPU が整数値を操作するときに戻すことがある、特殊な値である。詳細は 8.2.1. 項「不定値」を参照のこと。

4.2.2. 浮動小数点データ型

IA-32 アーキテクチャは、単精度浮動小数点、倍精度浮動小数点、拡張倍精度浮動小数点の 3 つの浮動小数点データ型を定義しており、これらのデータ型を操作する（図 4-3. を参照）。これらのデータ型のデータ・フォーマットは、2 進浮動小数点演算に関する IEEE 規格 754 で指定されたフォーマットに直接対応する。

表 4-2. は、それぞれの浮動小数点データ型で表現できる、長さ、精度、および近似的な正規化範囲を示している。これらのデータ型では、デノーマル値もサポートされる。

表 4-2. 浮動小数点データ型の長さ、精度、および範囲

データ型	長さ	精度（ビット）	ノーマル型の近似範囲	
			2 進	0 進
単精度	32	24	$2^{-126} \sim 2^{127}$	$1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$
倍精度	64	53	$2^{-1022} \sim 2^{1023}$	$2.23 \times 10^{-308} \sim 1.79 \times 10^{308}$
拡張倍精度	80	64	$2^{-16382} \sim 2^{16383}$	$3.37 \times 10^{-4932} \sim 1.18 \times 10^{4932}$

注記

4.8. 節「実数フォーマットと浮動小数点フォーマット」では、IEEE 規格 754 の浮動小数点フォーマットの概要を説明し、整数ビット、QNaN、SNaN、デノーマル値などの用語を定義する。

表 4-3. は、3 つの浮動小数点データ型について、0、デノーマル型有限数、ノーマル型有限数、無限大、NaN の浮動小数点エンコーディングを示している。また、QNaN の浮動小数点不定値のフォーマットを示す。(QNaN の浮動小数点不定値の使い方の説明は、4.8.3.7. 項「QNAN 浮動小数点不定数」を参照のこと。)

単精度フォーマットと倍精度フォーマットでは、仮数部の小数部分だけがコード化される。整数部分は、0 とデノーマル型有限数を除き、すべて 1 と見なされる。拡張倍精度フォーマットでは、整数部分がビット 63 に、小数部分の最上位ビットがビット 62 に格納される。この場合、整数部分はノーマル型数、無限大、NaN に対しては明示的に 1 に設定され、ゼロおよびデノーマル型数については 0 に設定される。

表 4-3. 浮動小数点と NaN のエンコーディング

クラス		符号	バイアス付き指数	仮部数	
				整数部分 [1]	小数部分
正	+∞	0	11..11	1	00..00
	+ ノーマル	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+ デノーマル	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+ ゼロ	0	00..00	0	00..00
負	– ゼロ	1	00..00	0	00..00
	– デノーマル	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	– ノーマル	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	– ∞	1	11..11	1	00..00
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	QNaN 浮動小数点不定数	1	11..11	1	10..00
	単精度 : 倍精度 : 拡張倍精度 :		← 8 ビット → ← 11 ビット → ← 15 ビット →		← 23 ビット → ← 52 ビット → ← 63 ビット →

注 :

1. 整数ビットは暗黙であり、単精度や倍精度フォーマットでは格納されない。

2. SNaN のエンコーディングの小数部分は、ゼロでない値で、最上位ビットが 0 でなければならない。

それぞれの浮動小数点データ型の指数は、バイアス付きフォーマットでコード化される。4.8.2.2. 項「バイアス付き指数」を参照のこと。バイアス定数は、単精度フォーマットでは 127、倍精度フォーマットでは 1023、拡張倍精度フォーマットでは 16,383 になる。

浮動小数点値をメモリに格納する場合は、単精度値はメモリ内の連続する 4 バイトに、倍精度値は連続する 8 バイトに、拡張倍精度値は連続する 10 バイトにそれぞれ格納される。

単精度および倍精度浮動小数点データ型は、x87 FPU 命令、SSE、SSE2、SS3 によって操作される。拡張倍精度浮動小数点フォーマットは、x87 FPU だけが操作できる。x87 FPU 命令と、SSE、SSE2、SSE3 の間の、単精度および倍精度浮動小数点データ型の互換性については、11.6.8. 項「SIMD 浮動小数点データ型と x87 FPU 浮動小数点データ型の互換性」を参照のこと。

4.3. ポインタデータ型

ポインタは、メモリ内のロケーションに対するアドレスである（図 4-4. 参照）。IA-32 アーキテクチャは、near **ポインタ**（32 ビット）と far **ポインタ**（48 ビット）の 2 種類のポインタを定義する。near ポインタは、セグメント内の 32 ビット・オフセット（**実効アドレス**とも呼ばれる）である。near ポインタは、フラット・メモリ・モデルではすべてのメモリを参照するのに使用される。また、セグメント化モデルでの参照にも使用されるが、アクセスの対象となるセグメントは暗黙的に決まる。これに対し、far ポインタは、16 ビットのセグメント・セレクタと 32 ビットのオフセットからなる 48 ビットの論理アドレスである。far ポインタをセグメント化メモリモデルでのメモリ参照に使用する場合には、アクセスの対象となるセグメントは明示的に指定しなければならない。

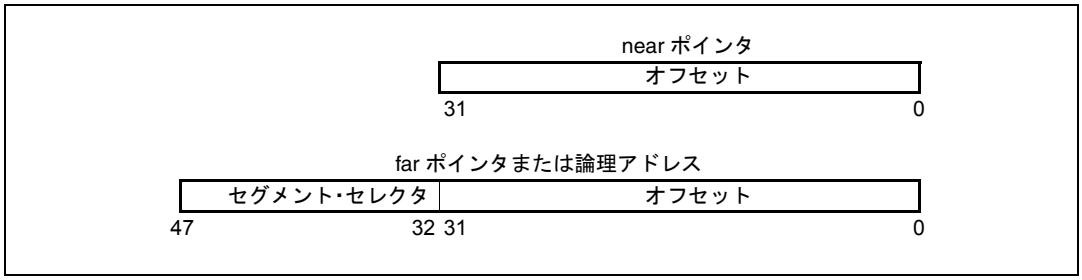


図 4-4. ポインタデータ型

4.4. ビット・フィールド・データ型

ビット・フィールド（図 4-5. 参照）は、連続するビット・シーケンスである。ビット・フィールドは、メモリ内にある任意のバイトの任意のビット位置から開始でき、また最大 32 ビットを格納できる。

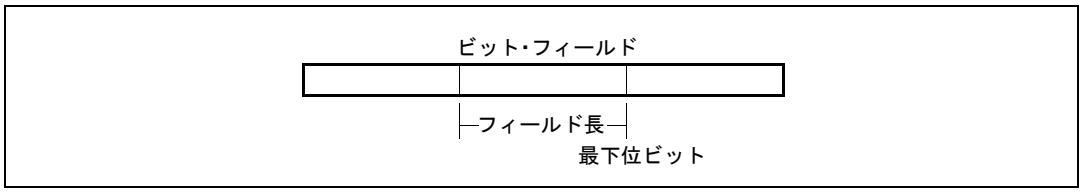


図 4-5. ビット・フィールド・データ型

4.5. ストリング・データ型

ストリングは、連続するビット、バイト、ワード、またはダブルワードのシーケンスである。**ビット・ストリング**は、任意のバイトの任意のビット位置から開始でき、また最大 2^{32}-1 ビットを格納できる。**バイト・ストリング**は、バイト、ワード、ダブルワードを格納でき、また 0 ～ 2^{32}-1 バイト（4G バイト）の範囲である。

4.6. パックド SIMD データ型

IA-32 アーキテクチャは、SIMD 演算に使用される、一連の 64 ビットおよび 128 ビット・パックド・データ型を定義しており、それらのデータ型を操作する。これらのデータ型は、基本データ型（パックドバイト、パックドワード、パックド・ダブルワード、パックド・クワッドワード）と基本データ型の数値表現で構成され、パックド整数演算およびパックド浮動小数点演算に使用される。

4.6.1. 64 ビット・パックド SIMD データ型

64 ビット・パックド SIMD データ型は、インテル MMX テクノロジで IA-32 アーキテクチャに導入された。これらのデータ型は、MMX テクノロジ・レジスタ内で操作される。64 ビット・パックド基本データ型は、パックドバイト、パックドワード、パックド・ダブルワードである（図 4-6. を参照）。これらのデータ型の数値 SIMD 演算を実行する場合、これらのデータ型は、バイト整数、ワード整数、またはダブルワード整数の値を含むものとして解釈される。

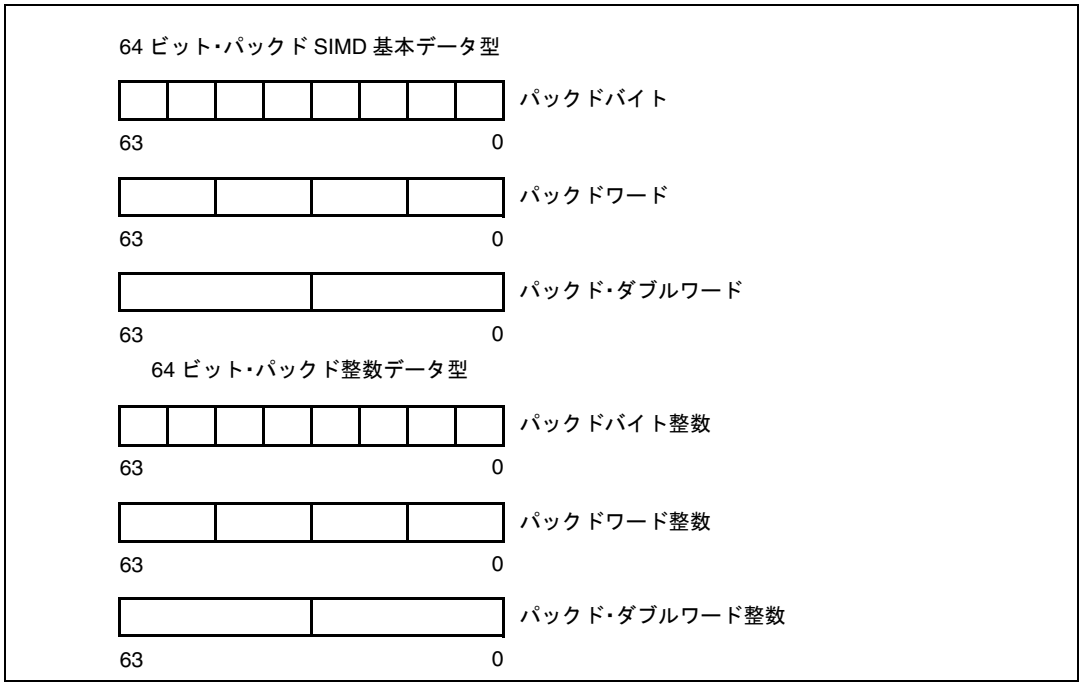


図 4-6. 64 ビット・パックド SIMD データ型

4.6.2. 128 ビット・パックド SIMD データ型

128 ビット・パックド SIMD データ型は、SSE で IA-32 アーキテクチャに導入され、SSE2 と SSE3 で使用された。これらのデータ型は、主に 128 ビット XMM レジスタとメモリ内で操作される。128 ビット・パックド基本データ型は、パックドバイト、パックドワード、パックド・ダブルワード、およびパックド・クワッドワードである（図 4-7. を参照）。XMM レジスタ内でこれらの基本データ型の SIMD 演算を実行する場合、これらのデータ型は、パックドまたはスカラ形式の単精度浮動小数点値または倍精度浮動小数点値を含むものとして解釈されるか、パックドバイト整数、パックドワード整数、パックド・ダブルワード整数、またはパックド・クワッドワード整数の値を含むものとして解釈される。

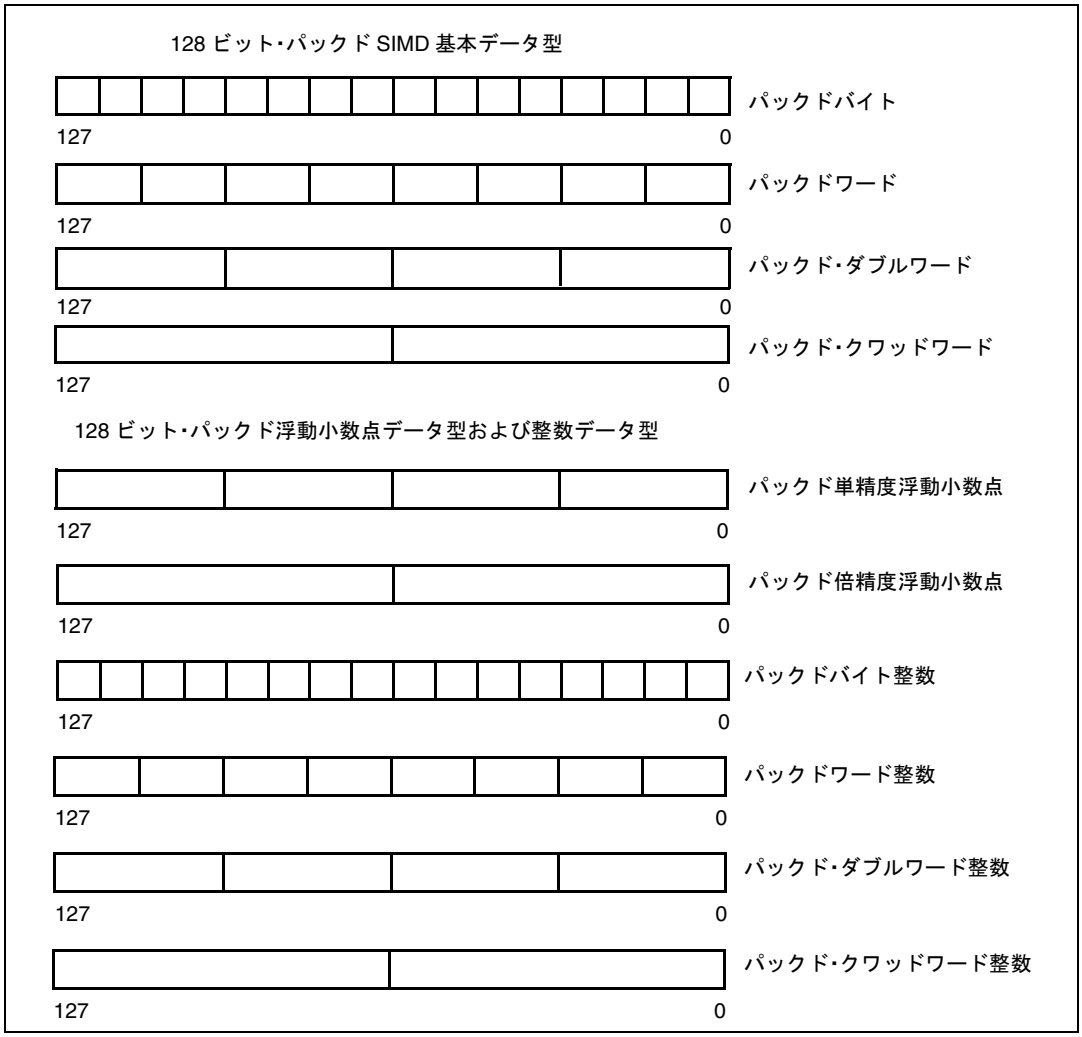


図 4-7. 128 ビット・パック ド SIMD データ型

4.7. BCD およびパックド BCD 整数

2 進化 10 進整数（BCD 整数）は、範囲 0 ～ 9 の有効値を持つ符号なし 4 ビット整数である。IA-32 アーキテクチャは、1 つ以上の汎用レジスタ内または 1 つ以上の x87 FPU レジスタ内にある BCD 整数の演算を定義している（図 4-8. を参照）。

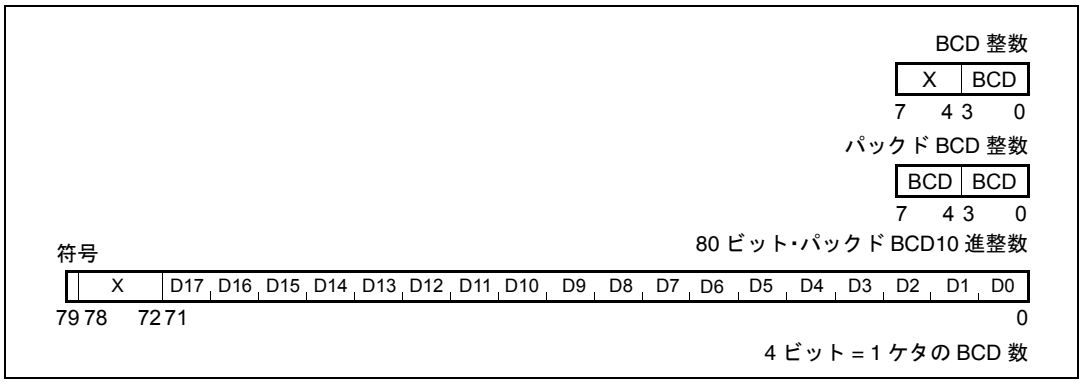


図 4-8. BCD データ型

汎用レジスタ内の BCD 整数を操作する場合、BCD 値は、アンパック形式（1 バイトあたり 1 ケタの BCD）かパック形式（1 バイトあたり 2 ケタの BCD 数）のいずれかをとることができる。アンパック形式 BCD 整数の値は、下位半バイト（ビット 0 ～ 3）の 2 進値になる。上位半バイト（ビット 4 ～ 7）は、加算や減算時には任意の値をとることができるが、乗算や除算時にはゼロでなければならない。パック形式 BCD 整数を使用すれば、2 ケタの BCD を 1 バイトに格納できる。この場合、上位半バイト内の桁が、下位半バイト内の桁より上位になる。

x87 FPU データレジスタ内の BCD 整数を操作する場合、BCD 値は 80 ビット・フォーマットのパックド値となり、10 進整数として参照される。10 進整数は、10 バイトのパックド BCD フォーマットで格納される。このフォーマットでは、最初の 9 バイトが、1 バイト当たり 2 ケタずつ、18 ケタの BCD 数を保持する。最下位の桁はバイト 0 の下位半バイトに格納され、最上位の桁はバイト 9 の上位半バイトに格納される。バイト 10 の最上位ビットは、符号ビット（0 = 正、1 = 負）を格納する（バイト 10 のビット 0 ～ 6 は無視される）。負の 10 進整数は、2 の補数形式では格納されない。負の 10 進整数と正の 10 進整数は、符号ビットでのみ区別される。このフォーマットでコード化できる 10 進整数の範囲は、$-10^{18} + 1$ ～ $10^{18}-1$ である。

10 進整数フォーマットは、メモリ内にのみ存在する。10 進整数は、x87 FPU データレジスタにロードされると、自動的に拡張倍精度浮動小数点フォーマットに変換される。すべての 10 進整数は、拡張倍精度フォーマットで正確に表現できる。

10 進整数は、10 バイトのパック形式 BCD フォーマットで格納される。表 4-2. に、このデータ型の精度と範囲を、また図 4-8. にそのフォーマットを示す。このフォーマットでは、最初の 9 バイトに（1 バイトあたり 2 ケタずつ）あわせて 18 ケタの BCD が格納される。最下位桁はバイト 0 の下位半バイトに、また最上位桁はバイト 9 の上位半バイトにそれぞれ格納される。バイト 10 の最上位ビットは、符号ビットである（0 = 正、1 = 負）。

（バイト 10 のビット 0 ～ 6 は、「無視」ビット。）負の 10 進整数は、2 の補数形式では格納されない。負の 10 進数と正の 10 進数は、符号ビットだけで区別される。

表 4-4. に、10 進整数データ型の値の可能なエンコーディングを示す。

表 4-4. パック形式 10 進整数のエンコーディング

			絶対値					
クラス	符号		桁	桁	桁	桁	…	桁
正								
最大	0	0000000	1001	1001	1001	1001	...	1001
	.	.			.			
	.	.			.			
最小	0	0000000	0000	0000	0000	0000	...	0001
ゼロ	0	0000000	0000	0000	0000	0000	...	0000
負								
ゼロ	1	0000000	0000	0000	0000	0000	...	0000
最小	1	0000000	0000	0000	0000	0000	...	0001
	.	.			.			
	.	.			.			
最大	1	0000000	1001	1001	1001	1001	...	1001
パック形式 BCD 整数不定値	1	1111111	1111	1111	1100	0000	...	0000
	← 1 バイト →		← 9 バイト →					

10 進整数フォーマットは、メモリ内でだけの表現である。10 進整数は、x87 FPU のデータレジスタにロードされると自動的に拡張倍精度フォーマットに変換される。すべての 10 進整数は、拡張倍精度フォーマットで正確に表現できる。

パック形式の 10 進不定数のエンコーディングは、（FFFFC000000000000000H）は、EBSTP 命令によって、マスクされている浮動小数点無効操作例外に対する応答として格納される。この値を FBLD 命令でロードしようとすると、未定義の結果になる。

4.8. 実数フォーマットと浮動小数点フォーマット

本節では、x87 FPU、SSE、SSE2、SSE3 浮動小数点命令において実数がどのように浮動小数点フォーマットで表現されるかを説明する。また、ノーマル型数、デノーマル型数、バイアス付き指数、符号付きゼロ、NaN などの用語についても説明する。2 進浮動小数点演算技法やIEEE 754浮動小数点演算規格をすでに熟知している読者であれば、本節を飛ばしても差し支えない。

4.8.1. 実数体系

図 4-9. に示すように、実数体系はマイナス無限大（- ∞）からプラス無限大（+ ∞）までの範囲にある実数の連続体で構成される。

コンピュータが持つことができるレジスタのサイズや数には制限があるため、実数（浮動小数点）の計算では実数の連続体の一部分しか使用できない。図 4-9. の下部に示すように、IA-32 アーキテクチャがサポートする実数の部分集合は、実数体系を近似的に表現したものである。この実数の部分集合の範囲と精度は、IEEE 754 規格の浮動小数点フォーマットによって決まる。

4.8.2. 浮動小数点フォーマット

実数計算の速度と効率を上げるため、コンピュータやマイクロプロセッサでは一般的に実数を 2 進浮動小数点フォーマットで表す。このフォーマットでは、実数は符号、仮数、指数の 3 つの部分で構成される（図 4-10. を参照）。

符号部は、数値が正（0）か負（1）のいずれであるかを示す 2 進値である。仮数部は、1 ビットの 2 進整数部分（J ビットとも呼ばれる）と 2 進小数部分で構成される。J ビットは表現されないで、暗黙の値となる場合が多い。指数部は、仮数部が累乗される 2 のベキ値を表す 2 進整数である。

表 4-5. に、通常の 10 進フォーマットの実数 178.125 が、どのように IEEE 規格 754 浮動小数点フォーマットで格納されるかを示す。この表から、実数表記が、単精度実数の 32 ビット浮動小数点フォーマット（FPU がサポートする浮動小数点フォーマットの 1 つ）に移行する経緯が分かる。浮動小数点フォーマットでは、仮数部はノーマライズされ（4.8.2.1. 項「ノーマル型数」を参照）、また指数部にはバイアスがかけられる（4.8.2.2. 項「バイアス付き指数」を参照）。単精度浮動小数点フォーマットでは、バイアス定数は +127 になる。

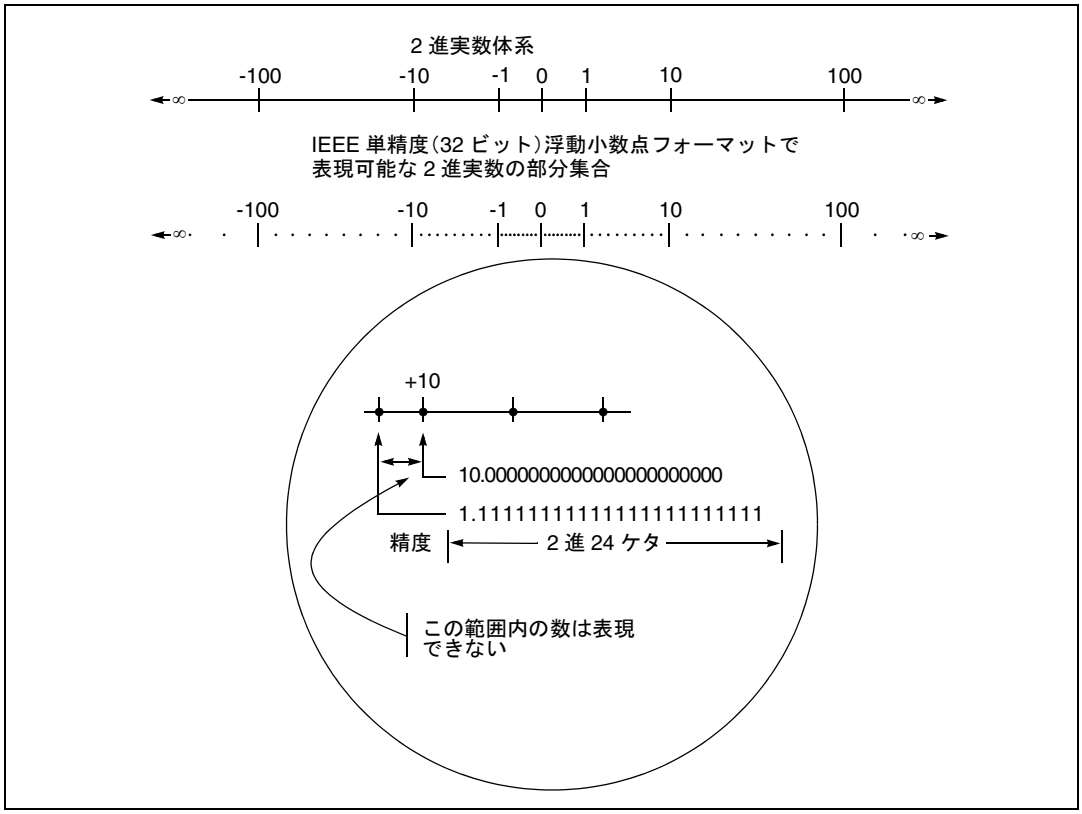


図 4-9. 2 進実数体系

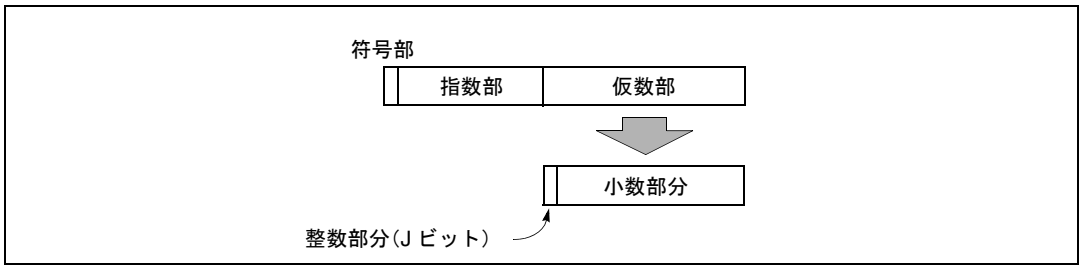


図 4-10. 2 進浮動小数点フォーマット

表 4-5. 実数および浮動小数点数表記法

表記法	値		
通常の 10 進	178.125		
科学計算用 10 進	$1.78125E_{10}2$		
科学計算用 2 進	$1.0110010001E_{2}111$		
科学計算用 2 進 (バイアス付き指数)	$1.0110010001E_{2}10000110$		
IEEE 単精度フォーマット	符号	バイアス付き指数	ノーマル型仮数
	0	10000110	01100100010000000000000 ↑ 1.(暗黙)

4.8.2.1. ノーマル型数

ほとんどの場合、浮動小数点値はノーマル型形式でコード化される。つまり、ゼロの場合を除き、仮数部は必ず 1 の整数部分と、その後に続く小数部分で構成される。

- 1.fff...ff

1 より小さい値に対しては、先行のゼロが削られる。(先行ゼロが 1 つ削られるごとに、指数部が 1 ずつデクリメントされる。)

ノーマル型形式で数値を表すと、仮数部に与えられた長さの最大限まで有効桁数を増やすことができる。要約すると、ノーマル型実数は、1 と 2 の間にある実数を表すノーマル型仮数部と、その実数値の 2 進小数点を指定する指数部で構成される。

4.8.2.2. バイアス付き指数

IA-32 アーキテクチャでは、浮動小数点数指数部をバイアス付きの形式でコード化する。つまり、バイアス付き指数が常に正の数になるよう、実際の指数に定数(バイアス定数)が加算される。このバイアス定数の値は、現在使用されている浮動小数点フォーマットで指数部を表すのに使用可能なビット数によって決まる。バイアス定数は、最小のノーマル型数がオーバーフローを生じないで逆数に変換できるように選択される。

IA-32 アーキテクチャが各種サイズの浮動小数点データ型で使用するバイアス定数の一覧については、4.2.2. 項「浮動小数点データ型」を参照のこと。

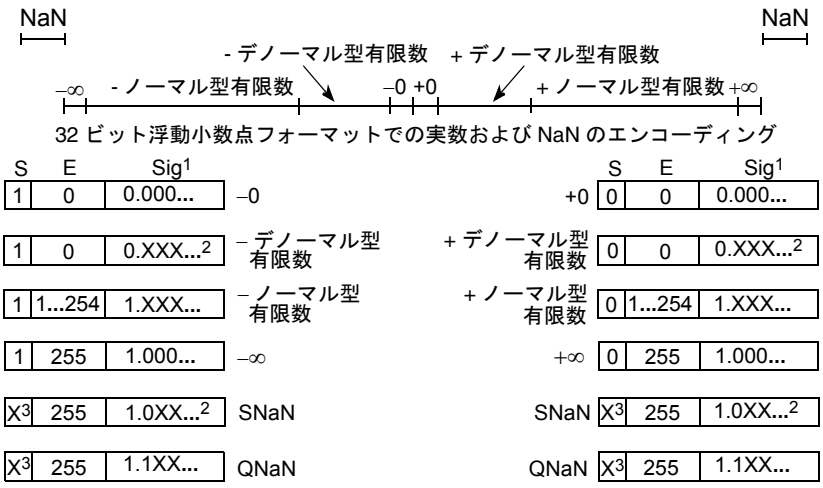
4.8.3. 実数および非数のエンコーディング

IEEE 754 規格の浮動小数点フォーマットでは、各種の実数や特殊な値をコード化できる。これらの数や値は、通常は次のクラスに分類できる。

- 符号付きゼロ
- デノーマル型有限数
- ノーマル型有限数
- 符号付き無限大
- NaN
- 不定数

（NaN は "Not a Number（非数）" を表す。）

図 4-11. に、これらの数や非数のエンコーディングが実数の連続体のどの部分を占めるかを示す。ここに示すエンコーディングは、IEEE の単精度（32 ビット）フォーマットである。略号 "S" は符号ビットを、"E" はバイアス付き指数を、"Sig" は仮数部分をそれぞれ表す。指数値は、10 進で示されている。単精度浮動小数点フォーマットでは整数ビットは暗黙的に指定されるが、仮数部分については整数ビットを示す。



注:
1. 小数部分の整数ビットは単精度浮動小数点フォーマットを表す。
2. 小数部分は、非ゼロでなければならない。
3. 符号ビットは無視される。

図 4-11. 実数と NaN

IA-32 プロセッサは、実行される演算のタイプによっては、これらの値の任意のものを演算処理したり、演算結果として返せる。以降の各項では、これらの数と非数のクラスについて説明する。

4.8.3.1. 符号付きゼロ

ゼロは、符号ビットによって +0 か -0 として表現できる。どちらのエンコーディングも、等しい値を表す。結果としてゼロが得られた場合は、実行された演算と使用された丸めモードによって符号が決まる。符号付きゼロは、区間演算を実現できるように用意されたものである。ゼロに符号を付けることによって、アンダーフローが発生した方向を示すことができる。あるいは逆数変換された結果である∞の符号を示すことができる。

4.8.3.2. ノーマル型有限数とデノーマル型有限数

非ゼロの有限数は、ノーマル型とデノーマル型の 2 つのクラスに分類できる。ノーマル型有限数はあらゆる非ゼロの有限値が含まれるが、これらの値はゼロから∞までの範囲のノーマル型実数フォーマットでコード化できる。図 4-11. に示す単精度浮動小数点フォーマットでは、1 から 254_{10} までの範囲のバイアス付き指数を持つすべての数値が含まれる（バイアスなしの場合は、指数の範囲は -126_{10} から $+127_{10}$ までになる）。

浮動小数点値が限りなくゼロに近づくと、ノーマル型数値フォーマットを使用して数を表すことは不可能になる。これは、指数の範囲が不足して、先行のゼロを削るために 2 進小数点を右にシフトすることができなくなるためである。

バイアス付き指数がゼロの場合は、仮数部の整数ビットを（場合によっては他の先行ビットも）ゼロにすることによって表現できるのは小さい数だけになる。この範囲の数は、**デノーマル型**（または**極小**）数と呼ばれる。デノーマル型数の前にゼロを使用すれば、小さい数を表現することができる。ただし、このデノーマライズ処理によって、精度は失われる（小数部分の有効ビット数が、先行ゼロの数だけ減るため）。

ノーマル型浮動小数点計算を実行する場合は、IA-32 プロセッサは通常はノーマル型数に対して演算を実行し、ノーマル型数で結果を生成する。結果がデノーマル型数になった場合は、**アンダーフロー**条件が発生したことを表す。厳密な条件については、4.9.1.5. 項「数値アンダーフロー例外（#U）」を参照のこと。

デノーマル型数は、段階的アンダーフローと呼ばれる技法によって計算される。表 4-6. に、デノーマライズ処理における段階的アンダーフローの例を示す。この例では、単精度フォーマットが使用されているため、最小の指数（バイアスなし）は -126_{10} になる。この例で、真の結果をノーマル型数で得るためには、-129_{10} の指数が必要になる。-129_{10} は、使用可能な指数の範囲を超えているため、最小の指数である -126_{10} に達するまで先行ゼロを挿入することで、結果がデノーマライズされる。

表 4-6. デノーマライズ処理

操作	符号	指数 *	仮数
真の結果	0	-129	1.01011100000...00
デノーマライズ	0	-128	0.10101110000...00
デノーマライズ	0	-127	0.01010111000...00
デノーマライズ	0	-126	0.00101011100...00
デノーマライズ結果	0	-126	0.00101011100...00

* バイアスなし、10 進数として表現。

極端な場合では、先行ゼロの数だけ右にシフトされてすべての有効ビットが落ちてしまい、結果がゼロになるときがある。

IA-32 アーキテクチャは、デノーマル値を次の方法で処理する。

- 可能な限り数値をノーマライズすることで、デノーマル値が生成されるのを避ける。
- デノーマル値が生成された場合にプログラマが検出できるよう、浮動小数点アンダーフロー例外を設ける。
- 計算のソース・オペランドとしてデノーマル値が使用されようとした場合に、プロシージャまたはプログラムがそれを検出できるよう、浮動小数点デノーマル・オペランド例外を設ける。

4.8.3.3. 符号付き無限大

+∞と -∞の 2 つの無限大は、浮動小数点フォーマットで表すことができる最大の正の実数と負の実数を表す。無限大は、常に、1.00...00 の仮数部（整数ビットは暗黙的に設定される）と、指定されたフォーマットで許容される最大のバイアスされた指数部（例えば、単精度フォーマットでは 255_{10}）で表現される。

無限大の符号は検出と比較が可能である。無限大においては、常に、-∞はあらゆる有限数より小さく、+∞はあらゆる有限数より大きいと解釈される。また、無限大に対する演算は、常に正確になる。例外が発生するのは、無限大をソース・オペランドとして使用したために無効演算になる場合だけである。

結果がデノーマル型数になった場合はアンダーフロー状態を表すのに対し、結果が正もしくは負いずれかの無限大（∞）数になった場合は結果がオーバーフローであることを表す。この場合、ノーマライズされた計算結果は、指定の結果のフォーマットに対して許可されている最大の指数より大きいバイアス付き指数を持つ。

4.8.3.4. NaN（Not a Number）

NaN は非数であるため、実数ラインの一部ではない。図 4-11. において、浮動小数点フォーマットでの NaN に対するエンコーディング空間は、実数ラインの両端の上部に示されている。この空間には、許容される最大のバイアスされた指数部と 0 でない小数部分を持つ任意の値が含まれる（NaN では、符号ビットは無視される）。

IA-32 アーキテクチャ規格では、クワイエット型 NaN（QNaN）とシグナル型 NaN（SNaN）の 2 クラスの NaN が定義されている。QNaN は、小数部分の最上位ビットがセットされている NaN であり、SNaN は小数部分の最上位ビットがクリアされている NaN である。QNaN は、大部分の算術演算において例外が通知されずに処理される。SNaN は、一般的には算術演算でオペランドとして現れた場合に浮動小数点無効操作例外を通知する。

SNaN は、一般的に例外ハンドラをトラップしたり呼び出すのに使用する。また、プロセッサが浮動小数点操作の結果として SNaN を生成することはないため、SNaN はソフトウェアによって挿入しなければならない。

4.8.3.5. SNaN と QNaN の操作

SNaN または QNaN、あるいはその両方に対して浮動小数点演算を実行すると、以下の規則に基づいて、デスティネーション・オペランドに QNaN が格納されるか、または浮動小数点無効操作例外が生成される。

- ソース・オペランドのうち 1 つが SNaN であり、浮動小数点無効操作例外がマスクされていない場合は（4.9.1.1. 項「無効操作例外（#I）」を参照）、浮動小数点無効操作例外が報告され、結果はデスティネーション・オペランドに格納されない。
- いずれかまたは両方のソース・オペランドが NaN であり、浮動小数点無効操作例外がマスクされている場合は、結果は表 4-7. に示すようになる。SNaN が QNaN に変換される場合は、SNaN の最上位の小数ビットが 1 にセットされる。また、いずれかのソース・オペランドが SNaN の場合は、浮動小数点無効操作例外フラグがセットされる。ただし、ソース・オペランドの組み合わせによっては、x87 FPU 操作と SSE/SSE2/SSE3 操作では演算結果が異なる。
- どちらのソース・オペランドも NaN ではないにもかかわらず、演算によって浮動小数点無効操作例外が生成された場合（表 8-10. と表 11-1. を参照）は、通常演算結果は QNaN または QNaN 浮動小数点不定値に変換された SNaN ソース・オペランドになる。

表 4-7. で説明した動作の例外については、8.5.1.2. 項「無効算術オペランド例外（#IA）」と 11.5.2.1. 項「無効操作例外（#I）」を参照のこと。

表 4-7. NaN の処理の規則

ソース・オペランド	結果 [1]
SNaN と QNaN	x87 FPU - QNaN ソース・オペランド SSE、SSE2 または SSE3 - 第 1 オペランド（このオペランドが SNaN である場合は、QNaN に変換される）。
2 つの SNaN	x87 FPU - 仮数が大きい方の SNaN ソース・オペランドが、QNaN に変換される。 SSE、SSE2 または SSE3 - 第 1 オペランドが QNaN に変換される。
2 つの QNaN	x87 FPU - 仮数が大きい方の QNaN ソース・オペランド SSE、SSE2 または SSE3 - 第 1 オペランド
SNaN と浮動小数点値	SNaN ソース・オペランドが QNaN に変換される。
QNaN と浮動小数点値	QNaN ソース・オペランド
SNaN（オペランドを 1 つだけ使用する命令の場合）	SNaN ソース・オペランドが QNaN に変換される。
QNaN（オペランドを 1 つだけ使用する命令の場合）	QNaN ソース・オペランド

注 :

1 SSE、SSE2、SSE3 では、一般的に第 1 オペランドがソース・オペランドであり、このオペランドがデスティネーション・オペランドになる。「結果」列では、x87 FPU の表記が SSE3 の FISTTP 命令にも適用される。SSE3 の表記は、12.3.3. 項で説明されている SIMD 浮動小数点命令に適用される。

4.8.3.6. アプリケーションでの SNAN と QNAN の使用

4.8.3.4. 項「NaN（Not a Number）」の始めに示した SNaN と QNaN に関する規則以外は、ソフトウェア上で NaN の仮数部のビットを任意の目的で使用できる。SNaN と QNaN は、いずれも診断情報などのデータの伝達や格納の目的でエンコーディングできる。

無効操作例外をアンマスクすれば、シグナル型 NaN を使用して例外ハンドラをトラップできる。この技法が持つ汎用性と、大量に使用可能な NaN 値を利用すれば、プログラマはさまざまな特殊状況に適用可能なツールを得られる。

例えば、コンパイラでは、初期化されていない（実数の）配列要素に対するリファレンスとしてシグナル型 NaN を使用できる。コンパイラでは、仮数部に要素のインデックス（相対位置）が格納されたシグナル型 NaN を使用して配列の各要素をあらかじめ初期化できる。この後、まだ初期化されていない要素にアプリケーション・プログラムがアクセスを試みた場合、プログラムからはコンパイラによって該当位置に配置された NaN を使用できる。無効操作例外がマスクされていない場合は、割り込みが発生し、例外ハンドラが呼び出される。例外ハンドラは、例外ポインタのオペランド・アドレス・フィールドによって NaN がポイントされ、しかも NaN に配列要素のインデックス番号が格納されるため、いずれの要素がアクセスされたかを判断できる。

多くの場合、クワイエット型 NaN は、デバッグの効率を改善するために使用される。初期のテスト段階では、プログラムには複数のエラーが含まれることが多い。例外ハ

ンドラを作成すれば、例外ハンドラが呼び出されるたびに診断情報をメモリにセーブできる。診断データを格納した後、例外ハンドラは誤操作の原因となった命令の結果としてクワイエット型NaNを与えられる。このNaNは、メモリ内での自身に関連付けられた診断部分をポイントできる。この後、プログラムは実行を再開し、エラーが発生するたびに異なるNaNを作成できる。プログラムが終了した後にNaNの結果を使用すれば、エラーが発生した時点でセーブされた診断データにアクセスできる。これにより、1回のテスト実行で、多くのエラーを診断し、修正できる。

計算済みの結果をさらに次の計算で使用するような組み込み型アプリケーションでは、検出されなかったQNaNによって後に続くすべての結果が無効になることがある。したがって、このようなアプリケーションでは、QNaNの有無を定期的にチェックし、QNaN の結果が検出された場合に使用する回復機構を組み込んでおかなければならない。

4.8.3.7. QNAN 浮動小数点不定数

浮動小数点データ型のエンコーディング（単精度、倍精度、拡張倍精度）については、**QNaN浮動小数点不定値**と呼ばれる特殊な値を表現するために、1つの独自のエンコーディング（QNaN）が予約されている。x87 FPU 命令、SSE、SSE2、SSE3 は、マスクされている浮動小数点例外に対する応答として、これらの不定値を返す。表 4-3. は、QNaN浮動小数点不定値に使用されるエンコーディングを示している。

4.8.4. 丸め

浮動小数点演算を実行するとき、プロセッサは、可能な限り、デスティネーション・フォーマット（単精度、倍精度、または拡張倍精度浮動小数点）で、無限精度の浮動小数点数の結果を返す。しかし、IEEE 規格 754 の浮動小数点フォーマットでは実数連続体の値の一部しか表現できないため、無限精度の結果を、デスティネーション・オペランドのフォーマットで正確にコード化できないことがある。

例えば、以下の値（*a*）は、24 ビットの小数部分を持つ。この小数の最下位ビット（下線のビット）は、単精度フォーマットでは正確にコード化できない（単精度実数フォーマットの小数部分は23 ビットしかない）。

(*a*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011\underline{1}E_2\ 101$

プロセッサは、この結果（*a*）を丸めるために、最初に、*a* を最も近い値で囲む2つの表現可能な小数 *b* と c を選択する（$b < a < c$）。

(*b*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$

(*c*) $1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$

次に、プロセッサは、選択された丸めモードにしたがって、結果を *b* または *c* に設定する。丸めによって結果に誤差が生じるが、この誤差は、結果を丸めた値の最後の桁（浮動小数点値の最下位ビットの位置）の 1 単位より小さい。

IEEE754 は、直近値への丸め、切り上げ、切り捨て、ゼロ方向への丸めの 4 つの丸めモードを定義している（表 4-8. を参照）。IA-32 アーキテクチャのデフォルトの丸めモードは、直近値への丸めである。このモードは、真の結果に対する最も正確で統計的に偏りのない推定が可能であり、ほとんどのアプリケーションに適合する。

表 4-8. 丸めモードと丸め制御（RC）フィールドのエンコーディング

丸めモード	RC フィールドの設定	説明
直近値への丸め（偶数）	00B	丸められた結果は、無限精度の結果に最も近い値になる。2 つの値が同じ近さの場合は、結果は偶数値（すなわち、最下位ビットが 0 の値）になる。これがデフォルトである。
切り捨て（- ∞方向）	01B	丸められた結果は、無限精度の結果に最も近い値（ただし、無限精度の結果より大きくない値）になる。
切り上げ（+ ∞の方向）	10B	丸められた結果は、無限精度の結果に最も近い値（ただし、無限精度の結果より小さくない値）になる。
ゼロ方向への丸め（真の切り捨て）	11B	丸められた結果は、無限精度の結果に最も近い値（ただし、無限精度の結果より絶対値が大きくない値）になる。

切り上げモードと切り捨てモードは、**有向丸め**と呼ばれ、区間演算に使用される。多段階にわたる計算で中間結果が丸められる場合は、区間演算を使用して、真の結果の上限と下限を求めることができる。

ゼロ方向への丸めモード（「チョップ」モードとも呼ばれる）は、x87 FPU で整数演算を実行するときによく使用される。

丸められた結果は、不正確結果と呼ばれる。プロセッサが不正確結果を返した場合は、浮動小数点精度（不正確）フラグ（PE）がセットされる（4.9.1.6. 項「不正確結果（精度）例外（#P)」を参照)。

丸めモードは、比較演算、正確な結果を返す演算、NaN の結果を返す演算には影響を与えない。

4.8.4.1. 丸め制御（RC）フィールド

IA-32 アーキテクチャでは、丸めモードは、2 ビットの丸め制御（RC）フィールドによって制御される（このフィールドのエンコーディングを表 4-8. に示す)。RC フィールドは、次の 2 つの異なる位置に実装されている。

- x87 FPU コントロール・レジスタ（ビット 10 とビット 11）
- MXCSR レジスタ（ビット 13 とビット 14）

これらの 2 つの RC フィールドは、同じ機能を持っているが、プロセッサ内の異なる実行環境の丸めモードを制御する。x87 FPU コントロール・レジスタの RC フィールドは、x87 FPU 命令によって実行される計算の丸めを制御する。MXCSR レジスタの RC フィールドは、SSE と SSE2 によって実行される SIMD 浮動小数点計算の丸めを制御する。

4.8.4.2. SSE および SSE2 変換命令による切り捨て

SSE および SSE2 の変換命令 CVTTPD2DQ、CVTTPS2DQ、CVTTPD2PI、CVTTPS2PI、CVTTSD2SI、CVTTSS2SI は、浮動小数点値から整数への変換の結果が不正確である場合、その結果を自動的に切り捨てる。切り捨てとは、表 4-8. で説明したゼロ方向への丸めモードを意味する。

4.9. 浮動小数点例外の概要

この節では、IA-32 アーキテクチャの浮動小数点例外とその処理の概要について説明する。x87 FPU、SSE、SSE2、SSE3 に固有の内容については、以下の各節を参照のこと。

- 8.4. 節「x87 FPU 浮動小数点例外処理」
- 11.5. 節「SSE、SSE2、SSE3 の例外」

IA-32 アーキテクチャは、浮動小数点オペランドを操作するとき、以下の 6 クラスの例外条件を認識し、検出する。

- 無効操作（#I）
- ゼロ除算（#Z）
- デノーマル・オペランド（#D）
- 数値オーバーフロー（#O）
- 数値アンダーフロー（#U）
- 不正確結果（精度）（#P）

本書では、"#" 記号に続く大文字 1 文字または 2 文字の表記（例えば、#P）を使用して、例外条件を表す。これは単なる省略形であり、アセンブラのニーモニックとは無関係である。

注記

上記のすべての例外は、デノーマル・オペランド例外（#D）を除いて、IEEE 規格 754 に定義されている。

無効操作例外、ゼロ除算例外、デノーマル・オペランド例外は、計算前型の例外（すなわち、算術演算が実行される前に検出される例外）である。数値アンダーフロー例外、数値オーバーフロー例外、精度例外は、計算後型の例外である。

6 つの例外クラスのそれぞれに、対応するフラグビット（IE、ZE、OE、UE、DE、またはPE）とマスクビット（IM、ZM、OM、UM、DM、またはPM）がある。1 つ以上の浮動小数点例外条件が検出されると、プロセッサは、該当するフラグビットをセットし、それに対応するマスクビットの設定に基づいて、次のいずれかの処置をとる。

- マスクビットがセットされている場合。例外を自動的に処理して、あらかじめ定義された（通常はそのまま使用可能な）結果を返し、プログラムの実行を続ける。
- マスクビットがクリアされている場合。ソフトウェア例外ハンドラを起動して、例外を処理する。

例外に対するマスク応答（デフォルト）は、各例外条件に対して妥当な結果が得られるように選択されており、ほとんどの浮動小数点アプリケーションでは、一般に満足のいく結果が得られる。プログラマは、それぞれの浮動小数点例外をマスクしたり、マスクを解除することで、ほとんどの例外の処理をプロセッサに任せて、最も重大な例外条件だけをソフトウェア例外ハンドラで処理できる。

例外フラグは「スティッキー・フラグ」であるため、前回クリアされた後に発生した例外を累積的に記録している。プログラマは、すべての例外をマスクしておき、計算を実行した後で例外フラグを調べて、計算中に例外が検出されたかどうかを確認できる。

IA-32 アーキテクチャでは、浮動小数点例外フラグビットとマスクビットは、次の 2 つの異なる位置に実装されている。

- x87 FPU ステータス・ワードおよび制御ワード。フラグビットは、x87 FPU ステータス・ワードのビット 0 ～ 5 にある。マスクビットは、x87 FPU 制御ワードのビット 0 ～ 5 にある（図 8-6 と図 8-4 を参照）。
- MXCSR レジスタ。フラグビットは、MXCSR レジスタのビット 0 ～ 5 にある。マスクビットは、MXCSR レジスタのビット 7 ～ 12 にある（図 10-3. を参照）。

これらの 2 組のフラグビットとマスクビットは、同じ機能を持っているが、プロセッサ内の異なる実行環境の例外の報告と制御に使用される。x87 FPU ステータス・ワードおよび制御ワード内のフラグビットとマスクビットは、x87 FPU 命令によって実行される計算で発生した例外の報告とマスクを制御する。MXCSR レジスタ内のそれに対応するビットは、SSE と SSE2 によって実行される SIMD 浮動小数点計算で発生した例外の報告とマスクを制御する。

ただし、例外がマスクされている場合、プロセッサはマスク応答の実行後も命令の実行を続けるため、1 つの命令で複数の例外が検出されることがある。例えば、プロセッ

サは、デノーマル・オペランドを検出し、この例外に対するマスク応答を実行した後、数値アンダーフローを検出することがある。

1 つの命令に対して複数の浮動小数点例外条件が検出された場合の例外の優先規則については、4.9.2. 項「浮動小数点例外の優先順位」を参照のこと。

4.9.1. 浮動小数点例外条件

以下の各項では、SIMD 浮動小数点数値例外を発生させる各種の条件と、これらの条件の検出時のプロセッサのマスク応答について説明する。各浮動小数点命令について通知される浮動小数点例外の一覧は、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」に記載されている。

4.9.1.1. 無効操作例外（#I）

無効操作例外は、1 つ以上の無効な算術演算オペランドに対して報告される。無効操作例外がマスクされている場合は、プロセッサは IE フラグをセットし、不定値または QNaN を返す。この値は、命令によって指定されたデスティネーション・レジスタを上書きする。無効操作例外がマスクされていない場合は、IE フラグがセットされ、ソフトウェア例外ハンドラが起動され、オペランドは変更されない。

SNaN によって無効操作例外が発生した場合に返される結果については、4.8.3.6. 項「アプリケーションでの SNAN と QNAN の使用」を参照のこと。

プロセッサは、プログラム内にコーディングされる、各種の無効な算術演算を検出することができる。これらの演算は、一般的に、∞を∞で割るなどのプログラミング・エラーを示す。x87 FPU 命令、または SSE、SSE2、SSE3 の実行中に検出される無効操作例外については、以下の各項を参照のこと。

- x87 FPU：8.5.1. 項「無効操作例外」
- SIMD 浮動小数点例外：11.5.2.1. 項「無効操作例外（#I）」

4.9.1.2. デノーマル・オペランド例外（#D）

算術命令がデノーマル・オペランドを操作しようとすると、プロセッサはデノーマル・オペランド例外を通知する（4.8.3.2. 項「ノーマル型有限数とデノーマル型有限数」を参照）。デノーマル・オペランド例外がマスクされている場合は、プロセッサは DE フラグをセットし、命令の実行を続ける。デノーマル数をそのまま処理した場合は、デノーマル数をゼロにフラッシュした場合と比べて、少なくとも同程度に正確な結果

（多くの場合はより正確な結果）が得られる。プログラマは、この例外をマスクしておいて計算を続行し、最終結果が得られた段階で、精度の低下を分析できる。

デノーマル・オペランド例外がマスクされていない場合は、DE ビットがセットされ、ソフトウェア例外ハンドラが起動され、オペランドは変更されない。下位ビットが失われたためにデノーマル・オペランドの有意性が低下した場合は、デノーマル・オペランドに対して演算を行わない方がよい。マスクされていないデノーマル・オペランド例外に応答する例外ハンドラを使用すれば、デノーマル・オペランドをあらかじめ計算から除外できる。

x87 FPU 命令、または SSE、SSE2、SSE3 の実行中に検出されるデノーマル・オペランド例外については、以下の各項を参照のこと。

- x87 FPU：8.5.2. 項「デノーマル・オペランド例外（#D）」
- SIMD 浮動小数点例外：11.5.2.2. 項「デノーマル・オペランド例外（#D）」

4.9.1.3. ゼロ除算例外（#Z）

ゼロでない有限のオペランドを 0 で割ろうとすると、浮動小数点ゼロ除算例外が報告される。ゼロ除算例外に対するマスク応答は、ZE フラグをセットし、各オペランドの符号の排他的論理和（XOR）演算によって符号が付けられた無限大を返す。ゼロ除算例外がマスクされていない場合は、ZE フラグがセットされ、ソフトウェア例外ハンドラが起動され、オペランドは変更されない。

x87 FPU 命令、または SSE、SSE2 の実行中に検出されるゼロ除算例外については、以下の各項を参照のこと。

- x87 FPU：8.5.3. 項「ゼロ除算例外（#Z）」
- SIMD 浮動小数点例外：11.5.2.3. 項「ゼロ除算例外（#Z）」

4.9.1.4. 数値オーバーフロー例外（#O）

命令の丸められた結果が、デスティネーション・オペランドの有効範囲内で最も大きい有限値を超えた場合は、プロセッサは浮動小数点数値オーバーフロー例外を報告する。表 4-9. は、各浮動小数点フォーマットについて、数値オーバーフローのスレッショルド範囲を示している。丸められた結果がこのスレッショルド範囲から外れるか、範囲の両端の値と一致した場合に、オーバーフローが発生する。

表 4-9. 数値オーバーフローのスレッショルド

浮動小数点フォーマット	オーバーフロー・スレッショルド
単精度	$\|x\| \geq 1.0 * 2^{128}$
倍精度	$\|x\| \geq 1.0 * 2^{1024}$
拡張倍精度	$\|x\| \geq 1.0 * 2^{16384}$

数値オーバーフロー例外が発生したときに例外がマスクされていると、プロセッサは OE フラグをセットし、現在の丸めモードにしたがって、表 4-10. に示す値のうち 1 つを返す。4.8.4. 項「丸め」を参照のこと。

表 4-10. 数値オーバーフローに対するマスク応答

丸めモード	真の結果の符号	結果
最近値	+	+∞
	–	–∞
–∞ 方向	+	正の最大有限数。
	–	–∞
+∞ 方向	+	+∞
	–	負の最大有限数。
ゼロ方向	+	正の最大有限数。
	–	負の最大有限数。

数値オーバーフローが発生し、数値オーバーフロー例外がマスクされていない場合は、OE フラグがセットされ、ソフトウェア例外ハンドラが起動される。オーバーフロー例外が SSE、SSE2、または SSE3 浮動小数点演算で発生した場合は、ソース・オペランドとデスティネーション・オペランドはどちらも変更されない。オーバーフロー例外が x87 FPU 演算で発生した場合は、バイアスさ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。

x87 FPU 命令、または SSE、SSE2、SSE3 の実行中に検出される数値オーバーフロー例外については、以下の各項を参照のこと。

- x87 FPU：8.5.4. 項「数値オーバーフロー例外（#O）」
- SIMD 浮動小数点例外：11.5.2.4. 項「数値オーバーフロー例外（#O）」

4.9.1.5. 数値アンダーフロー例外（#U）

丸められた結果が極小数（すなわち、デスティネーション・オペランドの有効範囲内で最も小さいノーマル型有限値より小さい値）になった場合、プロセッサは浮動小数点数値アンダーフロー条件を検出する。表 4-11. は、各浮動小数点フォーマットについて、数値アンダーフローのスレッショルド範囲を示している（正規化された結果を想定）。丸められた結果がこのスレッショルド範囲内（両端の値を除く）に入った場合に、アンダーフローが発生する。アンダーフローを検出して処理する機能は、非常に小さい結果が計算によって伝搬され、その後で他の例外（除算時のオーバーフローなど）を発生させるのを防ぐために用意されている。

表 4-11. 数値アンダーフローの（正規化された）スレッショルド

浮動小数点フォーマット	アンダーフロー・スレッショルド
単精度	$\|x\| < 1.0 * 2^{-126}$
倍精度	$\|x\| < 1.0 * 2^{-1022}$
拡張倍精度	$\|x\| < 1.0 * 2^{-16382}$

プロセッサがアンダーフロー条件をどのように処理するかは、それに関連する次の 2 つの条件によって異なる。

- 極小の結果の発生。
- 不正確結果の発生。すなわち、演算結果がデスティネーション・フォーマットで正確に表現できない場合。

どのイベントでアンダーフロー例外が報告されるか、またプロセッサがアンダーフロー例外条件にどのように応答するかは、アンダーフロー例外がマスクされているかどうかによって異なる。

- **アンダーフロー例外がマスクされている場合。**計算の結果が極小かつ不正確である場合にのみ、アンダーフロー例外が報告される（UE フラグがセットされる）。結果が不正確かどうかに関係なく、プロセッサはデスティネーション・オペランドにデノーマル型の結果を返す。
- **アンダーフロー例外がマスクされていない場合。**結果が不正確であるかどうかに関係なく、結果が極小である場合に、アンダーフロー例外が報告される。アンダーフロー例外が SSE、SSE2、または SSE3 浮動小数点演算で発生した場合は、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。アンダーフロー例外が x87 FPU 演算で発生した場合は、バイアスさ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。いずれの場合も、ソフトウェア例外ハンドラが起動される。

x87 FPU 命令、または SSE、SSE2、SSE3 の実行中に検出される数値アンダーフロー例外については、以下の各項を参照のこと。

- x87 FPU：8.5.4.項「数値オーバーフロー例外（#O）」
- SIMD浮動小数点例外：11.5.2.5.項「数値アンダーフロー例外（#U）」

4.9.1.6. 不正確結果（精度）例外（#P）

不正確結果例外（精度例外とも呼ばれる）は、演算の結果がデスティネーション・フォーマットで正確に表現できない場合に発生する。例えば、分数1/3は2進浮動小数点形式では正確には表現できない。この例外は頻繁に発生し、精度に多少のロス（通常は許容できる範囲内）が生じたことを示す。この例外は、正確な演算を実行する必要があるアプリケーションに対してだけサポートされる。丸められた結果は、一般的に大部分のアプリケーションにとって満足できるものであるため、この例外はマスクされることが多い。

不正確結果条件が発生したときに不正確結果例外がマスクされており、しかも数値オーバーフローまたはアンダーフローのいずれの条件も発生していない場合は、プロセッサはPEフラグをセットし、丸められた結果をデスティネーション・オペランドに格納する。結果の丸めに使用される方法は、現在の丸めモードによって決まる。4.8.4.項「丸め」を参照のこと。

不正確な結果が発生したときに不正確結果例外がマスクされておらず、数値オーバーフローも数値アンダーフローも発生しなかった場合は、PEフラグがセットされ、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、ソフトウェア例外ハンドラが起動される。

数値オーバーフローまたは数値アンダーフローと同時に不正確結果例外が発生した場合は、次の操作のいずれかが実行される。

- 不正確結果が、マスクされている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、OEフラグとUEフラグのいずれかとPEフラグがセットされ、さらにオーバーフロー例外やアンダーフロー例外で説明した方法で結果が格納される。4.9.1.4.項「数値オーバーフロー例外（#O）」と4.9.1.5.項「数値アンダーフロー例外（#U）」」を参照のこと。不正確結果例外がマスクされていない場合は、プロセッサはソフトウェア例外ハンドラも呼び出す。
- 不正確結果が、マスクされていないオーバーフローまたはアンダーフローと一緒に発生し、しかもデスティネーション・オペランドがレジスタである場合は、OEフラグとUEフラグのいずれかとPEフラグがセットされ、さらにオーバーフロー例外やアンダーフロー例外で説明した方法で結果が格納される。また、ソフトウェア例外ハンドラが呼び出される。

マスクされていない数値オーバーフロー例外またはアンダーフロー例外が発生し、デスティネーション・オペランドがメモリ・ロケーションである場合は（これは浮動小

数点のストアの場合に限られる)、不正確結果条件は報告されず、C1 フラグがクリアされる。

x87 FPU 命令、または SSE、SSE2、SSE3 の実行中に検出される不正確例外については、以下の各項を参照のこと。

- x87 FPU：8.5.6. 項「不正確結果（精度）例外（#P)」
- SIMD 浮動小数点例外

4.9.2. 浮動小数点例外の優先順位

プロセッサは、あらかじめ定められた優先順位にしたがって例外を処理する。1 つの命令が 2 つ以上の例外条件を生成したときは、場合によっては例外の優先順位のために、優先順位が高い例外が処理され、優先順位が低い例外が無視されるという結果が生じることがある。例えば、SNaN をゼロで割ると、原則的には（SNaN オペランドによる）無効算術オペランド例外とゼロ除算例外が通知されるはずである。しかし、両方の例外がマスクされていると、プロセッサは優先順位の高い方の例外（無効算術オペランド例外）だけを処理し、QNaN をデスティネーションに返す。また、デノーマル・オペランド例外あるいは不正確結果例外は数値アンダーフロー例外または数値オーバーフロー例外を伴う可能性があるが、この場合は両方の例外が処理される。

浮動小数点例外の優先順位は、次のようになる。

1. 無効操作例外。これらは、さらに次のように分類される。
   a. スタック・アンダーフロー（x87 FPU でのみ発生）
   b. スタック・オーバーフロー（x87 FPU でのみ発生）
   c. サポートされていないフォーマットのオペランド（拡張倍精度浮動小数点使用時に x87 FPU でのみ発生）
   d. SNaN オペランド
2. QNaN オペランド。これは例外ではないが、QNaN オペランドの処理の優先順位は、低優先順位の例外よりも高い。例えば、QNaN がゼロで割られると、ゼロ除算例外にはならずに QNaN が生じる。
3. 上記以外のすべての無効操作例外またはゼロ除算例外。
4. デノーマル・オペランド例外。マスクされている場合は、命令の実行が続行され、低優先順位の例外も発生できる。
5. 不正確結果例外と同時に発生する数値オーバーフロー例外および数値アンダーフロー例外。
6. 不正確結果例外。

無効操作、ゼロ除算、デノーマル・オペランドの各例外は、浮動小数点演算が開始される前に検出される。オーバーフロー、アンダーフロー、精度の各例外は真の結果が算出されるまで検出されない。マスクされていない演算前型の例外が検出された時点では、デスティネーション・オペランドはまだ更新されておらず、例外発生の原因となった命令がまだ実行されていないように見える。マスクされていない演算後型の例外が検出されたときは、デスティネーション・オペランドの結果で更新される可能性がある（ただし、SSE、SSE2、SSE3 の場合を除く。SSE、SSE2、SSE3 は、このような場合にデスティネーション・オペランドを更新しない)。

4.9.3. 浮動小数点例外ハンドラの一般的な動作

浮動小数点例外ハンドラが起動された後、プロセッサは、浮動小数点例外以外の例外を処理するのと同じ方法で浮動小数点例外を処理する。浮動小数点例外ハンドラは、通常はオペレーティング・システムまたはエグゼクティブ・ソフトウェアの一部である。このプログラムは、通常は、ユーザが登録した浮動小数点例外ハンドラを起動する。

例外ハンドラの一般的な動作は、ステート情報をメモリに格納することである。その他の例外ハンドラの一般的な動作には、以下のものがある。

- 格納されたステート情報をチェックして、エラーの性質を判定する。
- エラーの原因となった条件を修正するための処置をとる。
- 例外フラグをクリアする。
- 割り込みをかけられたプログラムに戻り、通常の実行を再開する。

例外ハンドラは、上記の回復手続きの代わりに、以下の処置を実行することもできる。

- 後で表示または印刷できるように、ソフトウェア内で例外カウンタをインクリメントする。
- 診断情報（ステート情報など）を印刷または表示する。
- プログラムの実行を停止する。

5

命令セットの要約

第 5 章
命令セットの要約

5

本章では、すべての IA-32 命令の概要を示す。IA-32 命令は、以下の主要グループに分けられる。

- 汎用命令
- x87 FPU 命令
- x87 FPU 命令と SIMD ステート管理
- MMX® テクノロジ命令
- SSE
- SSE2
- SSE3
- システム命令

表 5-1. は、各グループと、そのグループをサポートする IA-32 プロセッサを示している。各グループの命令は、さらに機能別のサブグループに分けられる。

表 5-1. 命令グループと IA-32 プロセッサ

命令セット・アーキテクチャ	サポートする IA-32 プロセッサ
汎用命令	すべての IA-32 プロセッサ
x87 FPU 命令	インテル® Intel486™ プロセッサ、インテル® Pentium® プロセッサ、インテル® MMX® テクノロジ Pentium® プロセッサ、インテル® Celeron® プロセッサ、インテル® Pentium® Pro プロセッサ、インテル® Pentium® II プロセッサ、インテル® Pentium® II Xeon™ プロセッサ、インテル® Pentium® III プロセッサ、インテル® Pentium® III Xeon™ プロセッサ、インテル® Pentium® 4 プロセッサ
x87 FPU 命令と SIMD ステート管理	インテル Pentium II プロセッサ、インテル Pentium II Xeon プロセッサ、インテル Pentium III プロセッサ、インテル Pentium III Xeon プロセッサ、インテル Pentium 4 プロセッサ
MMX テクノロジ命令	インテル MMX テクノロジ Pentium プロセッサ、インテル Celeron プロセッサ、インテル Pentium II プロセッサ、インテル Pentium II Xeon プロセッサ、インテル Pentium III プロセッサ、インテル Pentium III Xeon プロセッサ、インテル Pentium 4 プロセッサ
SSE	インテル Pentium III プロセッサ、インテル Pentium III Xeon プロセッサ、インテル Pentium 4 プロセッサ
SSE2	インテル Pentium 4 プロセッサ、インテル Xeon プロセッサ
SSE3	HT テクノロジに対応したインテル Pentium 4 プロセッサ（90nm プロセス・テクノロジを利用）
システム命令	すべての IA-32 プロセッサ

以下の各節では、各主要グループおよびサブグループの命令の一覧を示す。各命令のニーモニックと記述名が示される。2つ以上のニーモニック（例えば、CMOVA/CMOVNBE）は、同じ命令オペコードを表す異なるニーモニックである。いくつかの命令については、コードリストが読みやすくなるように、アセンブラが冗長ニーモニックをサポートしている。例えば、CMOVA（Conditional move if above）とCMOVNBE（Conditional move if not below or equal）は、同じ条件を表している。個別の命令についての詳細は、『IA-32 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、中巻 A』の第3章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第4章「命令セット・リファレンス N-Z」を参照のこと。

5.1. 汎用命令

汎用命令は、IA-32プロセッサ上で動作するアプリケーションとシステム・ソフトウェアを作成するときにプログラマがよく使用する、基本的なデータ転送操作、算術演算、論理演算、プログラム・フロー操作、ストリング操作を実行する。汎用命令は、メモリ内、汎用レジスタ（EAX、EBX、ECX、EDX、EDI、ESI、EBP、ESP）内、EFLAGSレジスタ内のデータを操作する。また、汎用命令は、メモリ内、汎用レジスタ内、セグメント・レジスタ（CS、DS、SS、ES、FS、GS）内のアドレス情報も操作する。

この命令グループには、データ転送命令、2進整数算術命令、10進算術命令、論理演算命令、シフト命令とローテート命令、ビット命令とバイト命令、プログラム制御命令、ストリング命令、フラグ制御命令、セグメント・レジスタ命令、その他の命令が含まれる。以下の各項では、各サブグループについて説明する。

汎用命令についての詳細は、第7章「汎用命令によるプログラミング」を参照のこと。

5.1.1. データ転送命令

データ転送命令は、メモリと汎用レジスタ / セグメント・レジスタの間でデータを転送する。また、データ転送命令は、条件付き転送、スタックアクセス、データ変換などの特殊な操作も実行する。

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers 汎用レジスタ間でデータを転送、メモリと汎用レジスタ / セグメント・レジスタ間でデータを転送、即値を汎用レジスタに転送
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero 等しい場合は条件付き転送 / ゼロの場合は条件付き転送
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero 等しくない場合は条件付き転送 / ゼロでない場合は条件付き転送
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal より大きい場合は条件付き転送 / より小さくなく等しくない場合は条件付き転送
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below より大きいか等しい場合は条件付き転送 / より小さくない場合は条件付き転送
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal より小さい場合は条件付き転送 / より大きくなく等しくない場合は条件付き転送
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above より小さいか等しい場合は条件付き転送 / より大きくない場合は条件付き転送
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal より大きい場合は条件付き転送 / より小さくなく等しくない場合は条件付き転送
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less より大きいか等しい場合は条件付き転送 / より小さくない場合は条件付き転送
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal より小さい場合は条件付き転送 / より大きくなく等しくない場合は条件付き転送
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move ifnot greater より小さいか等しい場合は条件付き転送 / より大きくない場合は条件付き転送
CMOVC	Conditional move if carry キャリーありの場合は条件付き転送
CMOVNC	Conditional move if not carry キャリーなしの場合は条件付き転送
CMOVO	Conditional move if overflow オーバーフローありの場合は条件付き転送

CMOVNO	Conditional move if not overflow オーバーフローなしの場合は条件付き転送
CMOVS	Conditional move if sign（negative） 符号付きの場合（負の場合）は条件付き転送
CMOVNS	Conditional move if not sign（non-negative） 符号なしの場合（負でない場合）は条件付き転送
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even パリティありの場合は条件付き転送 / 偶数パリティの場合は条件付き転送
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd パリティなしの場合は条件付き転送 / 奇数パリティの場合は条件付き転送
XCHG	Exchange 交換
BSWAP	Byte swap バイト・スワップ
XADD	Exchange and add 交換して加算
CMPXCHG	Compare and exchange 比較して交換
CMPXCHG8B	Compare and exchange 8 bytes 比較して 8 バイトを交換
PUSH	Push onto stack スタックにプッシュ
POP	Pop off of stack スタックからポップ
PUSHA/PUSHAD	Push general-purpose registers onto stack 汎用レジスタをスタックにプッシュ
POPA/POPAD	Pop general-purpose registers from stack スタックから汎用レジスタをポップ
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword ワードをダブルワードに変換 / ダブルワードをクワッドワードに変換
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register バイトをワードに変換 /EAX レジスタ内でワードをダブルワードに変換
MOVSX	Move and sign extend 転送して符号で拡張
MOVZX	Move and zero extend 転送してゼロで拡張

5.1.2. 2 進算術命令

2 進算術命令は、メモリまたは汎用レジスタ内のバイト整数、ワード整数、ダブルワード整数の基本的な 2 進整数計算を実行する。

ADD	Integer add 整数の加算
ADC	Add with carry キャリーあり加算
SUB	Subtract 減算
SBB	Subtract with borrow ボローあり減算
IMUL	Signed multiply 符号付き乗算
MUL	Unsigned multiply 符号なし乗算
IDIV	Signed divide 符号付き除算
DIV	Unsigned divide 符号なし除算
INC	Increment インクリメント
DEC	Decrement デクリメント
NEG	Negate 符号変更
CMP	Compare 比較

5.1.3. 10 進算術命令

10 進算術命令は、2 進化 10 進数（BCD）データの 10 進算術演算を実行する。

DAA	Decimal adjust after addition 加算後に 10 進調整
DAS	Decimal adjust after subtraction 減算後に 10 進調整
AAA	ASCII adjust after addition 加算後に ASCII 調整
AAS	ASCII adjust after subtraction 減算後に ASCII 調整
AAM	ASCII adjust after multiplication 乗算後に ASCII 調整
AAD	ASCII adjust before division 除算前に ASCII 調整

5.1.4. 論理命令

論理命令は、バイト値、ワード値、ダブルワード値の基本的な AND、OR、XOR、NOT 論理演算を実行する。

AND	Perform bitwise logical AND ビットごとの AND（論理積）演算を実行
OR	Perform bitwise logical OR ビットごとの OR（論理和）演算を実行
XOR	Perform bitwise logical exclusive OR ビットごとの XOR（排他的論理和）演算を実行
NOT	Perform bitwise logical NOT ビットごとの NOT（否定論理）演算を実行

5.1.5. シフト命令とローテート命令

シフト命令とローテート命令は、ワード・オペランドおよびダブルワード・オペランド内のビットのシフトとローテートを実行する。

SAR	Shift arithmetic right 算術右シフト
SHR	Shift logical right 論理右シフト
SAL/SHL	Shift arithmetic left/Shift logical left 算術左シフト / 論理左シフト
SHRD	Shift right double 右ダブルシフト

SHLD	Shift left double 左ダブルシフト
ROR	Rotate right 右ローテート
ROL	Rotate left 左ローテート
RCR	Rotate through carry right キャリーを通した右ローテート
RCL	Rotate through carry left キャリーを通した左ローテート

5.1.6. ビット命令とバイト命令

ビット命令は、ワード・オペランドおよびダブルワード・オペランド内の個々のビットをテストし、変更する。バイト命令は、EFLAGS レジスタ内のフラグの状態を示すように、バイト・オペランドの値を設定する。

BT	Bit test ビットのテスト
BTS	Bit test and set ビットのテストおよびセット
BTR	Bit test and reset ビットのテストおよびリセット
BTC	Bit test and complement ビットのテストおよび補数
BSF	Bit scan forward 上位の方向にビットスキャン
BSR	Bit scan reverse 下位の方向にビットスキャン
SETE/SETZ	Set byte if equal/Set byte if zero 等しい場合はバイトをセット / ゼロの場合はバイトをセット
SETNE/SETNZ	Set byte if not equal/Set byte if not zero 等しくない場合はバイトをセット / ゼロでない場合はバイトをセット
SETA/SETNBE	Set byte if above/Set byte if not below or equal より大きい場合はバイトをセット / より小さくなく等しくない場合はバイトをセット
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry より大きいか等しい場合はバイトをセット / より小さくない場合はバイトをセット / キャリーなしの場合はバイトをセット
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry より小さい場合はバイトをセット / より大きくなく等しくない場合はバイトをセット / キャリーありの場合はバイトをセット

SETBE/SETNA	Set byte if below or equal/Set byte if not above より小さいか等しい場合はバイトをセット / より大きくない場合はバイトをセット
SETG/SETNLE	Set byte if greater/Set byte if not less or equal より大きい場合はバイトをセット / より小さくなく等しくない場合はバイトをセット
SETGE/SETNL	Set byte if greater or equal/Set byte if not less より大きいか等しい場合はバイトをセット / より小さくない場合はバイトをセット
SETL/SETNGE	Set byte if less/Set byte if not greater or equal より小さい場合はバイトをセット / より大きくなく等しくない場合はバイトをセット
SETLE/SETNG	Set byte if less or equal/Set byte if not greater より小さいか等しい場合はバイトをセット / より大きくない場合はバイトをセット
SETS	Set byte if sign（negative） 符号付きの場合（負の場合）はバイトをセット
SETNS	Set byte if not sign（non-negative） 符号なしの場合（負でない場合）はバイトをセット
SETO	Set byte if overflow オーバーフローありの場合はバイトをセット
SETNO	Set byte if not overflow オーバーフローなしの場合はバイトをセット
SETPE/SETP	Set byte if parity even/Set byte if parity 偶数パリティの場合はバイトをセット / パリティありの場合はバイトをセット
SETPO/SETNP	Set byte if parity odd/Set byte if not parity 奇数パリティの場合はバイトをセット / パリティなしの場合はバイトをセット
TEST	Logical compare 論理比較

5.1.7. 制御転送命令

制御転送命令は、ジャンプ、条件付きジャンプ、ループ、コールとリターンの操作を実行して、プログラム・フローを制御する。

JMP	Jump ジャンプ
JE/JZ	Jump if equal/Jump if zero 等しい場合はジャンプ / ゼロの場合はジャンプ
JNE/JNZ	Jump if not equal/Jump if not zero 等しくない場合はジャンプ / ゼロでない場合はジャンプ

JA/JNBE	Jump if above/Jump if not below or equal より大きい場合はジャンプ / より小さくなく等しくない場合はジャンプ
JAE/JNB	Jump if above or equal/Jump if not below より大きいか等しい場合はジャンプ / より小さくない場合はジャンプ
JB/JNAE	Jump if below/Jump if not above or equal より小さい場合はジャンプ / より大きくなく等しくない場合はジャンプ
JBE/JNA	Jump if below or equal/Jump if not above より小さいか等しい場合はジャンプ / より大きくない場合はジャンプ
JG/JNLE	Jump if greater/Jump if not less or equal より大きい場合はジャンプ / より小さくなく等しくない場合はジャンプ
JGE/JNL	Jump if greater or equal/Jump if not less より大きいか等しい場合はジャンプ / より小さくない場合はジャンプ
JL/JNGE	Jump if less/Jump if not greater or equal より小さい場合はジャンプ / より大きくなく等しくない場合はジャンプ
JLE/JNG	Jump if less or equal/Jump if not greater より小さいか等しい場合はジャンプ / より大きくない場合はジャンプ
JC	Jump if carry キャリーありの場合はジャンプ
JNC	Jump if not carry キャリーなしの場合はジャンプ
JO	Jump if overflow オーバーフローありの場合はジャンプ
JNO	Jump if not overflow オーバーフローなしの場合はジャンプ
JS	Jump if sign（negative） 符号付きの場合（負の場合）はジャンプ
JNS	Jump if not sign（non-negative） 符号なしの場合（負でない場合）はジャンプ
JPO/JNP	Jump if parity odd/Jump if not parity 奇数パリティの場合はジャンプ / パリティなしの場合はジャンプ
JPE/JP	Jump if parity even/Jump if parity 偶数パリティの場合はジャンプ / パリティありの場合はジャンプ
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero CX レジスタがゼロの場合はジャンプ /ECX レジスタがゼロの場合はジャンプ

LOOP	Loop with ECX counter ECX をカウンタとしてループ
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal ECX をカウンタに、ゼロの場合はループ /ECX をカウンタに、等しい場合はループ
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal ECX をカウンタに、ゼロでない場合はループ /ECX をカウンタに、等しくない場合はループ
CALL	Call procedure プロシージャの呼び出し
RET	Return 戻る
IRET	Return from interrupt 割り込みから戻る
INT	Software interrupt ソフトウェア割り込み
INTO	Interrupt on overflow オーバーフローで割り込み
BOUND	Detect value out of range 範囲外の値を検出
ENTER	High-level procedure entry 高度なプロシージャの開始
LEAVE	High-level procedure exit 高度なプロシージャの終了

5.1.8. ストリング命令

ストリング命令は、バイト・ストリングを操作し、メモリとの間で転送する。

MOVS/MOVSB	Move string/Move byte string ストリングを転送 / バイト・ストリングを転送
MOVS/MOVSW	Move string/Move word string ストリングを転送 / ワード・ストリングを転送
MOVS/MOVSD	Move string/Move doubleword string ストリングを転送 / ダブルワード・ストリングを転送
CMPS/CMPSB	Compare string/Compare byte string ストリングを比較 / バイト・ストリングを比較
CMPS/CMPSW	Compare string/Compare word string ストリングを比較 / ワード・ストリングを比較
CMPS/CMPSD	Compare string/Compare doubleword string ストリングを比較 / ダブルワード・ストリングを比較
SCAS/SCASB	Scan string/Scan byte string ストリングをスキャン / バイト・ストリングをスキャン
SCAS/SCASW	Scan string/Scan word string ストリングをスキャン / ワード・ストリングをスキャン
SCAS/SCASD	Scan string/Scan doubleword string ストリングをスキャン / ダブルワード・ストリングをスキャン
LODS/LODSB	Load string/Load byte string ストリングをロード / バイト・ストリングをロード
LODS/LODSW	Load string/Load word string ストリングをロード / ワード・ストリングをロード
LODS/LODSD	Load string/Load doubleword string ストリングをロード / ダブルワード・ストリングをロード
STOS/STOSB	Store string/Store byte string ストリングをストア / バイト・ストリングをストア
STOS/STOSW	Store string/Store word string ストリングをストア / ワード・ストリングをストア
STOS/STOSD	Store string/Store doubleword string ストリングをストア / ダブルワード・ストリングをストア
REP	Repeat while ECX not zero ECX がゼロでない間は反復
REPE/REPZ	Repeat while equal/Repeat while zero 等しい間は反復 / ゼロの間は反復
REPNE/REPNZ	Repeat while not equal/Repeat while not zero 等しくない間は反復 / ゼロでない間は反復

5.1.9. I/O 命令

I/O 命令は、プロセッサの I/O ポートと、レジスタまたはメモリの間でデータを転送する。

IN	Read from a port ポートから読み込み
OUT	Write to a port ポートへ書き込み
INS/INSB	Input string from port/Input byte string from port ポートからストリングを入力 / ポートからバイト・ストリングを入力
INS/INSW	Input string from port/Input word string from port ポートからストリングを入力 / ポートからワード・ストリングを入力
INS/INSD	Input string from port/Input doubleword string from port ポートからストリングを入力 / ポートからダブルワード・ストリングを入力
OUTS/OUTSB	Output string to port/Output byte string to port ストリングをポートに出力 / バイト・ストリングをポートに出力
OUTS/OUTSW	Output string to port/Output word string to port ストリングをポートに出力 / ワード・ストリングをポートに出力
OUTS/OUTSD	Output string to port/Output doubleword string to port ストリングをポートに出力 / ダブルワード・ストリングをポートに出力

5.1.10. ENTER 命令と LEAVE 命令

ENTER 命令と LEAVE 命令は、ブロック構造言語でのプロシージャ・コールに対してマシン語をサポートする。

ENTER	High-level procedure entry 高度なプロシージャの開始
LEAVE	High-level procedure exit 高度なプロシージャの終了

5.1.11. フラグ制御（EFLAG）命令

フラグ制御命令は、EFLAGS レジスタ内のフラグを操作する。

STC	Set carry flag キャリーフラグをセット
CLC	Clear the carry flag キャリーフラグをクリア

CMC	Complement the carry flag キャリーフラグに補数を設定
CLD	Clear the direction flag 方向フラグをクリア
STD	Set direction flag 方向フラグをセット
LAHF	Load flags into AH register フラグを AH レジスタにロード
SAHF	Store AH register into flags AH レジスタをフラグにストア
PUSHF/PUSHFD	Push EFLAGS onto stack EFLAGS をスタックにプッシュ
POPF/POPFD	Pop EFLAGS from stack スタックから EFLAGS をポップ
STI	Set interrupt flag 割り込みフラグをセット
CLI	Clear the interrupt flag 割り込みフラグをクリア

5.1.12. セグメント・レジスタ命令

セグメント・レジスタ命令は、セグメント・レジスタ内に far ポインタ（セグメント・アドレス）をロードする。

LDS	Load far pointer using DS DS を使用して far ポインタをロード
LES	Load far pointer using ES ES を使用して far ポインタをロード
LFS	Load far pointer using FS FS を使用して far ポインタをロード
LGS	Load far pointer using GS GS を使用して far ポインタをロード
LSS	Load far pointer using SS SS を使用して far ポインタをロード

5.1.13. その他の命令

その他の命令は、実効アドレスのロード、「非操作（no-operation）」の実行、プロセッサ識別情報の取得などの機能を持つ。

LEA	Load effective address 実効アドレスをロード
NOP	No operation 非操作
UD2	Undefined instruction 未定義命令
XLAT/XLATB	Table lookup translation テーブル・ルックアップの変換
CPUID	Processor Identification プロセッサ識別

5.2. x87 FPU 命令

x87 FPU 命令は、プロセッサの x87 FPU によって実行される。これらの命令は、浮動小数点オペランド、整数オペランド、2 進化 10 進数（BCD）オペランドを操作する。x87 FPU 命令についての詳細は、第 8 章「x87 FPU によるプログラミング」を参照のこと。

この命令は、データ転送命令、定数ロード命令、FPU 制御命令の各サブグループに分けられる。以下の各項では、各サブグループについて説明する。

5.2.1. x87 FPU データ転送命令

データ転送命令は、メモリと x87 FPU レジスタの間で、浮動小数点値、整数値、BCD 値を転送する。また、データ転送命令は、浮動小数点オペランドの条件付き転送操作も実行する。

FLD	Load floating-point value 浮動小数点値をロード
FST	Store floating-point value 浮動小数点値をストア
FSTP	Store floating-point value and pop 浮動小数点値をストアしてポップ
FILD	Load integer 整数をロード
FIST	Store integer 整数をストア

FISTP[1]	Store integer and pop 整数をストアしてポップ
FBLD	Load BCD BCD をロード
FBSTP	Store BCD and pop BCD をストアしてポップ
FXCH	Exchange registers レジスタを交換
FCMOVE	Floating-point conditional move if equal 等しい場合は浮動小数点値の条件付き転送
FCMOVNE	Floating-point conditional move if not equal 等しくない場合は浮動小数点値の条件付き転送
FCMOVB	Floating-point conditional move if below より小さい場合は浮動小数点値の条件付き転送
FCMOVBE	Floating-point conditional move if below or equal より小さいか等しい場合は浮動小数点値の条件付き転送
FCMOVNB	Floating-point conditional move if not below より小さくない場合は浮動小数点値の条件付き転送
FCMOVNBE	Floating-point conditional move if not below or equal より小さくなく等しくない場合は浮動小数点値の条件付き転送
FCMOVU	Floating-point conditional move if unordered 順序化不可能の場合は浮動小数点値の条件付き転送
FCMOVNU	Floating-point conditional move if not unordered 順序化不可能でない場合は浮動小数点値の条件付き転送

5.2.2. x87 FPU 基本算術命令

基本算術命令は、浮動小数点オペランドと整数オペランドの基本算術演算を実行する。

FADD	Add floating-point 浮動小数点値を加算
FADDP	Add floating-point and pop 浮動小数点値を加算してポップ
FIADD	Add integer 整数を加算
FSUB	Subtract floating-point 浮動小数点値を減算
FSUBP	Subtract floating-point and pop 浮動小数点値を減算してポップ

1. SSE3 では、整数変換用に FISTTP 命令を提供。

FISUB	Subtract integer 整数を減算
FSUBR	Subtract floating-point reverse 浮動小数点値を逆減算
FSUBRP	Subtract floating-point reverse and pop 浮動小数点値を逆減算してポップ
FISUBR	Subtract integer reverse 整数を逆減算
FMUL	Multiply floating-point 浮動小数点値を乗算
FMULP	Multiply floating-point and pop 浮動小数点値を乗算してポップ
FIMUL	Multiply integer 整数を乗算
FDIV	Divide floating-point 浮動小数点値を除算
FDIVP	Divide floating-point and pop 浮動小数点値を除算してポップ
FIDIV	Divide integer 整数を除算
FDIVR	Divide floating-point reverse 浮動小数点値を逆除算
FDIVRP	Divide floating-point reverse and pop 浮動小数点値を逆除算してポップ
FIDIVR	Divide integer reverse 整数を逆除算
FPREM	Partial remainder 部分剰余
FPREM1	IEEE Partial remainder IEEE 部分剰余
FABS	Absolute value 絶対値
FCHS	Change sign 符号を変更
FRNDINT	Round to integer 整数への丸め
FSCALE	Scale by power of two 2 のべき乗でスケーリング
FSQRT	Square root 平方根
FXTRACT	Extract exponent and significand 指数部と仮数部を抽出

5.2.3. x87 FPU 比較命令

比較命令は、浮動小数点オペランドまたは整数オペランドのチェックまたは比較を実行する。

FCOM	Compare floating-point 浮動小数点値を比較
FCOMP	Compare floating-point and pop 浮動小数点値を比較してポップ
FCOMPP	Compare floating-point and pop twice 浮動小数点値を比較して 2 回ポップ
FUCOM	Unordered compare floating-point 順序化不可能条件付きで浮動小数点値を比較
FUCOMP	Unordered compare floating-point and pop 順序化不可能条件付きで浮動小数点値を比較してポップ
FUCOMPP	Unordered compare floating-point and pop twice 順序化不可能条件付きで浮動小数点値を比較して 2 回ポップ
FICOM	Compare integer 整数を比較
FICOMP	Compare integer and pop 整数を比較してポップ
FCOMI	Compare floating-point and set EFLAGS 浮動小数点値を比較して EFLAGS をセット
FUCOMI	Unordered compare floating-point and set EFLAGS 順序化不可能条件付きで浮動小数点値を比較して EFLAGS をセット
FCOMIP	Compare floating-point, set EFLAGS, and pop 浮動小数点値を比較し、EFLAGS をセットしてポップ
FUCOMIP	Unordered compare floating-point, set EFLAGS, and pop 順序化不可能条件付きで浮動小数点値を比較し、EFLAGS をセットしてポップ
FTST	Test floating-point 浮動小数点値をテスト
FXAM	Examine floating-point 浮動小数点値を検査

5.2.4. x87 FPU 超越関数命令

超越関数命令は、浮動小数点オペランドの基本的な三角関数演算と対数演算を実行する。

FSIN	Sine 正弦

FCOS	Cosine 余弦
FSINCOS	Sine and cosine 正弦と余弦
FPTAN	Partial tangent 部分正接
FPATAN	Partial arctangent 部分逆正接
F2XM1	$2^x - 1$
FYL2X	$y*\log_2 x$
FYL2XP1	$y*\log_2(x+1)$

5.2.5. x87 FPU 定数ロード命令

定数ロード命令は、π などの一般的な定数を x87 FPU レジスタにロードする。

FLD1	Load +1.0 +1.0 をロード
FLDZ	Load +0.0 +0.0 をロード
FLDPI	Load π π をロード
FLDL2E	Load $\log_2 e$ $\log_2 e$ をロード
FLDLN2	Load $\log_e 2$ $\log_e 2$ をロード
FLDL2T	Load $\log_2 10$ $\log_2 10$ をロード
FLDLG2	Load $\log_{10} 2$ $\log_{10} 2$ をロード

5.2.6. x87 FPU 制御命令

x87 FPU 制御命令は、x87 FPU レジスタスタックを操作し、x87 FPU ステートのセーブとリストアを行う。

FINCSTP	Increment FPU register stack pointer FPU レジスタのスタックポインタをインクリメント
FDECSTP	Decrement FPU register stack pointer FPU レジスタのスタックポインタをデクリメント
FFREE	Free floating-point register 浮動小数点レジスタを解放

FINIT	Initialize FPU after checking error conditions エラー条件をチェックしてから FPU を初期化
FNINIT	Initialize FPU without checking error conditions エラー条件をチェックせずに FPU を初期化
FCLEX	Clear floating-point exception flags after checking for error conditions エラー条件をチェックしてから浮動小数点例外フラグをクリア
FNCLEX	Clear floating-point exception flags without checking for error conditions エラー条件をチェックせずに浮動小数点例外フラグをクリア
FSTCW	Store FPU control word after checking error conditions エラー条件をチェックしてから FPU 制御ワードをストア
FNSTCW	Store FPU control word without checking error conditions エラー条件をチェックせずに FPU 制御ワードをストア
FLDCW	Load FPU control word FPU 制御ワードをロード
FSTENV	Store FPU environment after checking error conditions エラー条件をチェックしてから FPU 環境をストア
FNSTENV	Store FPU environment without checking error conditions エラー条件をチェックせずに FPU 環境をストア
FLDENV	Load FPU environment FPU 環境をロード
FSAVE	Save FPU state after checking error conditions エラー条件をチェックしてから FPU ステートをセーブ
FNSAVE	Save FPU state without checking error conditions エラー条件をチェックせずに FPU ステートをセーブ
FRSTOR	Restore FPU state FPU ステートをリストア
FSTSW	Store FPU status word after checking error conditions エラー条件をチェックしてから FPU ステータス・ワードをストア
FNSTSW	Store FPU status word without checking error conditions エラー条件をチェックせずに FPU ステータス・ワードをストア
WAIT/FWAIT	Wait for FPU FPU を待機
FNOP	FPU no operation FPU の非操作

5.3. x87 FPU および SIMD ステートの管理命令

インテル® Pentium® II プロセッサ・ファミリで、2 つのステート管理命令が IA-32 アーキテクチャに追加された。

FXSAVE	x87 FPU および SIMD ステートをセーブする。
FXRSTOR	x87 FPU および SIMD ステートをリストアする。

最初は、これらの命令は、x87 FPU（および MMX® テクノロジ）レジスタだけを操作して、x87 FPU および MMX テクノロジ・ステートの高速セーブとリストアを実行するものであった。インテル® Pentium® III プロセッサ・ファミリで SSE が導入されると、これらの命令は、XMM レジスタと MXCSR レジスタの状態のセーブとリストアも実行するように拡張された。

詳細は、10.5. 節「FXSAVE 命令と FXRSTOR 命令」を参照のこと。

5.4. MMX® 命令

4 つの拡張命令が IA-32 アーキテクチャに導入され、IA-32 プロセッサは SIMD（Single Instruction, Multiple Data）演算を実行できるようになった。この拡張命令とは、MMX® テクノロジ、SSE、SSE2、SSE3 である。SIMD 命令の歴史的な経緯については、2.3. 節「SIMD 命令」を参照のこと。

MMX 命令は、MMX テクノロジ・レジスタまたは汎用レジスタ内のメモリでパックドバイト、パックドワード、パックド・ダブルワード、またはクワッドワード整数オペランドを操作する。これらの命令の詳細は、第 9 章「インテル® MMX® テクノロジによるプログラミング」を参照のこと。

MMX 命令は、MMX テクノロジをサポートする IA-32 プロセッサ上でのみ実行できる。プロセッサが MMX 命令をサポートしているかどうかは、CPUID 命令によって検出できる。『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の CPUID 命令の説明を参照のこと。

MMX 命令は、データ転送命令、変換命令、パックド算術命令、比較命令、論理演算命令、シフト命令、ローテート命令、ステート管理命令というサブグループに分けられる。以下の各項では、各サブグループについて説明する。

5.4.1. MMX® テクノロジ・データ転送命令

データ転送命令は、MMX®テクノロジ・レジスタ同士およびMMXテクノロジ・レジスタとメモリの間で、ダブルワードおよびクワッドワード・オペランドを転送する。

MOVD	Move doubleword. ダブルワードを転送
MOVQ	Move quadword. クワッドワードを転送

5.4.2. MMX® テクノロジ変換命令

変換命令は、バイト、ワード、ダブルワードのパックとアンパックを実行する。

PACKSSWB	Pack words into bytes with signed saturation. 符号付き飽和演算によりワードをバイトにパック
PACKSSDW	Pack doublewords into words with signed saturation. 符号付き飽和演算によりダブルワードをワードにパック
PACKUSWB	Pack words into bytes with unsigned saturation. 符号なし飽和演算によりワードをバイトにパック
PUNPCKHBW	Unpack high-order bytes. 上位バイトをアンパック
PUNPCKHWD	Unpack high-order words. 上位ワードをアンパック
PUNPCKHDQ	Unpack high-order doublewords. 上位ダブルワードをアンパック
PUNPCKLBW	Unpack low-order bytes. 下位バイトをアンパック
PUNPCKLWD	Unpack low-order words. 下位ワードをアンパック
PUNPCKLDQ	Unpack low-order doublewords. 下位ダブルワードをアンパック

5.4.3. MMX® テクノロジ・パックド算術命令

パックド算術命令は、パックドバイト、パックドワード、パックド・ダブルワード整数のパックド算術演算を実行する。

PADDB	Add packed byte integers. パックドバイト整数を加算
PADDW	Add packed word integers. パックドワード整数を加算
PADDD	Add packed doubleword integers. パックド・ダブルワード整数を加算
PADDSB	Add packed signed byte integers with signed saturation. 符号付き飽和演算によりパックド符号付きバイト整数を加算
PADDSW	Add packed signed word integers with signed saturation. 符号付き飽和演算によりパックド符号付きワード整数を加算
PADDUSB	Add packed unsigned byte integers with unsigned saturation. 符号なし飽和演算によりパックド符号なしバイト整数を加算
PADDUSW	Add packed unsigned word integers with unsigned saturation. 符号なし飽和演算によりパックド符号なしワード整数を加算
PSUBB	Subtract packed byte integers. パックドバイト整数を減算
PSUBW	Subtract packed word integers. パックドワード整数を減算
PSUBD	Subtract packed doubleword integers. パックド・ダブルワード整数を減算
PSUBSB	Subtract packed signed byte integers with signed saturation. 符号付き飽和演算によりパックド符号付きバイト整数を減算
PSUBSW	Subtract packed signed word integers with signed saturation. 符号付き飽和演算によりパックド符号付きワード整数を減算
PSUBUSB	Subtract packed unsigned byte integers with unsigned saturation. 符号なし飽和演算によりパックド符号なしバイト整数を減算
PSUBUSW	Subtract packed unsigned word integers with unsigned saturation. 符号なし飽和演算によりパックド符号符なしワード整数を減算
PMULHW	Multiply packed signed word integers and store high result. パックド符号付きワード整数を乗算して上位結果をストア
PMULLW	Multiply packed signed word integers and store low result. パックド符号付きワード整数を乗算して下位結果をストア
PMADDWD	Multiply and add packed word integers. パックドワード整数を乗算および加算

5.4.4. MMX® テクノロジ比較命令

比較命令は、パックドバイト、パックドワード、またはパックド・ダブルワードの比較を実行する。

PCMPEQB	Compare packed bytes for equal. パックドバイトを比較し、一致しているか判定
PCMPEQW	Compare packed words for equal. パックドワードを比較し、一致しているか判定
PCMPEQD	Compare packed doublewords for equal. パックド・ダブルワードを比較し、一致しているか判定
PCMPGTB	Compare packed signed byte integers for greater than. パックド符号付きバイト整数を比較し、大小関係を判定
PCMPGTW	Compare packed signed word integers for greater than. パックド符号付きワード整数を比較し、大小関係を判定
PCMPGTD	Compare packed signed doubleword integers for greater than. パックド符号付きダブルワード整数を比較し、大小関係を判定

5.4.5. MMX® テクノロジ論理演算命令

論理演算命令は、クワッドワード・オペランドのAND、AND NOT、OR、およびXOR演算を実行する。

PAND	Bitwise logical AND. ビットごとのAND（論理積）演算
PANDN	Bitwise logical AND NOT. ビットごとのAND NOT（否定論理積）演算
POR	Bitwise logical OR. ビットごとのOR（論理和）演算
PXOR	Bitwise logical exclusive OR. ビットごとのXOR（排他的論理和）演算

5.4.6. MMX® テクノロジ・シフト命令とローテート命令

シフト命令とローテート命令は、64ビット・オペランド内のパックドバイト、パックドワード、パックド・ダブルワード、またはクワッドワードのシフトとローテートを実行する。

PSLLW	Shift packed words left logical. パックドワードを論理左シフト
PSLLD	Shift packed doublewords left logical. パックド・ダブルワードを論理左シフト
PSLLQ	Shift packed quadword left logical. パックド・クワッドワードを論理左シフト

PSRLW	Shift packed words right logical. パックドワードを論理右シフト
PSRLD	Shift packed doublewords right logical. パックド・ダブルワードを論理右シフト
PSRLQ	Shift packed quadword right logical. パックド・クワッドワードを論理右シフト
PSRAW	Shift packed words right arithmetic. パックドワードを算術右シフト
PSRAD	Shift packed doublewords right arithmetic. パックド・ダブルワードを算術右シフト

5.4.7. MMX® テクノロジ・ステート管理

EMMS 命令は、MMX® テクノロジ・レジスタから MMX テクノロジ・ステートをクリアする。

EMMS	Empty MMX state. MMX ステートをクリア

5.5. SSE

SSE は、MMX® テクノロジで導入された SIMD 実行モデルを拡張したものである。SSE についての詳細は、第 10 章「ストリーミング SIMD 拡張命令（SSE）によるプログラミング」を参照のこと。

SSE は、SSE をサポートする IA-32 プロセッサ上でのみ実行できる。プロセッサが SSE をサポートしているかどうかは、CPUID 命令によって検出できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の CPUID 命令の説明を参照）。

SSE は、以下の 4 つのサブグループに分けられる（最初のサブグループは下位のサブグループを持つことに注意）。

- XMM レジスタを操作する、SIMD 単精度浮動小数点命令
- MXSCR ステート管理命令
- MMX テクノロジ・レジスタを操作する、64 ビット SIMD 整数命令
- キャッシュ制御命令、プリフェッチ命令、および命令順序付け命令

以下の各項では、これらのグループの概要について述べる。

5.5.1. SSE SIMD 単精度浮動小数点命令

これらの命令は、XMM レジスタまたはメモリ内のパックド/スカラ単精度浮動小数点値を操作する。このサブグループは、さらに下位のサブグループであるデータ転送命令、パックド算術命令、比較命令、論理演算命令、シャッフル命令とアンパック命令、変換命令に分けられる。

5.5.1.1. SSE データ転送命令

SSE データ転送命令は、XMM レジスタ同士および XMM レジスタとメモリの間で、パックド/スカラ単精度浮動小数点オペランドを転送する。

MOVAPS	Move four aligned packed single-precision floating-point values between XMM registers or between and XMM register and memory. アライメントの合った 4 つのパックド単精度浮動小数点値を XMM レジスタ同士の間、または XMM レジスタとメモリとの間で転送
MOVUPS	Move four unaligned packed single-precision floating-point values between XMM registers or between and XMM register and memory. アライメントの合っていない 4 つのパックド単精度浮動小数点値を XMM レジスタ同士の間、または XMM レジスタとメモリとの間で転送
MOVHPS	Move two packed single-precision floating-point values to an from the high quadword of an XMM register and memory. 2 つのパックド単精度浮動小数点値を XMM レジスタの上位クワッドワードとメモリとの間で転送
MOVHLPS	Move two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register. 2 つのパックド単精度浮動小数点値を XMM レジスタの上位クワッドワードから別の XMM レジスタの下位クワッドワードに転送
MOVLPS	Move two packed single-precision floating-point values to an from the low quadword of an XMM register and memory. 2 つのパックド単精度浮動小数点値を XMM レジスタの下位クワッドワードとメモリとの間で転送
MOVLHPS	Move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register. 2 つのパックド単精度浮動小数点値を XMM レジスタの下位クワッドワードから別の XMM レジスタの上位クワッドワードに転送
MOVMSKPS	Extract sign mask from four packed single-precision floating-point values. 4 つのパックド単精度浮動小数点値から符号マスクを抽出

MOVSS	Move scalar single-precision floating-point value between XMM registers or between an XMM register and memory. スカラ単精度浮動小数点値を XMM レジスタ同士の間、または XMM レジスタとメモリとの間で転送

5.5.1.2. SSE パックド算術命令

SSE パックド算術命令は、パックド / スカラ単精度浮動小数点オペランドに対する、パックド / スカラ算術演算を実行する。

ADDPS	Add packed single-precision floating-point values. パックド単精度浮動小数点値を加算
ADDSS	Add scalar single-precision floating-point values. スカラ単精度浮動小数点値を加算
SUBPS	Subtract packed single-precision floating-point values. パックド単精度浮動小数点値を減算
SUBSS	Subtract scalar single-precision floating-point values. スカラ単精度浮動小数点値を減算
MULPS	Multiply packed single-precision floating-point values. パックド単精度浮動小数点値を乗算
MULSS	Multiply scalar single-precision floating-point values. スカラ単精度浮動小数点値を乗算
DIVPS	Divide packed single-precision floating-point values. パックド単精度浮動小数点値を除算
DIVSS	Divide scalar single-precision floating-point values. スカラ単精度浮動小数点値を除算
RCPPS	Compute reciprocals of packed single-precision floating-point values. パックド単精度浮動小数点値の逆数を計算
RCPSS	Compute reciprocal of scalar single-precision floating-point values. スカラ単精度浮動小数点値の逆数を計算
SQRTPS	Compute square roots of packed single-precision floating-point values. パックド単精度浮動小数点値の平方根を計算
SQRTSS	Compute square root of scalar single-precision floating-point values. スカラ単精度浮動小数点値の平方根を計算
RSQRTPS	Compute reciprocals of square roots of packed single-precision floating-point values. パックド単精度浮動小数点値の平方根の逆数を計算
RSQRTSS	Compute reciprocal of square root of scalar single-precision floating-point values. スカラ単精度浮動小数点値の平方根の逆数を計算
MAXPS	Return maximum packed single-precision floating-point values. パックド単精度浮動小数点値の最大値を返す
MAXSS	Return maximum scalar single-precision floating-point values. スカラ単精度浮動小数点値の最大値を返す

MINPS	Return minimum packed single-precision floating-point values. パック ド単精度浮動小数点値の最小値を返す
MINSS	Return minimum scalar single-precision floating-point values. スカラ単精度浮動小数点値の最小値を返す

5.5.1.3. SSE 比較命令

SSE比較命令は、パック ド/スカラ単精度浮動小数点オペランドの比較を実行する。

CMPPS	Compare packed single-precision floating-point values. パック ド単精度浮動小数点値を比較
CMPSS	Compare scalar single-precision floating-point values. スカラ単精度浮動小数点値を比較
COMISS	Perform ordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register. スカラ単精度浮動小数点値を順序付きで比較し、EFLAGS レジスタにフラグをセット
UCOMISS	Perform unordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register. スカラ単精度浮動小数点値を順序付けなしで比較し、EFLAGS レジスタにフラグをセット

5.5.1.4. SSE 論理演算命令

SSE 論理演算命令は、パック ド単精度浮動小数点オペランドのビット単位の AND、AND NOT、OR、またはXOR 演算を実行する。

ANDPS	Perform bitwise logical AND of packed single-precision floating-point values. パック ド単精度浮動小数点値のビットごとの AND（論理積）演算を実行
ANDNPS	Perform bitwise logical AND NOT of packed single-precision floating-point values. パック ド単精度浮動小数点値のビットごとの AND NOT（否定論理積）演算を実行
ORPS	Perform bitwise logical OR of packed single-precision floating-point values. パック ド単精度浮動小数点値のビットごとの OR（論理和）演算を実行
XORPS	Perform bitwise logical XOR of packed single-precision floating-point values. パック ド単精度浮動小数点値のビットごとの XOR（排他的論理和）演算を実行

5.5.1.5. SSE シャッフル命令とアンパック命令

SSE シャッフル命令とアンパック命令は、パックド単精度浮動小数点オペランド内の単精度浮動小数点値のシャッフルまたはインターリーブを実行する。

SHUFPS	Shuffles values in packed single-precision floating-point operands. パックド単精度浮動小数点オペランド内の値をシャッフル
UNPCKHPS	Unpacks and interleaves the two high-order values from two single-precision floating-point operands. 2 つの単精度浮動小数点オペランドから上位の値を 2 つアンパックしてインターリーブ
UNPCKLPS	Unpacks and interleaves the two low-order values from two single-precision floating-point operands. 2 つの単精度浮動小数点オペランドから下位の値を 2 つアンパックしてインターリーブ

5.5.1.6. SSE 変換命令

SSE 変換命令は、パックドまたは個々のダブルワード整数を、パックドまたはスカラ単精度浮動小数点値に変換する。あるいは、その逆方向の変換を行う。

CVTPI2PS	Convert packed doubleword integers to packed single-precision floating-point values. パックド・ダブルワード整数をパックド単精度浮動小数点値に変換
CVTSI2SS	Convert doubleword integer to scalar single-precision floating-point value. ダブルワード整数をスカラ単精度浮動小数点値に変換
CVTPS2PI	Convert packed single-precision floating-point values to packed doubleword integers. パックド単精度浮動小数点値をパックド・ダブルワード整数に変換
CVTTPS2PI	Convert with truncation packed single-precision floating-point values to packed doubleword integers. 切り捨てを使用して、パックド単精度浮動小数点値をパックド・ダブルワード整数に変換
CVTSS2SI	Convert scalar single-precision floating-point value to a doubleword integer. スカラ単精度浮動小数点値をダブルワード整数に変換
CVTTSS2SI	Convert with truncation scalar single-precision floating-point value to scalar doubleword integer. 切り捨てを使用して、スカラ単精度浮動小数点値をスカラ・ダブルワード整数に変換

5.5.2. SSE MXCSR ステート管理命令

MXCSR ステート管理命令は、MXCSR 制御およびステータス・レジスタのステートのセーブとリストアを実行する。

LDMXCSR	Load MXCSR register. MXCSR レジスタをロード
STMXCSR	Save MXCSR register state. MXCSR レジスタ・ステートをセーブ

5.5.3. SSE 64 ビット SIMD 整数命令

SSE 64 ビット SIMD 整数命令は、MMX® テクノロジ・レジスタ内のパックドバイト、パックドワード、またはパックド・ダブルワードに対する追加の演算を実行する。これらの命令は、5.4. 節「MMX® 命令」で説明した MMX 命令セットを拡張したものである。

PAVGB	Compute average of packed unsigned byte integers. パックド符号なしバイト整数の平均を計算
PAVGW	Compute average of packed unsigned byte integers. パックド符号なしワード整数の平均を計算
PEXTRW	Extract word. ワードを抽出
PINSRW	Insert word. ワードを挿入
PMAXUB	Maximum of packed unsigned byte integers. パックド符号なしバイト整数の最大値
PMAXSW	Maximum of packed signed word integers. パックド符号付きワード整数の最大値
PMINUB	Minimum of packed unsigned byte integers. パックド符号なしバイト整数の最小値
PMINSW	Minimum of packed signed word integers. パックド符号付きワード整数の最小値
PMOVMSKB	Move Byte Mask. バイトマスクを転送
PMULHUW	Multiply packed unsigned integers and store high result. パックド符号なし整数を乗算して上位結果をストア
PSADBW	Compute Sum of absolute differences. 絶対差の和を計算
PSHUFW	Shuffle packed integer word in MMX register. MMX レジスタ内のパックド整数ワードをシャッフル

5.5.4. SSE キャッシュ制御命令、プリフェッチ命令、および命令順序付け命令

キャッシュ制御命令は、MMX® テクノロジ・レジスタおよびXMM レジスタからメモリにデータをストアするときの非テンポラル・データのキャッシュ処理を制御する。PREFETCH*h* 命令は、選択されたキャッシュ・レベルにデータをプリフェッチする。SFENCE 命令は、ストア操作時の命令の順序を制御する。

MASKMOVQ	Non-temporal store of selected bytes from an MMX register into memory. MMX レジスタからメモリへの、選択したバイトの非テンポラルなストア
MOVNTQ	Non-temporal store of quadword from an MMX register into memory. MMX レジスタからメモリへの、クワッドワードの非テンポラルなストア
MOVNTPS	Non-temporal store of four packed single-precision floating-point values from an XMM register into memory. XMM レジスタからメモリへの、4 つのパックド単精度浮動小数点値の非テンポラルなストア
PREFETCH*h*	Load 32 or more of bytes from memory to a selected level of the processor's cache hierarchy. メモリ内の 32 バイト以上のデータを、プロセッサのキャッシュ階層内の選択されたレベルにロード
SFENCE	Serializes store operations. ストア操作をシリアル化

5.6. SSE2

SSE2 は、MMX® テクノロジと SSE で導入された SIMD 実行モデルを拡張したものである。SSE2 は、XMM レジスタ内の、パックド倍精度浮動小数点オペランドと、パックドバイト、パックドワード、パックド・ダブルワード、パックド・クワッドワード・オペランドを操作する。SSE2 についての詳細は、第 11 章「ストリーミング SIM D 拡張命令 2（SSE2）によるプログラミング」を参照のこと。

SSE2 は、SSE2 をサポートする IA-32 プロセッサ上でのみ実行できる。プロセッサが SSE2 をサポートしているかどうかは、CPUID 命令によって検出できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の CPUID 命令の説明を参照）。

これらの命令は、以下の 4 つのサブグループに分けられる（最初のサブグループは下位のサブグループを持つことに注意）。

- パックドおよびスカラ倍精度浮動小数点命令
- パックド単精度浮動小数点変換命令
- 128 ビット SIMD 整数命令

- キャッシュ制御命令と命令順序付け命令

以下の各項では、各グループの命令の概要について説明する。

5.6.1. SSE2 パックドおよびスカラ倍精度浮動小数点命令

SSE2パックドおよびスカラ倍精度浮動小数点命令は､倍精度浮動小数点オペランドのデータ転送、算術演算、比較演算、変換、論理演算、シャッフルの下位のサブグループに分けられる。これらの命令については、以下の各項で説明する。

5.6.1.1. SSE2 データ転送命令

SSE2 データ転送命令は、XMM レジスタ同士および XMM レジスタとメモリの間で、倍精度浮動小数点データを転送する。

MOVAPD — Move two aligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
アライメントの合った 2 つのパックド倍精度浮動小数点値を XMM レジスタ同士の間、または XMM レジスタとメモリとの間で転送

MOVUPD — Move two unaligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
アライメントの合っていない 2 つのパックド倍精度浮動小数点値を XMM レジスタ同士の間、または XMM レジスタとメモリとの間で転送

MOVHPD — Move high packed double-precision floating-point value to an from the high quadword of an XMM register and memory.
上位パックド倍精度浮動小数点値を XMM レジスタの上位クワッドワードとメモリとの間で転送

MOVLPD — Move low packed single-precision floating-point value to an from the low quadword of an XMM register and memory.
下位パックド倍精度浮動小数点値を XMM レジスタの下位クワッドワードとメモリとの間で転送

MOVMSKPD — Extract sign mask from two packed double-precision floating-point values.
2 つのパックド倍精度浮動小数点値から符号マスクを抽出

MOVSD — Move scalar double-precision floating-point value between XMM registers or between an XMM register and memory.
スカラ倍精度浮動小数点値を XMM レジスタ同士の間、または XMM レジスタとメモリとの間で転送

5.6.1.2. SSE2 パックド算術命令

SSE2 パックド算術命令は、パックド / スカラ倍精度浮動小数点オペランドに対する、加算、減算、乗算、除算、平方根、最大値 / 最小値演算を実行する。

ADDPD	Add packed double-precision floating-point values. パックド倍精度浮動小数点値を加算
ADDSD	Add scalar double precision floating-point values. スカラ倍精度浮動小数点値を加算
SUBPD	Subtract scalar double-precision floating-point values. パックド倍精度浮動小数点値を減算
SUBSD	Subtract scalar double-precision floating-point values. スカラ倍精度浮動小数点値を減算
MULPD	Multiply packed double-precision floating-point values. パックド倍精度浮動小数点値を乗算
MULSD	Multiply scalar double-precision floating-point values. スカラ倍精度浮動小数点値を乗算
DIVPD	Divide packed double-precision floating-point values. パックド倍精度浮動小数点値を除算
DIVSD	Divide scalar double-precision floating-point values. スカラ倍精度浮動小数点値を除算
SQRTPD	Compute packed square roots of packed double-precision floating-point values. パックド倍精度浮動小数点値のパックド平方根を計算
SQRTSD	Compute scalar square root of scalar double-precision floating-point value. スカラ倍精度浮動小数点値のスカラ平方根を計算
MAXPD	Return maximum packed double-precision floating-point values. パックド倍精度浮動小数点値の最大値を返す
MAXSD	Return maximum scalar double-precision floating-point value. スカラ倍精度浮動小数点値の最大値を返す
MINPD	Return minimum packed double-precision floating-point values. パックド倍精度浮動小数点値の最小値を返す
MINSD	Return minimum scalar double-precision floating-point value. スカラ倍精度浮動小数点値の最小値を返す

5.6.1.3. SSE2 論理演算命令

SSE2 論理演算命令は、パック ド倍精度浮動小数点値の AND、AND NOT、OR、XOR 演算を実行する。

ANDPD	Perform bitwise logical AND of packed double-precision floating-point values. パック ド倍精度浮動小数点値のビットごとの AND（論理積）演算を実行
ANDNPD	Perform bitwise logical AND NOT of packed double-precision floating-point values. パック ド倍精度浮動小数点値のビットごとの AND NOT（否定論理積）演算を実行
ORPD	Perform bitwise logical OR of packed double-precision floating-point values. パック ド倍精度浮動小数点値のビットごとの OR（論理和）演算を実行
XORPD	Perform bitwise logical XOR of packed double-precision floating-point values. パック ド倍精度浮動小数点値のビットごとの XOR（排他的論理和）演算を実行

5.6.1.4. SSE2 比較命令

SSE2 比較命令は、パック ド / スカラ倍精度浮動小数点値の比較を実行し、比較の結果をデスティネーション・オペランドまたは EFLAGS レジスタに返す。

CMPPD	Compare packed double-precision floating-point values. パック ド倍精度浮動小数点値を比較
CMPSD	Compare scalar double-precision floating-point values. スカラ倍精度浮動小数点値を比較
COMISD	Perform ordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register. スカラ倍精度浮動小数点値を順序付きで比較し、EFLAGS レジスタにフラグをセット
UCOMISD	Perform unordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register. スカラ倍精度浮動小数点値を順序付けなしで比較し、EFLAGS レジスタにフラグをセット

5.6.1.5. SSE2 シャッフル命令とアンパック命令

SSE2 シャッフル命令とアンパック命令は、パックド倍精度浮動小数点オペランド内の倍精度浮動小数点値のシャッフルまたはインターリーブを実行する。

SHUFPD	Shuffles values in packed double-precision floating-point operands. パックド倍精度浮動小数点オペランド内の値をシャッフル
UNPCKHPD	Unpacks and interleaves the high values from two packed double-precision floating-point operands. 2 つのパックド倍精度浮動小数点オペランドから上位の値をアンパックしてインターリーブ
UNPCKLPD	Unpacks and interleaves the low values from two packed double-precision floating-point operands. 2 つのパックド倍精度浮動小数点オペランドから下位の値をアンパックしてインターリーブ

5.6.1.6. SSE2 変換命令

SSE2 変換命令は、パックドまたは個々のダブルワード整数を、パックドまたはスカラ倍精度浮動小数点値に変換する。あるいは、その逆方向の変換を行う。また、変換命令は、パックドまたはスカラ形式の単精度浮動小数点値と倍精度浮動小数点値の間の変換も実行する。

CVTPD2PI	Convert packed double-precision floating-point values to packed doubleword integers. パックド倍精度浮動小数点値をパックド・ダブルワード整数に変換
CVTTPD2PI	Convert with truncation packed double-precision floating-point values to packed doubleword integers. 切り捨てを使用して、パックド倍精度浮動小数点値をパックド・ダブルワード整数に変換
CVTPI2PD	Convert packed doubleword integers to packed double-precision floating-point values. パックド・ダブルワード整数をパックド倍精度浮動小数点値に変換
CVTPD2DQ	Convert packed double-precision floating-point values to packed doubleword integers. パックド倍精度浮動小数点値をパックド・ダブルワード整数に変換
CVTTPD2DQ	Convert with truncation packed double-precision floating-point values to packed doubleword integers. 切り捨てを使用して、パックド倍精度浮動小数点値をパックド・ダブルワード整数に変換
CVTDQ2PD	Convert packed doubleword integers to packed double-precision floating-point values. パックド・ダブルワード整数をパックド倍精度浮動小数点値に変換

CVTPS2PD — Convert packed single-precision floating-point values to packed double-precision floating-point values.
パック ド単精度浮動小数点値をパック ド倍精度浮動小数点値に変換

CVTPD2PS — Convert packed double-precision floating-point values to packed single-precision floating-point values.
パック ド倍精度浮動小数点値をパック ド単精度浮動小数点値に変換

CVTSS2SD — Convert scalar single-precision floating-point values to scalar double-precision floating-point values.
スカラ単精度浮動小数点値をスカラ倍精度浮動小数点値に変換

CVTSD2SS — Convert scalar double-precision floating-point values to scalar single-precision floating-point values.
スカラ倍精度浮動小数点値をスカラ単精度浮動小数点値に変換

CVTSD2SI — Convert scalar double-precision floating-point values to a doubleword integer.
スカラ倍精度浮動小数点値をダブルワード整数に変換

CVTTSD2SI — Convert with truncation scalar double-precision floating-point values to scalar doubleword integers.
切り捨てを使用して、スカラ倍精度浮動小数点値をスカラ・ダブルワード整数に変換

CVTSI2SD — Convert doubleword integer to scalar double-precision floating-point value.
ダブルワード整数をスカラ倍精度浮動小数点値に変換

5.6.2. SSE2 パック ド単精度浮動小数点命令

SSE2パック ド単精度浮動小数点命令は、単精度浮動小数点オペランドと整数オペランドの変換操作を実行する。これらの命令は、SSE の単精度浮動小数点命令を拡張したものである。

CVTDQ2PS — Convert packed doubleword integers to packed single-precision floating-point values.
パック ド・ダブルワード整数をパック ド単精度浮動小数点値に変換

CVTPS2DQ — Convert packed single-precision floating-point values to packed signed doubleword integers.
パック ド単精度浮動小数点値をパック ド符号付きダブルワード整数に変換

CVTTPS2DQ — Convert with truncation packed single-precision floating-point values to packed doubleword integers.
切り捨てを使用して、パック ド単精度浮動小数点値をパック ド・ダブルワード整数に変換

5.6.3. SSE2 128 ビット SIMD 整数命令

SSE2 128 ビット SIMD 整数命令は、XMM レジスタ内のパックドワード、パックド・ダブルワード、パックド・クワッドワードに対する追加の演算を実行する。

MOVDQA	Move aligned double quadword. アライメントの合ったダブル・クワッドワードを転送
MOVDQU	Move unaligned double quadword. アライメントの合っていないダブル・クワッドワードを転送
MOVQ2DQ	Move quadword integer from MMX to XMM registers. クワッドワード整数を MMX レジスタから XMM レジスタに転送
MOVDQ2Q	Move quadword integer from XMM to MMX registers. クワッドワード整数を XMM レジスタから MMX レジスタに転送
PMULUDQ	Multiply packed unsigned doubleword integers. パックド符号なしダブルワード整数を乗算
PADDQ	Add packed quadword integers. パックド・クワッドワード整数を加算
PSUBQ	Subtract packed quadword integers. パックド・クワッドワード整数を減算
PSHUFLW	Shuffle packed low words. パックド下位ワードをシャッフル
PSHUFHW	Shuffle packed high words. パックド上位ワードをシャッフル
PSHUFD	Shuffle packed doublewords. パックド・ダブルワードをシャッフル
PSLLDQ	Shift double quadword left logical. ダブル・クワッドワードを論理左シフト
PSRLDQ	Shift double quadword right logical. ダブル・クワッドワードを論理右シフト
PUNPCKHQDQ	Unpack high quadwords. 上位クワッドワードをアンパック
PUNPCKLQDQ	Unpack low quadwords. 下位クワッドワードをアンパック

5.6.4. SSE2 キャッシュ制御命令と命令順序付け命令

SSE2 キャッシュ制御命令は、XMM レジスタからメモリにデータをストアするときの非テンポラル・データのキャッシュ処理の制御を強化する。LFENCE 命令と MFENCE 命令は、ストア操作時の命令の順序の制御を強化する。

CLFLUSH	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy. プロセッサのキャッシュ階層内の全レベルから、メモリ・オペランドおよび関連するキャッシュ・ラインをフラッシュし無効化
LFENCE	Serializes load operations. ロード操作をシリアル化
MFENCE	Serializes load and store operations. ロード操作およびストア操作をシリアル化
PAUSE	Improves the performance of "spin-wait loops." 「時間待ち（spin-wait）ループ」のパフォーマンスを改善
MASKMOVDQU	Non-temporal store of selected bytes from an XMM register into memory. XMM レジスタからメモリへの、選択したバイトの非テンポラルなストア
MOVNTPD	Non-temporal store of two packed double-precision floating-point values from an XMM register into memory. XMM レジスタからメモリへの、2 つのパック ド倍精度浮動小数点値の非テンポラルなストア
MOVNTDQ	Non-temporal store of double quadword from an XMM register into memory. XMM レジスタからメモリへの、ダブル・クワッドワードの非テンポラルなストア
MOVNTI	Non-temporal store of a doubleword from a general-purpose register into memory. 汎用レジスタからメモリへの、ダブルワードの非テンポラルなストア

5.7. SSE3

SSE3 では、SSE テクノロジ、SSE2 テクノロジ、および x87-FP 演算機能の性能を高める 13 個の命令が追加されている。SSE3 は、以下のカテゴリに分けられる。

- 整数変換で使用される x87FPU 命令 ×1
- アライメントの合っていないデータロードに対処する SIMD 整数命令 ×1
- SIMD 浮動小数点パック ド加算 / 減算命令 ×2
- SIMD 浮動小数点水平加算 / 減算命令 ×4
- SIMD 浮動小数点ロード / 転送 / 複製命令 ×3
- スレッド同期化命令 ×2

SSE3 は、SSE3 をサポートする IA-32 プロセッサ上でのみ実行できる。プロセッサが SSE3 をサポートしているかどうかは、CPUID 命令によって検出できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の CPUID 命令の説明を参照）。

以下の各項では、各サブグループについて説明する。

5.7.1. SSE3 x87-FP 整数変換命令

FISTTP — Behaves like the FISTP instruction but uses truncation, irrespective of the rounding mode specified in the floating-point control word (FCW).
FISTP 命令と同様の動作をするが、浮動小数点制御ワード（FCW）で指定された丸めモードにかかわらず切り捨てを使用

5.7.2. アライメントの合っていない SSE3 専用 128 ビット・データ・ロード命令

LDDQU — Special 128-bit unaligned load designed to avoid cache line splits.
キャッシュ・ラインの分割を防止するように設計された、アライメントの合っていない専用 128 ビット・ロード

5.7.3. SSE3 SIMD 浮動小数点パックド加算 / 減算命令

ADDSUBPS — Performs single-precision addition on the second and fourth pairs of 32-bit data elements within the operands; and single-precision subtraction on the first and third pairs.
オペランド内の 32 ビット・データ要素の 2 番目と 4 番目のペアに対して単精度の加算を実行、1 番目と 3 番目のペアに対して単精度の減算を実行

ADDSUBPD — Performs double-precision addition on the second pair of quadwords, and double-precision subtraction on the first pair.
クワッドワードの 2 番目のペアに対して倍精度の加算を実行、1 番目のペアに対して倍精度の減算を実行

5.7.4. SSE3 SIMD 浮動小数点水平加算 / 減算命令

HADDPS — Performs a single-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.
隣接したデータ要素に対して単精度の加算を実行。結果中の最初のデータ要素は、第 1 オペランド中の 1 番目と 2 番目の要素を足して得られたものである。同様に、2 番目のデータ要素は第 1 オペランド中の 3 番目と 4 番目の要素を、3 番目のデータ要素は第 2 オペランド中の 1 番目と 2 番目の要素を、4 番目のデータ要素は第 2 オペランド中の 3 番目と 4 番目の要素をそれぞれ足して得られたものである

HSUBPS — Performs a single-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.
隣接したデータ要素に対して単精度の減算を実行。結果中の最初のデータ要素は、第 1 オペランド中の 1 番目の要素から第 1 オペランド中の 2 番目の要素を引いて得られたものである。同様に、2 番目のデータ要素は第 1 オペランド中の 3 番目の要素から第 1 オペランド中の 4 番目の要素を、3 番目のデータ要素は第 2 オペランド中の 1 番目の要素から第 2 オペランド中の 2 番目の要素を、4 番目のデータ要素は第 2 オペランド中の 3 番目の要素から第 2 オペランド中の 4 番目の要素をそれぞれ引いて得られたものである

HADDPD	Performs a double-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand. 隣接したデータ要素に対して倍精度の加算を実行。結果中の最初のデータ要素は、第 1 オペランド中の 1 番目と 2 番目の要素を足して得られたものである。同様に、2 番目のデータ要素は第 2 オペランド中の 1 番目と 2 番目の要素を足して得られたものである
HSUBPD	Performs a double-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand. 隣接したデータ要素に対して倍精度の減算を実行。結果中の最初のデータ要素は、第 1 オペランド中の 1 番目の要素から第 1 オペランド中の 2 番目の要素を引いて得られたものである。同様に、2 番目のデータ要素は第 2 オペランド中の 1 番目の要素から第 2 オペランド中の 2 番目の要素を引いて得られたものである

5.7.5. SSE3 SIMD 浮動小数点ロード / 転送 / 複製命令

MOVSHDUP	Loads/moves 128-bits, duplicating the second and fourth 32-bit data elements. 128 ビットをロード / 転送し、2 番目と 4 番目の 32 ビット・データ要素を複製
MOVSLDUP	Loads/moves 128-bits, duplicating the first and third 32-bit data elements. 128 ビットをロード / 転送し、1 番目と 3 番目の 32 ビット・データ要素を複製
MOVDDUP	Loads/moves 64-bits (bits[63-0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register. This duplicates the 64 bits from the source. 64 ビット（ソースがレジスタの場合はビット [63 ～ 0]）をロード / 転送し、128 ビットの結果レジスタの下位半分と上位半分で同じ 64 ビットを返す。これにより、ソース中の 64 ビットが複製される

5.7.6. SSE3 エージェント同期化命令

MONITOR	Sets up an address range used to monitor write-back stores. ライトバック・ストアの監視に使用されるアドレス範囲をセットアップ
MWAIT	Enables a logical processor to enter into an optimized state while waiting for a write-back store to the address range set up by the MONITOR instruction. MONITOR 命令でセットアップされたアドレス範囲へのライトバック・ストアを待機する間に、論理プロセッサを最適化された状態にすることができる

5.8. システム命令

次に挙げるシステム命令はプロセッサの機能を制御するために使用するもので、オペレーティング・システムやエグゼクティブのサポート用に用意されている。

LGDT	Load global descriptor table（GDT）register グローバル・ディスクリプタ・テーブル（GDT）レジスタをロード
SGDT	Store global descriptor table（GDT）register グローバル・ディスクリプタ・テーブル（GDT）レジスタをストア
LLDT	Load local descriptor table（LDT）register ローカル・ディスクリプタ・テーブル（LDT）レジスタをロード
SLDT	Store local descriptor table（LDT）register ローカル・ディスクリプタ・テーブル（LDT）レジスタをストア
LTR	Load task register タスクレジスタをロード
STR	Store task register タスクレジスタをストア
LIDT	Load interrupt descriptor table（IDT）register 割り込みディスクリプタ・テーブル（IDT）レジスタをロード
SIDT	Store interrupt descriptor table（IDT）register 割り込みディスクリプタ・テーブル（IDT）レジスタをストア
MOV	Load and store control registers コントロール・レジスタをロードおよびストア
LMSW	Load machine status word マシン・ステータス・ワードをロード
SMSW	Store machine status word マシン・ステータス・ワードをストア
CLTS	Clear the task-switched flag タスク・スイッチング・フラグをクリア
ARPL	Adjust requested privilege level 要求された特権レベルを調整
LAR	Load access rights アクセス権をロード
LSL	Load segment limit セグメント・リミットをロード
VERR	Verify segment for reading セグメントが読み取り可能であるか確認
VERW	Verify segment for writing セグメントが書き込み可能であるか確認
MOV	Load and store debug registers デバッグレジスタをロードおよびストア
INVD	Invalidate cache, no writeback ライトバックせずにキャッシュを無効化

WBINVD	Invalidate cache, with writeback ライトバックしてキャッシュを無効化
INVLPG	Invalidate TLB Entry TLB エントリを無効化
LOCK（prefix）	Lock Bus バスをロック
HLT	Halt processor プロセッサを停止
RSM	Return from system management mode (SSM) システム管理モード（SMM）から復帰
RDMSR	Read model-specific register モデル固有レジスタの読み取り
WRMSR	Write model-specific register モデル固有レジスタの書き込み
RDPMC	Read performance monitoring counters パフォーマンス監視カウンタの読み取り
RDTSC	Read time stamp counter タイム・スタンプ・カウンタの読み取り
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL=0. 高速システムコール、CPL=0 でフラットな保護モードカーネルに転送
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL=3. 高速システムコール、CPL=3 でフラットな保護モードカーネルに転送

6

プロシージャ・コール、割り込み、例外

第 6 章
プロシージャ・コール、割り込み、例外

6

第 6 章では、プロシージャやサブルーチンのコールを実行するために IA-32 アーキテクチャに用意されている機能について説明する。また、割り込みや例外が、アプリケーション・プログラマの視点から見てどのように処理されるかについても説明する。

6.1. プロシージャ・コールのタイプ

プロセッサは、プロシージャ・コールを次の 2 つの方法でサポートする。

- CALL 命令および RET 命令。
- ENTER 命令および LEAVE 命令。CALL 命令および RET 命令と併用。

これらのプロシージャ・コールの機構はいずれも、プロシージャ・スタック（通常は単にスタックと呼ぶ）を使用して、コール元のプロシージャのステートをセーブし、パラメータをコールされたプロシージャに渡し、現在実行されているプロシージャのローカル変数を格納する。

割り込みや例外を処理するためのプロセッサの機能は、CALL 命令や RET 命令が使用する機能と同じである。

6.2. スタック

スタック（図 6-1. を参照）は、連続するメモリ・ロケーションの配列である。このスタックはセグメント内に格納され、SS レジスタ内のセグメント・セレクタによって識別される。フラット・メモリ・モデルを使用する場合は、スタックはプログラム用のリニアアドレス空間の任意の場所に配置できる。1 つのスタックは、セグメントの最大サイズである 4G バイトまでのサイズを持つことができる。

スタック上にアイテムを配置する場合は PUSH 命令を、スタックから取り出す場合は POP 命令を使用する。あるアイテムをスタックにプッシュする場合は、プロセッサはまず ESP レジスタをデクリメントし、次にそのアイテムを新たにスタックのトップに書き込む。アイテムをスタックからポップする場合は、プロセッサはスタックのトップからアイテムを読み取り、次に ESP レジスタをインクリメントする。このように、アイテムをスタックにプッシュした場合はスタックはメモリの**下位**アドレスに向

かって拡大し、アイテムをスタックからポップした場合はスタックは**上位**アドレスに向かって縮小されることになる。

プログラムやオペレーティング・システムあるいはエグゼクティブにおいては、多数のスタックをセットアップできる。例えば、マルチタスク・システムでは、それぞれのタスクが別々にスタックを持つことができる。システム内に設定できるスタックの数は、セグメントの最大数と、使用可能な物理メモリとによって制限される。

ただし、システムが多数のスタックをセットアップした場合でも、特定の時点で使用できるスタックは1つ（すなわち現在のスタック）だけである。**「現在のスタック」**とは、SS レジスタが参照するセグメント内に格納されているスタックである。

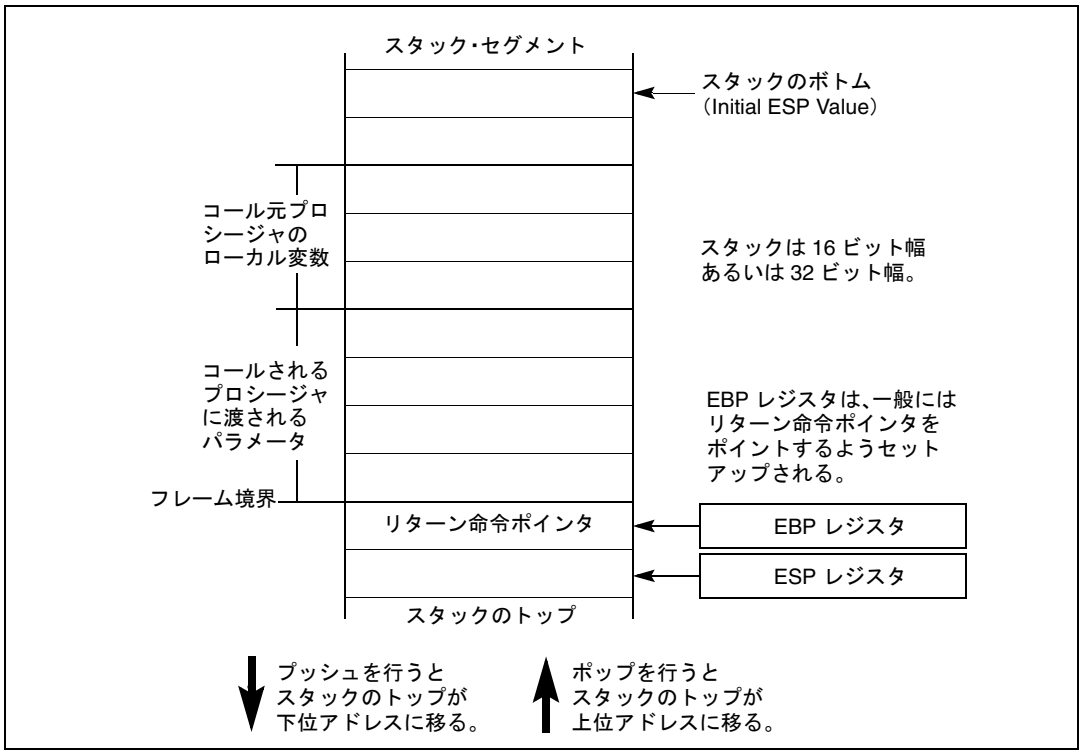


図 6-1. スタックの構造

プロセッサは、あらゆるスタック操作に対して自動的に SS レジスタを参照する。例えば、ESP レジスタがメモリアドレスとして使用されている場合は、SS レジスタは現在のスタック内のアドレスを自動的にポイントする。また、CALL、RET、PUSH、POP、ENTER、LEAVE の各命令もすべて、現在のスタックに対して操作を実行する。

6.2.1. スタックのセットアップ

スタックをセットし、それを現在のスタックとして設定するには、プログラムやオペレーティング・システムあるいはエグゼクティブは次の操作を実行しなければならない。

1. スタック・セグメントを設定する。
2. MOV、POP、LSS のいずれかの命令を使用して、スタック・セグメントのセグメント・セレクタを SS レジスタにロードする。
3. MOV、POP、LSS のいずれかの命令を使用して、スタックのスタックポインタを ESP レジスタにロードする。LSS 命令を使用すれば、1 つの操作で SS レジスタと ESP レジスタをロードできる。

セグメント・ディスクリプタをセットアップする方法や、スタック・セグメントに対するセグメントの制限については、『IA-32 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、下巻』の第 3 章「セグメント・ディスクリプタ」を参照のこと。

6.2.2. スタックのアライメント

スタック・セグメントのスタックポインタは、スタック・セグメントの幅によって 16 ビット（ワード）境界か 32 ビット（ダブルワード）境界のいずれかにアライメントを揃えなければならない。スタック・セグメントの幅は、現行コード・セグメントのセグメント・ディスクリプタ内にある D フラグによってセットされる（『IA-32 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、下巻』の第 3 章にある「セグメント・ディスクリプタ」を参照）。PUSH 命令と POP 命令では、プッシュ操作やポップ操作に対してスタックポインタをデクリメントしたりインクリメントする量は、D フラグによって決まる。スタック幅が 16 ビットの場合は、スタックポインタは 16 ビットずつインクリメントまたはデクリメントされる。スタック幅が 32 ビットの場合は、スタックポインタは 32 ビットずつインクリメントまたはデクリメントされる。16 ビット値を 32 ビット幅のスタックにプッシュすると、スタックのアライメントのずれが発生する（つまり、スタックポインタのアライメントがダブルワード境界に合わなくなる）。ただし、セグメント・レジスタ（16 ビット・セグメント・セレクタ）の内容を 32 ビット幅のスタックにプッシュした場合は例外である。この場合は、プロセッサが自動的に、スタックポインタのアライメントを次の 32 ビット境界に合わせる。

プロセッサは、スタックポインタのアライメントはチェックしない。スタックポインタのアライメントを適切に維持するためには、プロセッサ上で動作しているプログラム、タスク、およびシステム・プロシージャによって行う。スタックポインタのアラ

イメントが正しく行われていないと、処理能力が極端に低下するばかりでなく、場合によってはプログラム障害が発生する。

6.2.3. スタックアクセスにおけるアドレスサイズ属性

PUSH 命令や POP 命令などの暗黙的にスタックを使用する命令では、それぞれが 16 ビットまたは32ビットの2つのアドレスサイズ属性を持つ。その理由は、これらの命令が暗黙的にスタックのトップのアドレスを必ず持っており、場合によっては明示的なメモリアドレス（例えば、PUSH Array1[EBX]）も持つこともあるためである。明示的なアドレスの属性は、現行コード・セグメントのDフラグと、67Hのアドレス・サイズ・プリフィックスによって決まる。

スタックアクセスにおいて SP か ESP のどちらが使用されるかは、スタックのトップのアドレスサイズ属性で決まる。16 ビットのアドレスサイズ属性を持つスタック操作では、16 ビットの SP スタック・ポインタ・レジスタを使用して、最大スタックアドレスとして FFFFH まで使用できる。一方、32 ビットのアドレスサイズ属性を持つスタック操作では、32 ビットの ESP レジスタを使用して、最大スタックアドレスとして FFFFFFFFH まで使用できる。スタックとして使用されるデータ・セグメントについてのデフォルトのアドレスサイズ属性は、セグメントのディスクリプタの D フラグによって制御される。このフラグがクリアされている場合は、デフォルトのアドレスサイズ属性は16ビットになる。このフラグがセットされている場合は、アドレスサイズ属性は32ビットになる。

6.2.4. プロシージャのリンクに関する情報

プロセッサには、プロシージャ間をリンクさせるために、スタック・フレーム・ベース・ポインタとリターン命令ポインタの2つのポインタがある。これらのポインタを、ソフトウェア上で標準的なプロシージャ・コール技法と共に使用すれば、プロシージャ間のリンクを確実にしかもコヒーレンシを損なわずに実行できる。

6.2.4.1. スタック・フレーム・ベース・ポインタ

スタックは、一般に、一連のフレームに分割される。それぞれのスタックフレームには、ローカル変数、別のプロシージャに渡されるパラメータに加え、プロシージャのリンクに関する情報が格納される。EBP レジスタに格納されているスタック・フレーム・ベース・ポインタは、コールされるプロシージャについてのスタックフレーム内の固定参照点を示す。スタック・フレーム・ベース・ポインタを使用するために、通常は、コールされたプロシージャは、スタックにローカル変数をプッシュする前に、ESP レジスタの内容を EBP レジスタにコピーする。この後、スタック・フレーム・ベース・ポインタを使って、スタック上に渡されたデータ構造、リターン命令ポインタ、

コールされたプロシージャによってスタックに追加されたローカル変数に容易にアクセスできる。

ESP レジスタと同じように、EBP レジスタも現在のスタック・セグメント（すなわち、SS レジスタのその時点での内容によって指定されるセグメント）内のアドレスを自動的にポイントする。

6.2.4.2. リターン命令ポインタ

コールされたプロシージャの最初の命令に分岐する前に、CALL 命令によって EIP レジスタ内のアドレスが現在のスタックにプッシュされる。これ以後、このアドレスはリターン命令ポインタと呼ばれ、コールされたプロシージャから戻った後にコール元のプロシージャが実行を再開する命令をポイントする。コールされたプロシージャから戻ると、RET 命令によってリターン命令ポインタがスタックからポップされ、EIP レジスタに戻される。そこから、コール元プロシージャの実行が再開される。

プロセッサは、リターン命令ポインタの位置をトラッキングしていない。したがって、RET 命令を発行する前に、プログラマはスタックポインタが確実にスタック上のリターン命令ポインタをポイントするようにしなければならない。リターン命令ポインタをポイントするようスタックポインタをリセットするためには、通常、EBP レジスタの内容を ESP レジスタに移さなければならない。プロシージャ・コールの直後に EBP レジスタにスタックポインタがロードされていれば、EBP レジスタはスタック上のリターン命令ポインタをポイントしているはずである。

プロセッサにとっては、リターン命令ポインタがコール元プロシージャをポイントする必要はない。したがって、RET 命令を実行する前に、ソフトウェア上でリターン命令ポインタを操作することで、現行コード・セグメント内の任意のアドレスをポイントするか（near リターン）、別のコード・セグメント内の任意のアドレスをポイントする（far リターン）ことができる。このような操作では、明確に定義されたコード・エントリ・ポイントのみを使用し、注意して実行しなければならない。

6.3. CALL と RET によるプロシージャのコール

CALL 命令を使用すれば、現行コード・セグメント内のプロシージャに制御を転送する（**near コール**）か、異なるコード・セグメント内のプロシージャに制御を転送する（**far コール**）ことができる。一般に、near コールは現在実行されているプログラムやタスク内のローカル・プロシージャにアクセスする際に使用し、far コールはオペレーティング・システムのプロシージャや、異なるタスク内のプロシージャにアクセスする際に使用する。CALL 命令の詳しい説明については、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「CALL_Call Procedure」を参照のこと。

CALL 命令の near コールと far コールに対応させるため、RET 命令にも near リターンと far リターンが用意されている。また、RET 命令を使用すれば、リターン時にプログラム上でスタックポインタをインクリメントし、スタックからパラメータを開放することができる。スタックから開放されるバイト数は、RET 命令のオプションの引き数（*n*）で決まる。RET 命令の詳しい説明については、『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」の「RET_Return from Procedure」を参照のこと。

6.3.1. near コール操作と near リターン操作

near コールを実行するときには、プロセッサは以下の動作を行う（図 6-4. を参照）。

1. EIP レジスタの現在値を、スタックにプッシュする。
2. コールされたプロシージャのオフセットを、EIP レジスタにロードする。
3. コールされたプロシージャの実行を開始する。

near リターンを実行するときには、プロセッサは次の動作を行う。

1. スタックのトップ値（リターン命令ポインタ）を EIP レジスタにポップする。
2. RET 命令にオプション引き数の *n* がある場合は、パラメータをスタックから開放するため、*n* オペランドで指定されたバイト数だけスタックポインタをインクリメントする。
3. コール元プロシージャの実行を再開する。

6.3.2. far コール操作と far リターン操作

far コールを実行するときには、プロセッサは以下の動作を行う（図 6-4. を参照）。

1. CS レジスタの現在値を、スタックにプッシュする。
2. EIP レジスタの現在値を、スタックにプッシュする。
3. コールされたプロシージャを格納しているセグメントのセグメント・セレクタを、CS レジスタにロードする。
4. コールされたプロシージャのオフセットを、EIP レジスタにロードする。
5. コールされたプロシージャの実行を開始する。

far リターンを実行するときには、プロセッサは次の動作を行う。

1. スタックのトップ値（リターン命令ポインタ）を EIP レジスタにポップする。

2. スタックのトップ値（戻り先となるコード・セグメントのセグメント・セレクタ）を、CS レジスタにポップする。
3. RET 命令にオプション引き数の *n* がある場合は、パラメータをスタックから開放するため、*n* オペランドで指定されたバイト数だけスタックポインタをインクリメントする。
4. コール元プロシージャの実行を再開する。

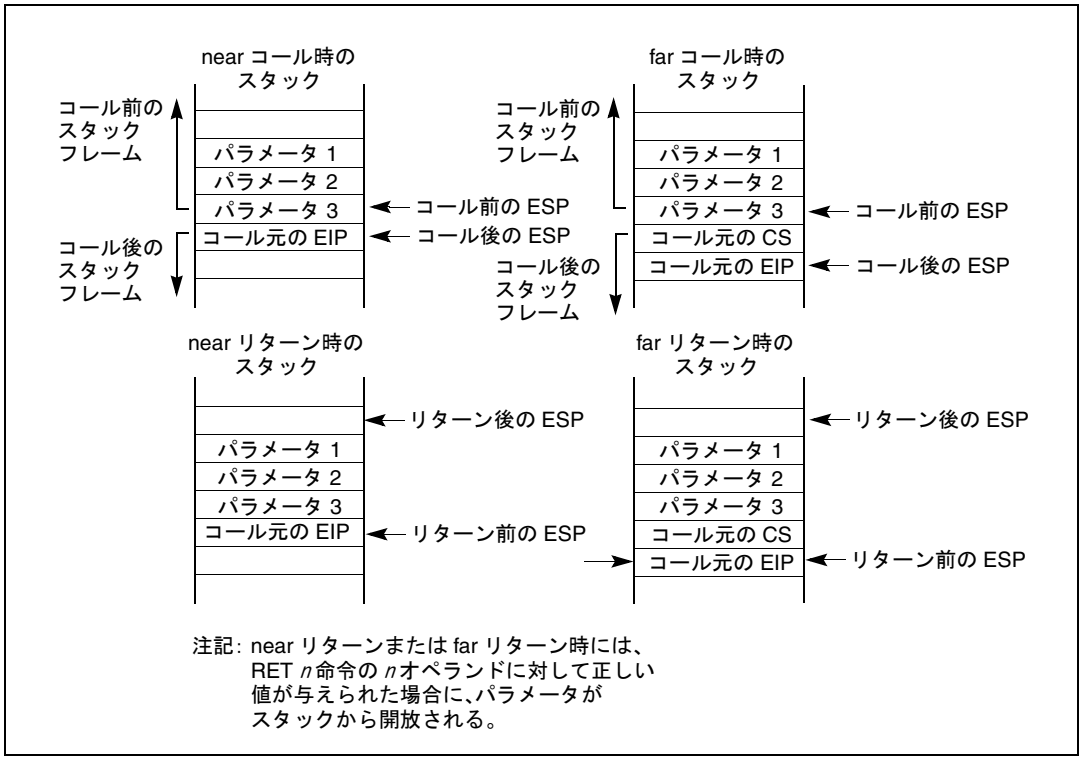


図 6-2. near コールと far コールでのスタック

6.3.3. パラメータの受け渡し

パラメータをプロシージャ間で受け渡すには、汎用レジスタを介する方法、引き数リストを使用する方法、スタックを利用する方法、の3種類の方法がある。

6.3.3.1. 汎用レジスタによるパラメータの受け渡し

プロセッサは、プロシージャ・コールに際して汎用レジスタのステートをセーブしない。したがって、コール元プロシージャは、CALL 命令を実行する前に、パラメータを（ESP レジスタと EBP レジスタを除く）任意の汎用レジスタにコピーすると、コールされるプロシージャに最大6つのパラメータを渡すことができる。コールされたプ

ロシージャも同様に、汎用レジスタを介してコール元プロシージャにパラメータを返すことができる。

6.3.3.2. スタックによるパラメータの受け渡し

多数のパラメータをコールされるプロシージャに渡す場合は、コール元プロシージャのスタックフレーム内のスタック上にパラメータを配置できる。このとき、(EBP レジスタ内にある) スタックフレームのベースポインタを使用してフレーム境界を設定すれば、パラメータへのアクセスが容易になる。

また、コールされたプロシージャからコール元プロシージャにパラメータを返す際にも、スタックを使用できる。

6.3.3.3. 引き数リストによるパラメータの受け渡し

多数のパラメータ（またはデータ構造）をコールされるプロシージャに渡すもう 1 つの方法として、メモリ上のいずれかのデータ・セグメントにある引き数リストにパラメータも配置できる。この後、汎用レジスタまたはスタックを介して、引き数リストに対するポインタをコールされたプロシージャに渡すことができる。また、同じ方法で、コール元プロシージャにパラメータを返すことができる。

6.3.4. プロシージャのステート情報のセーブ

プロセッサは、プロシージャ・コールに際して汎用レジスタ、セグメント・レジスタ、EFLAGS レジスタのいずれの内容もセーブしない。したがって、コール元プロシージャは、リターン後に実行を再開するにあたって必要な汎用レジスタの値を明示的にセーブしなければならない。これらの値は、スタック上、あるいはメモリ上のいずれかのデータ・セグメントにセーブできる。

PUSHA 命令や POPA 命令を使用すれば、汎用レジスタの内容を容易にセーブしリストアすることができる。PUSHA 命令は、すべての汎用レジスタ内の値をスタックにプッシュする。プッシュする順序は、EAX、ECX、EDX、EBX、ESP（PUSHA 命令を実行する前の値)、EBP、ESI、EDI である。これに対し、POPA 命令は、PUSHA 命令でセーブしたすべてのレジスタ値（ESI 値を除く）を、スタックからそれぞれの対応するレジスタにポップする。

コールされたプロシージャにおいて、いずれかのセグメント・レジスタのステートが明示的に変更された場合は、コール元プロシージャへのリターンを実行する前に、それらの値を元の値にリストアしなければならない。

コール元プロシージャが EFLAGS レジスタのステートを保持しておく必要がある場合には、PUSHF/PUSHFD 命令と POPF/POPFD 命令を使用することで、レジスタの全部、

または一部をセーブし、リストアすることができる。PUSHF 命令は、EFLAGS レジスタの下位ワードをスタックにプッシュし、PUSHFD 命令は、レジスタ全体をスタックにプッシュする。POPF 命令は、スタックから EFLAGS レジスタの下位ワードに 1 ワードをポップする。POPFD 命令は、スタックからレジスタに 1 ダブルワードをポップする。

6.3.5. 他の特権レベルに対するコール

IA-32 アーキテクチャの保護メカニズムにおいては、4 つの特権レベルを認識する。特権レベルはそれぞれ 0 ～ 3 の番号が付けられ、数が大きくなるほど特権レベルは低くなる。特権レベルを使用する理由は、オペレーティング・システムの信頼性を高めることにある。例えば、図 6-3. に、保護のリングとして見立てた場合、それぞれの特権レベルがどのように解釈できるかを示す。

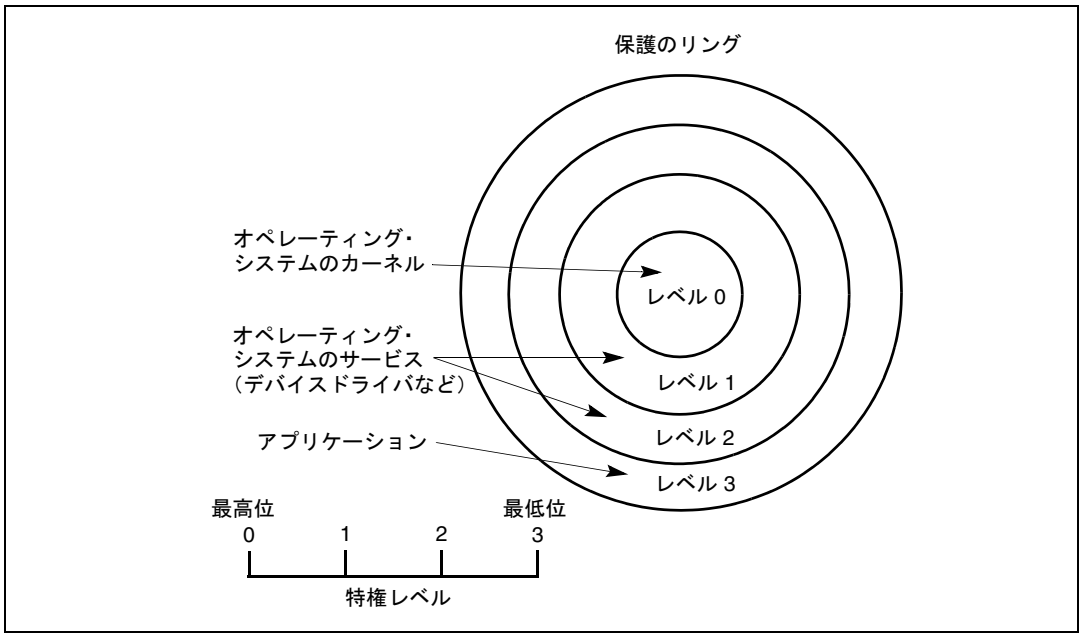


図 6-3. 保護のリング

この例では、最高の特権レベル 0（図の中央）が、システム内の最も重要なコード・モジュール（通常は、オペレーティング・システムのカーネル）を格納しているセグメントに対して使用されている。外側のリング（外にいくほど特権は小さくなる）に行くほど、重要度が低いソフトウェアのコード・モジュールを格納しているセグメントになる。

低い特権のセグメント内にあるコード・モジュールから高い特権のセグメントで動作するモジュールにアクセスするには、**ゲート**と呼ばれる厳密に制御され保護されてい

るインターフェイスを使用しなければならない。保護ゲートを介さず、しかも十分なアクセス権を持たないで高い特権のセグメントにアクセスしようとすると、一般保護例外（#GP）が発生する。

オペレーティング・システムやエグゼクティブがこのマルチレベルの保護機構を使用する場合は、コール元プロシージャより高い特権保護レベルにあるプロシージャへのコールは、far コールと同様の方法で処理される（6.3.2. 項「far コール操作と far リターン操作」を参照）。ただし、次の点で異なる。

- CALL 命令で与えられるセグメント・セレクタは、**コール・ゲート・ディスクリプタ**と呼ばれる特殊なデータ構造を参照する。コール・ゲート・ディスクリプタは、次の内容を保持している。
  - アクセス権に関する情報
  - コールされるプロシージャのコード・セグメントのセグメント・セレクタ
  - コード・セグメントに対するオフセット（すなわち、コールされるプロシージャの命令ポインタ）
- プロセッサは、コールされたプロシージャを実行するために、新しいスタックに切り替える（スタックスイッチ）。それぞれの特権レベルは、自身のスタックを持つ。特権レベル 3 のスタックのセグメント・セレクタとスタックポインタは、それぞれ SS レジスタと ESP レジスタに格納され、さらに、より高い特権レベルに対するコールが発生した時点で自動的にセーブされる。特権レベル 2、1、0 の各スタックのセグメント・セレクタとスタックポインタは、タスク・ステート・セグメント（TSS）と呼ばれるシステム・セグメント内に格納される。

スタックスイッチ実行時にコールゲートと TSS を使用することは、一般保護例外が発生した場合を除き、コール元プロシージャにとっては透過である。

6.3.6. 特権レベル間のコール操作とリターン操作

より高い特権保護レベルに対してコールを実行するときには、プロセッサは次の動作を行う（図 6-4. を参照）。

1. アクセス権のチェック（特権チェック）を実行する。
2. SS、ESP、CS、EIP の各レジスタの現在値を一時的に内部にセーブする。

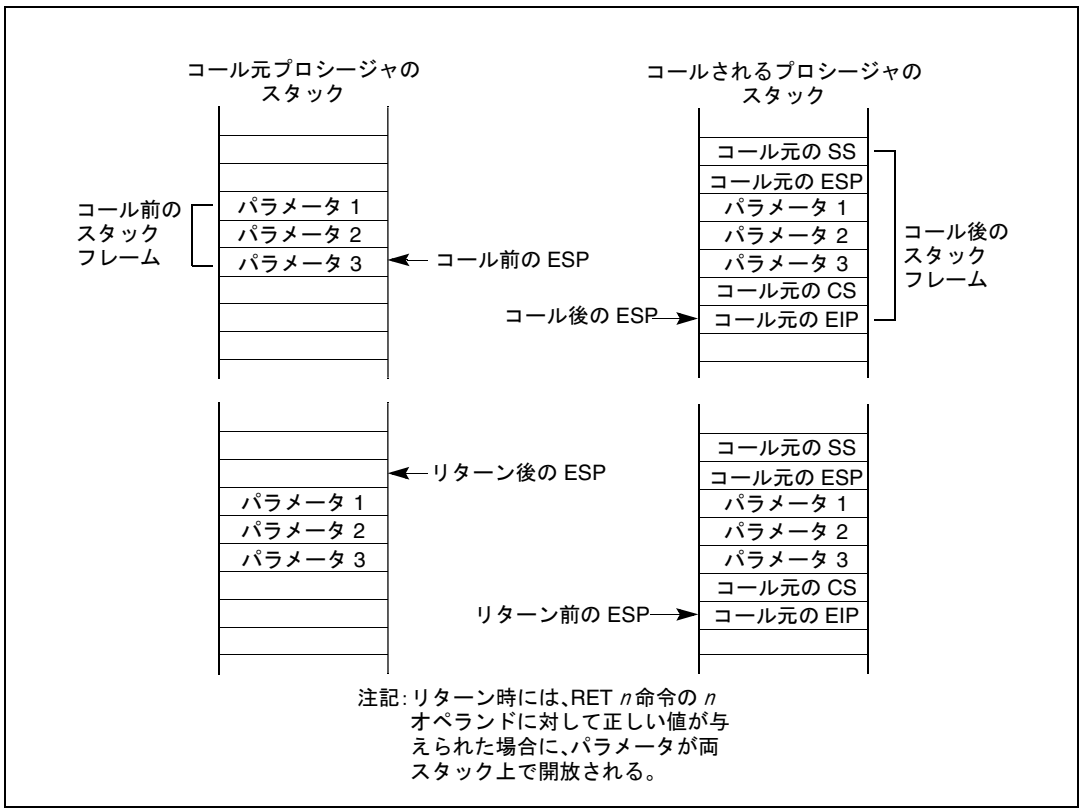


図 6-4. 異なる特権レベルへのコール時のスタックスイッチ

3. TSS レジスタに格納されている新しいスタック（すなわち、現在コールされている特権レベル用のスタック）のセグメント・セレクタとスタックポインタを SS レジスタと ESP レジスタにロードし、新しいスタックに切り替える。
4. コール元プロシージャのスタックに対して一時的にセーブしておいた SS 値と ESP 値を、この新しいスタックにプッシュする。
5. コール元プロシージャのスタックから新しいスタックにパラメータをコピーする。新しいスタックにコピーされるパラメータの数は、コール・ゲート・ディスクリプタ内の値で決まる。
6. コール元プロシージャに対して一時的にセーブしておいた CS 値と EIP 値を、新しいスタックにプッシュする。
7. 新しいコード・セグメントのセグメント・セレクタと新しい命令ポインタを、コール・ゲートから CS レジスタと EIP レジスタにそれぞれロードする。
8. コールされたプロシージャの実行を、新しい特権レベルで開始する。

特権プロシージャからリターンを実行するときには、プロセッサは次の動作を行う。

1. 特権チェックを実行する。
2. CS レジスタと EIP レジスタに、コール前の値をリストアする。
3. RET 命令にオプション引き数の *n* がある場合は、パラメータをスタックから開放するため、*n* オペランドで指定されたバイト数だけスタックポインタをインクリメントする。コール・ゲート・ディスクリプタが、スタック間で 1 つ以上のパラメータをコピーするよう指定している場合は、RET *n* 命令を使用して両スタックからパラメータを開放しなければならない。*n* オペランドには、各スタック上でパラメータが占有するバイト数を指定する。リターン時に、プロセッサは各スタックに対して *n* だけ ESP をインクリメントし、これらのパラメータをスタックから効率よく削除する。
4. SS レジスタと ESP レジスタに、コール前の値をリストアする。これで、コール元プロシージャのスタックへ切り替えられる。
5. RET 命令にオプション引き数の *n* がある場合は、パラメータをスタックから開放するため、*n* オペランドで指定されたバイト数だけスタックポインタをインクリメントする（ステップ 3 の説明を参照）。
6. コール元プロシージャの実行を再開する。

特権レベルに対するコールや、コール・ゲート・ディスクリプタに関する詳しい説明については、『IA-32 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、下巻』の第 4 章「保護」を参照のこと。

6.4. 割り込みと例外

プロセッサには、割り込みと例外という、プログラム実行を中断するためのメカニズムが 2 つある。

- **割り込み**は、一般的に I/O デバイスでトリガされる非同期イベントである。
- **例外**は、プロセッサが、命令実行時に、あらかじめ定義されている条件を検出した場合に生成される同期イベントである。IA では、フォルト、トラップ、アボートという 3 クラスの例外を指定している。

プロセッサは、割り込みや例外に対して基本的には同じ方法で応答する。すなわち、割り込みあるいは例外が通知されると、プロセッサは現在実行されているプログラムまたはタスクを停止し、割り込みあるいは例外の処理専用に作成されたハンドラ・プロシージャに切り替える。プロセッサは、割り込みディスクリプタ・テーブル（IDT）内のエントリを介してハンドラ・プロシージャにアクセスする。このハンドラが割り込みあるいは例外の処理を完了すると、割り込みをかけられたプログラムまたはタスクにプログラムの制御が戻される。

オペレーティング・システム、エグゼクティブ、デバイスドライバなどは、通常は、割り込みや例外をアプリケーション・プログラムやタスクからは独立して処理する。

ただし、アプリケーション・プログラムでは、アセンブリ言語のコールを介して、オペレーティング・システムやエグゼクティブに組み込まれている割り込みハンドラや例外ハンドラにアクセスできる。本節の以降では、プロセッサが持つ割り込みならびに例外の処理機構について簡単に説明する。この機構の詳しい説明については、『IA-32 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、下巻』の第5章「割り込みと例外の処理」を参照のこと。

IA-32アーキテクチャでは、17のプロセッサ定義の割り込みと例外、224のユーザ定義の割り込みがあらかじめ定義されている。これらの割り込みや例外は、IDT内のエントリに関連付けられている。IDT内の割り込みや例外はそれぞれ、**ベクタ**と呼ばれる番号で識別される。表6-1.に、割り込みと例外を、IDT内のエントリならびにそれぞれに対応するベクタ番号と共に併記する。ベクタ0～8、10～14、16～19は、定義済みの割り込みと例外である。ベクタ32～255は、ユーザ定義の割り込みであり、これらは**マスク可能割り込み**と呼ばれる。

プロセッサには、IDT内のエントリをポイントしない割り込みがいくつか定義されていることに注意しなければならない。この種の割り込みで最も注意しなければならないのが、SMI割り込みである。IA-32アーキテクチャがサポートする割り込みと例外の詳細については、『IA-32 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、下巻』の第5章「割り込みと例外の処理」を参照のこと。

割り込みあるいは例外を検出すると、プロセッサは次のいずれかの動作を行う。

- ハンドラ・プロシージャを暗黙的にコールする。
- ハンドラタスクを暗黙的にコールする。

6.4.1. 割り込み / 例外処理プロシージャのコール操作とリターン操作

割り込みあるいは例外ハンドラ・プロシージャのコールは、異なる保護レベルに対するプロシージャ・コールに似ている（6.3.6. 項「特権レベル間のコール操作とリターン操作」を参照）。割り込みあるいは例外ハンドラ・プロシージャのコールにおいては、割り込みベクタが2種のゲート（**割り込みゲート**か**トラップゲート**）のいずれかを参照する。割り込みゲートとトラップゲートは、次の情報を保持する点でコールゲートに似ている。

- アクセス権に関する情報
- ハンドラ・プロシージャを格納しているコード・セグメントのセグメント・セレクタ
- ハンドラ・プロシージャの最初の命令へのコード・セグメントのオフセット

割り込みゲートとトラップゲートとは、次の点で異なる。割り込みハンドラあるいは例外ハンドラが割り込みゲートを介してコールされた場合は、プロセッサはEFLAGS

レジスタ内の割り込み許可フラグ（IF）をクリアし、これ以降の割り込みによってハンドラの実行が妨害されるのを防ぐ。一方、ハンドラがトラップゲートを介してコールされた場合は、IF フラグのステートは変更されない。

表 6-1. 例外と割り込み

ベクタ番号	ニーモニック	説明	原因
0	#DE	除算エラー	DIV 命令と IDIV 命令。
1	#DB	デバッグ	任意のコードやデータの参照。
2		NMI	マスク不可能な外部割り込み。
3	#BP	ブレークポイント	INT 3 命令。
4	#OF	オーバーフロー	INTO 命令。
5	#BR	BOUND 範囲外	BOUND 命令。
6	#UD	無効オペコード（未定義オペコード）	UD2 命令または予約オペコード。[1]
7	#NM	デバイス使用不可能（数値演算コプロセッサなし）	浮動小数点命令または WAIT/FWAIT 命令。
8	#DF	ダブルフォルト	例外、NMI、または INTR を生成できる任意の命令。
9	#MF	コプロセッサ・セグメント・オーバーラン（予約）	浮動小数点命令。[2]
10	#TS	無効 TSS	タスクスイッチまたは TSS アクセス。
11	#NP	セグメント不在	セグメント・レジスタのロードまたはシステム・セグメントのアクセス。
12	#SS	スタック・セグメントのフォルト	スタック操作と SS レジスタのロード。
13	#GP	一般保護	任意のメモリ参照と、その他の保護チェック。
14	#PF	ページフォルト	任意のメモリ参照。
15		予約済み	
16	#MF	浮動小数点エラー（数値演算フォルト）	浮動小数点命令または WAIT/FWAIT 命令。
17	#AC	アライメント・チェック	メモリ内の任意のデータ参照。[3]
18	#MC	マシンチェック	エラーコード（存在する場合）、およびソースがモデルに依存。[4]
19	#XF	SIMD 浮動小数点例外 [5]	SIMD 浮動小数点命令。
20-31		予約済み	
32-255		マスク可能割り込み	INTR ピンによる外部割り込みまたは INT *n* 命令。

1. UD2 命令は、インテル® Pentium® Pro プロセッサで初めて導入された。

2. Intel386™ プロセッサ以降の IA プロセッサでは、この例外は生成されない。

3. この例外は、Intel486™ プロセッサで初めて導入された。

4. この例外は、インテル® Pentium® プロセッサで初めて導入され、P6 ファミリ・プロセッサで拡張された。

5. この例外は、インテル® Pentium® III プロセッサで初めて導入された。

ハンドラ・プロシージャのコード・セグメントが、現在実行されているプログラムまたはタスクと同じ特権レベルを持つ場合は、ハンドラ・プロシージャは現在のスタックを使用する。ハンドラの特権レベルの方が高い場合は、プロセッサはハンドラの特権レベル用のスタックに切り替える（スタックスイッチ）。

スタックスイッチが生じなかった場合は、割り込みあるいは例外のハンドラをコールするときに、プロセッサは次の動作を行う（図 6-5. を参照）。

1. EFLAGS、CS、および EIP の各レジスタの現在値を、この順序でスタックにプッシュする。
2. 該当する場合には、エラーコードをスタックにプッシュする。
3. 新しいコード・セグメントのセグメント・セレクタと（割り込みゲートかトラップゲートからの）新しい命令ポインタを、それぞれ CS レジスタと EIP レジスタにロードする。
4. コールが割り込みゲートを介する場合は、EFLAGS レジスタ内の IF フラグをクリアする。
5. 新しい特権レベルで、ハンドラ・プロシージャの実行を開始する。

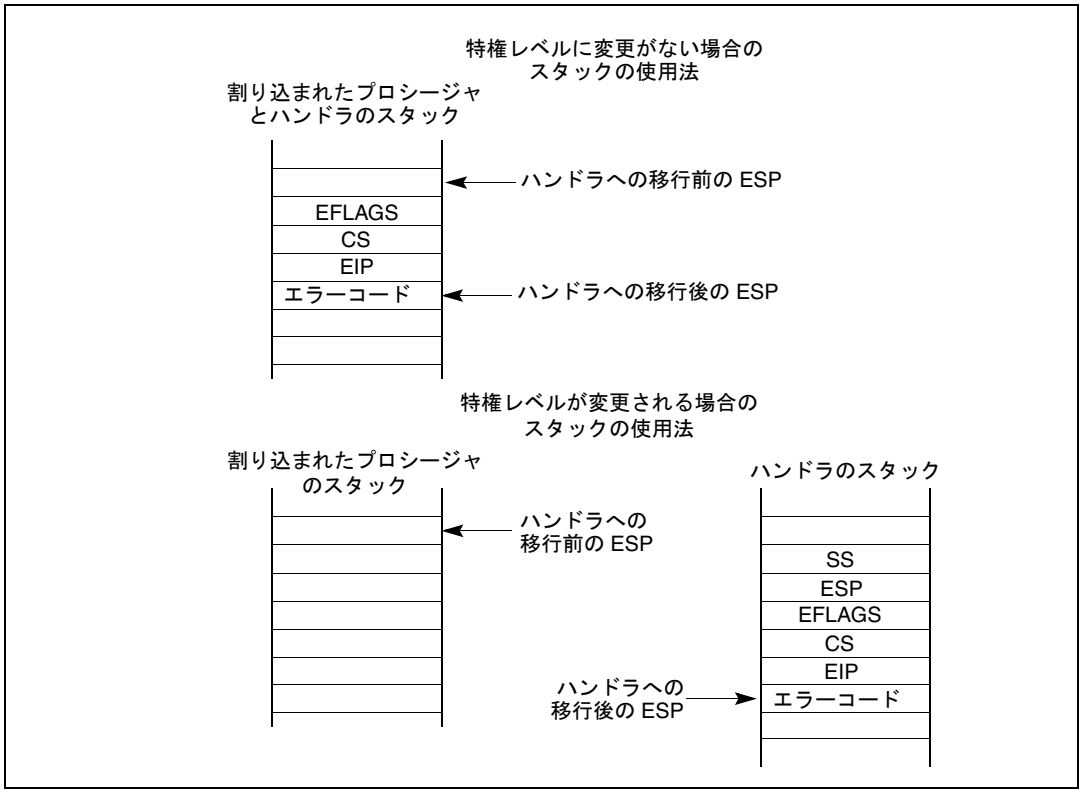


図 6-5. 割り込み / 例外処理ルーチンへの移行時のスタックの使用法

スタックスイッチが生じた場合は、プロセッサは次の動作を行う。

1. SS、ESP、EFLAGS、CS、EIP の各レジスタの現在値を一時的に内部にセーブする。
2. TSS に格納されている新しいスタック（すなわち、現在コールされている特権レベル用のスタック）のセグメント・セレクタとスタックポインタを SS レジスタと ESP レジスタにロードし、新しいスタックに切り替える。
3. 割り込みをかけられたプロセッサのスタックに対して一時的にセーブしておいた SS、ESP、EFLAGS、CS、EIP の各値を、この新しいスタックにプッシュする。
4. 該当する場合には、エラーコードを新しいスタックにプッシュする。
5. 新しいコード・セグメントのセグメント・セレクタと（割り込みゲートかトラップゲートからの）新しい命令ポインタを、それぞれ CS レジスタと EIP レジスタにロードする。
6. コールが割り込みゲートを介する場合は、EFLAGS レジスタ内の IF フラグをクリアする。
7. 新しい特権レベルで、ハンドラ・プロシージャの実行を開始する。

割り込みあるいは例外のハンドラからのリターンは、IRET 命令で開始する。IRET 命令は、割り込みをかけられたプロシージャに対して EFLAGS レジスタの内容も同時にリストアすることを除けば、far リターン命令と同じである。割り込みハンドラあるいは例外ハンドラからのリターンを、割り込みをかけられたプロシージャと同じ特権レベルで実行する場合は、プロセッサは次の動作を実行する。

1. CS レジスタと EIP レジスタに、割り込みあるいは例外が発生する前の値をリストアする。
2. EFLAGS レジスタに元の値をリストアする。
3. スタックポインタを正しくインクリメントする。
4. 割り込みをかけられたプロシージャの実行を再開する。

割り込みハンドラあるいは例外ハンドラからのリターンを、割り込みをかけられたプロシージャとは異なる特権レベルで実行する場合は、プロセッサは次の動作を実行する。

1. 特権チェックを行う。
2. CS レジスタと EIP レジスタに、割り込みあるいは例外が発生する前の値をリストアする。
3. EFLAGS レジスタに元の値をリストアする。
4. SS レジスタと ESP レジスタに、割り込みあるいは例外が発生する前の値をリストアする。これで、スタックが、割り込みをかけられたプロシージャのスタックに切り替わる。
5. 割り込みをかけれらたプロシージャの実行を再開する。

6.4.2. 割り込み / 例外ハンドラタスクのコール

割り込み / 例外ハンドラのルーチンは、個々のタスクとして実行することもできる。この場合、割り込みあるいは例外によって、ハンドラタスクへのタスクスイッチが生じる。ハンドラタスクには自身のアドレス空間が与えられ、またオプションによってアプリケーション・プログラムやタスクより高い保護レベルで実行できる。

ハンドラタスクへの切り替え（タスクスイッチ）は、**タスク・ゲート・ディスクリプタ**を参照する暗黙的なタスク・コールによって行われる。タスクゲートによって、ハンドラタスクのアドレス空間にアクセスできる。タスクスイッチの一環として、プロセッサは割り込みをかけられたプログラムやタスクについての完全なステート情報をセーブする。割り込みをかけられたプログラムやタスクのステートは、ハンドラタスクからリターンした時点でリストアされ、実行が再開される。プロセッサがハンドラタスクを介して割り込みや例外を処理するメカニズムの詳しい説明については、『IA-32 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、下巻』の第 5 章「割り込みと例外の処理」を参照のこと。

6.4.3. 実アドレスモードでの割り込みと例外の処理

実アドレスモードで動作しているときは、プロセッサは割り込みハンドラや例外ハンドラに対して暗黙的に far コールをかけることによって、割り込みや例外に応答する。プロセッサは、割り込みベクタ番号あるいは例外ベクタ番号を、割り込みテーブルへのインデックスとして使用する。割り込みテーブルには、割り込みハンドラ・プロシージャと例外ハンドラ・プロシージャへの命令ポインタが格納される。

プロセッサは、ハンドラ・プロシージャへの切り替えを実行する前に、EFLAGS、EIP、CS の各レジスタのステートをセーブすると共に、オプションでエラーコードのステートをスタック上にセーブする。

割り込みハンドラあるいは例外ハンドラからのリターンは、IRET 命令で実行される。

実アドレスモードにおける割り込みや例外の処理の詳細については、『IA-32 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、下巻』の第 16 章「8086 エミュレーション」を参照のこと。

6.4.4. INT *n*、INTO、INT 3、BOUND 命令

INT *n*、INTO、INT 3、BOUND の各命令を使用すれば、プログラムやタスクで割り込みハンドラや例外ハンドラを明示的にコールできる。INT *n* 命令では、割り込みベクタを引き数として使用するため、プログラム上で任意の割り込みハンドラをコールできる。

INTO 命令は、EFLAGS レジスタのオーバーフロー・フラグ（OF）がセットされている場合に、オーバーフロー例外（#OF）ハンドラを明示的にコールする。OF フラグは、算術命令でのオーバーフローを示すものであって、オーバーフロー例外を自動的に発生させることはない。オーバーフロー例外を明示的に発生させるには、次のいずれかの方法を使用しなければならない。

- INTO 命令を実行する。
- OF フラグをテストし、フラグがセットされている場合は、引き数に 4（オーバーフロー例外のベクタ番号）を指定して INT *n* 命令を実行する。

オーバーフロー条件を扱うこれらの方法を利用して、プログラム上で、命令ストリーム内の特定の位置でオーバーフローをテストできる。

INT 3 命令は、ブレークポイント例外（#BP）ハンドラを明示的にコールする。

BOUND 命令は、オペランドがメモリ内の定義済みの境界内にないことが検出された場合に、BOUND 範囲外例外（#BR）ハンドラを明示的にコールする。この命令は、配列などのデータ構造に対する参照をチェックする目的で用意されているものである。オーバーフロー例外と同じように、BOUND 範囲外例外は、BOUND 命令か、引き数に 5（境界チェック例外のベクタ番号）を指定して INT *n* 命令を使って明示的に発生させなければならない。プロセッサが境界チェックを行って BOUND 範囲外例外を発生させることはない。

6.4.5. 浮動小数点例外の処理

パックドまたは個々の浮動小数点値を操作するとき、IA-32 アーキテクチャは、6 種類の浮動小数点例外をサポートする。これらの例外は、x87 FPU 命令によって実行される操作中に発生することも、SSE、SSE2、SSE3 によって実行される操作中に発生することもある。x87 FPU 命令（SSE3 の FISTTP 命令を含む）が 1 つ以上の浮動小数点例外を発生させた場合は、浮動小数点エラー例外（#MF）が生成される。SSE、SEE2、SSE3 が浮動小数点例外を発生させた場合は、SIMD 浮動小数点例外（#XF）が生成される。

各浮動小数点例外の詳細と、各例外がどのように生成され、どのように処理されるかについては、以下の各項を参照のこと。

- 4.9.1. 項「浮動小数点例外条件」と 4.9.3. 項「浮動小数点例外ハンドラの一般的な動作」
- 8.4. 節「x87 FPU 浮動小数点例外処理」と 8.5. 節「x87 FPU 浮動小数点例外条件」
- 11.5.1. 項「SIMD 浮動小数点例外」

6.5. ブロック構造言語でのプロシージャ・コール

IA-32 アーキテクチャは、プロシージャ・コールを実行する代替手法として、ENTER（Enter procedure）命令と LEAVE（Leave procedure）命令を使用する手法をサポートしている。これらの命令はそれぞれ、コールされるプロシージャ用のスタックフレームを自動的に作成し、開放する。スタックフレームには、ローカル変数用にあらかじめ定義された空間と、コヒーレンシを乱すことなくコールされたプロシージャからリターンするために必要なポインタとがある。このスタックフレームによって、有効範囲規則をインプリメントすることが可能になるため、プロシージャが自身のローカル変数や、他のスタックファイル内に配置された他の変数にアクセスすることができる。

ENTER 命令と LEAVE 命令には、次に挙げる 2 つのメリットがある。

- マシン語をサポートするため、C や Pascal などのブロック構造言語をインプリメントできる。
- コンパイラが生成するコードにおいて、プロシージャの起動と終了が単純になる。

6.5.1. ENTER 命令

ENTER 命令は、ブロック構造言語において広く使用されている有効範囲規則（スコープルール）と互換性があるスタックフレームを作成する。ブロック構造言語では、プロシージャの有効範囲は、プロシージャがアクセスできる変数のセットである。有効範囲に対する規則は、言語によって異なる。これらの規則としては、プロシージャのネスト構造をベースとするもの、個別にコンパイルされるファイルへのプログラムの分割をベースとするもの、その他のモジュール化スキームをベースとするものなどがある。

ENTER 命令は、2 つのオペランドを持つ。第 1 のオペランドには、現在コールされているプロシージャについてスタックに確保される動的記憶領域をバイト数で指定する。動的記憶領域は、プロシージャがコールされる際に作成される変数（自動変数とも呼ばれる）用に割り当てられるメモリである。第 2 のオペランドは、プロシージャのレキシカル・ネスト・レベル（0～31）である。ネストレベルとは、プロシージャ・コールの階層におけるプロシージャの深さである。レキシカル・レベルは、現在実行されているプログラムやタスクの保護特権レベルやI/O特権レベルとは無関係である。

次の例に示す ENTER 命令は、スタック上に 2K バイトの動的記憶領域を割り当て、このプロシージャのスタックフレーム内で、前の 2 つのスタックフレームに対するポインタをセットアップする。

```
ENTER 2048,3
```

レキシカル・ネスト・レベルによって、前のスタックフレームから新しいスタックフレームにコピーするスタック・フレーム・ポインタの数が決まる。スタック・フレーム・ポインタは、プロシージャの変数にアクセスする際に使用される 1 ダブルワードである。プロシージャが、他のプロシージャの変数にアクセスする際に使用するスタック・フレーム・ポインタのセットは、ディスプレイと呼ばれる。このディスプレイ内の最初のダブルワードは、前のスタックフレームに対するポインタである。LEAVE 命令でこのポインタを使用して、現在のスタックフレームを廃棄して ENTER 命令の効果を元に戻せる。

プロシージャに対するディスプレイを作成した後、ENTER 命令は第 1 のパラメータで指定されたバイト数だけ ESP レジスタの内容をデクリメントし、そのプロシージャに対する動的ローカル変数を割り当てる。ESP レジスタ内のこの新しい値は、プロシージャ内のすべての PUSH 操作や POP 操作に対するスタックのトップの初期値として機能する。

プロシージャがそのディスプレイをアドレス指定できるようにするため、ENTER 命令によって、EBP レジスタはディスプレイ内の最初のダブルワードをポイントする。スタックは下方向に増えるため、このダブルワードには実際にはディスプレイ内で最高位のアドレスが入る。EBP レジスタをベースレジスタとして指定するデータ操作命令においては、データ・セグメント内ではなくスタック・セグメント内の位置を自動的にアドレス指定する。

ENTER 命令は、ネスト形式と非ネスト形式の 2 つの方法で使用できる。レキシカル・レベルが 0 の場合は、非ネスト形式が使用される。非ネスト形式では、EBP レジスタの内容がスタックにプッシュされ、ESP レジスタの内容が EBP レジスタにコピーされる。同時に、動的記憶領域を割り当てるために ESP レジスタの内容から第 1 オペランドの内容が引かれる。非ネスト形式は、スタック・フレーム・ポインタがコピーされない点でネスト形式とは異なる。ネスト形式の ENTER 命令は、第 2 オペランド（レキシカル・レベル）がゼロでない場合に使用される。

ENTER 命令の正式な定義を、次の疑似コードで示す。STORAGE は、ローカル変数用に割り当てる動的記憶領域のバイト数で、LEVEL はレキシカル・ネスト・レベルである。

```
PUSH EBP;
FRAME_PTR ← ESP;
IF LEVEL > 0
    THEN
        DO (LEVEL – 1) times
            EBP ← EBP – 4;
            PUSH Pointer(EBP); (* EBP がポイントするダブルワード *)
        OD;
    PUSH FRAME_PTR;
FI;
EBP ← FRAME_PTR;
ESP ← ESP – STORAGE;
```

（他のすべてのプロシージャがネストされる）メイン・プロシージャは、最高位のレキシカル・レベル、つまりレベル1で動作する。メイン・プロシージャがコールする最初のプロシージャは、次のレキシカル・レベル、つまりレベル2で動作する。レベル2のプロシージャは、（コンパイラが指定する固定位置にある）メイン・プログラムの変数にアクセスできる。レベル1の場合は、コピーの対象となる前回のディスプレイが存在しないため、ENTER命令によってリクエストされた動的格納領域だけがスタック上に割り当てられる。

あるプロシージャが、それより低いレキシカル・レベルにある別のプロシージャをコールする場合は、コールされるプロシージャは、コール元の変数にアクセスできる。ENTER命令により、コール元プロシージャのスタックフレームへのポインタをディスプレイに配置することで、このアクセスが可能になる。

あるプロシージャが、同じレキシカル・レベルにある別のプロシージャをコールする場合は、コールされるプロシージャにコール元の変数にアクセスさせることはできない。この場合、ENTER命令は、すでにネストされている（自身より高いレキシカル・レベルで動作している）プロシージャを参照するディスプレイ部分だけをコール元プロシージャからコピーする。新しいスタックフレームには、コール元プロシージャのスタックフレームをアドレス指定するためのポインタは含まれない。

ENTER命令は、再入可能なプロシージャを、同じレキシカル・レベルにあるプロシージャへのコールとして処理する。この場合、再入可能なプロシージャが繰り返されるたびに、そのプロシージャの変数と、それがネストされているプロシージャの変数しかアドレス指定することはできない。再入可能なプロシージャは、常に自身の変数についてはアドレス指定でき、以前に繰り返されたスタックフレームへのポインタは不要である。

ENTER命令は、当該プロシージャより高いレキシカル・レベルにあるプロシージャのスタック・フレーム・ポインタだけをコピーすることで、プロシージャが同じレキシカル・レベルの変数ではなく、それより高いレキシカル・レベルの変数だけにアクセスすることを可能にしている（図6-6.を参照）。

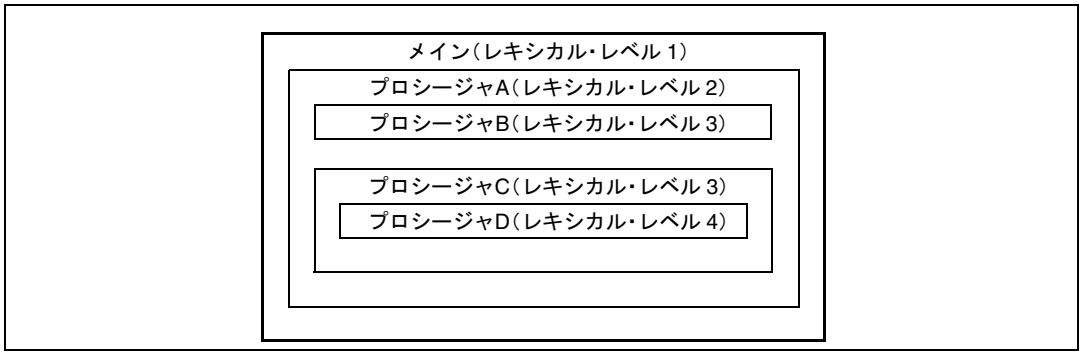


図 6-6. ネストされたプロシージャ

ブロック構造言語では、ENTER 命令で定義されたレキシカル・レベルを使用して、ネストされたプロシージャの変数に対するアクセスを制御する。例えば、図 6-6. において、プロシージャ A がプロシージャ B を、プロシージャ B がプロシージャ C をコールする場合、プロシージャ C はメイン・プロシージャとプロシージャ A の変数にはアクセスできるが、同じレキシカル・レベルのプロシージャ B の変数にはアクセスできない。図 6-6. に示すネストされたプロシージャにおいて、変数へのアクセスを定義すると次のようになる。

1. メインは、固定位置に変数を持つ。
2. プロシージャ A は、メインの変数だけにアクセスできる。
3. プロシージャ B は、プロシージャ A とメインの変数だけにアクセスできる。プロシージャ B は、プロシージャ C やプロシージャ D の変数にはアクセスできない。
4. プロシージャ C は、プロシージャ A とメインの変数だけにアクセスできる。プロシージャ C は、プロシージャ B やプロシージャ D の変数にはアクセスできない。
5. プロシージャ D は、プロシージャ C、プロシージャ A、メインの変数にアクセスできる。プロシージャ D は、プロシージャ B の変数にはアクセスできない。

図 6-7. において、メイン・プロシージャの先頭にある ENTER 命令によって、メインに対して動的記憶領域としてダブルワードが 3 つ作成されるが、他のスタックフレームからポインタをコピーすることはしない。ディスプレイ内の最初のダブルワードは、ENTER 命令が実行される前に EBP レジスタにあった最後の値のコピーを保持する。2 番目のダブルワードは、ENTER 命令後の EBP レジスタの内容のコピーを保持する。命令が実行された後、EBP レジスタはスタックにプッシュされた最初のダブルワードをポイントし、また ESP レジスタはスタックフレーム内の最後のダブルワードをポイントする。

メインがプロシージャ A をコールすると、ENTER 命令によって新しいディスプレイが作成される（図 6-8. を参照）。最初のダブルワードは、メインの EBP レジスタに保持されていた最後の値である。2 番目のダブルワードは、メインのスタックフレームに対するポインタであり、メインのディスプレイの 2 番目のダブルワードからコピーされたものである。これは、メインの EBP レジスタに保持されていた最後の値のコピーでもある。プロシージャ A は、メインがレベル 1 にあるため、メインの変数にアクセスできる。

したがって、メインが使用する動的記憶領域のベースアドレスは、EBP レジスタ内の現在のアドレスに、メインの EBP レジスタに保存されている内容の 4 バイトを加えたものになる。メインに対する動的変数はすべて、この値から正の固定オフセット位置にある。

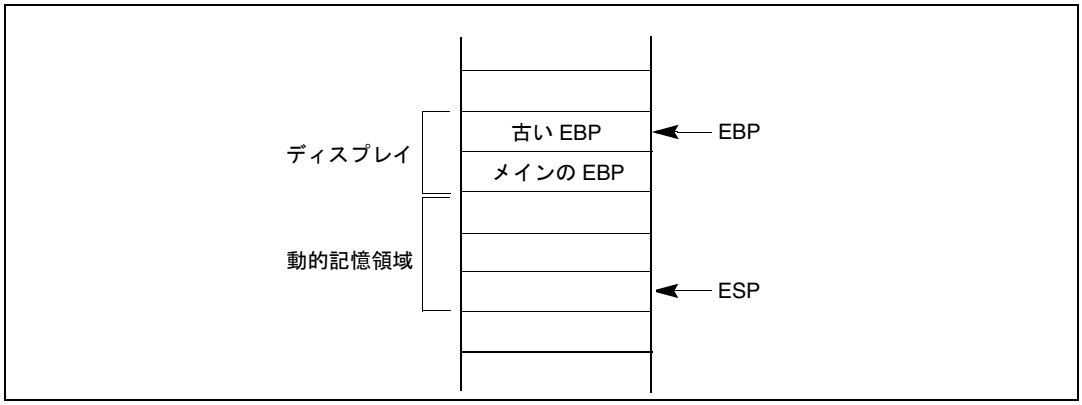


図 6-7. メイン・プロシージャに移行後のスタックフレーム

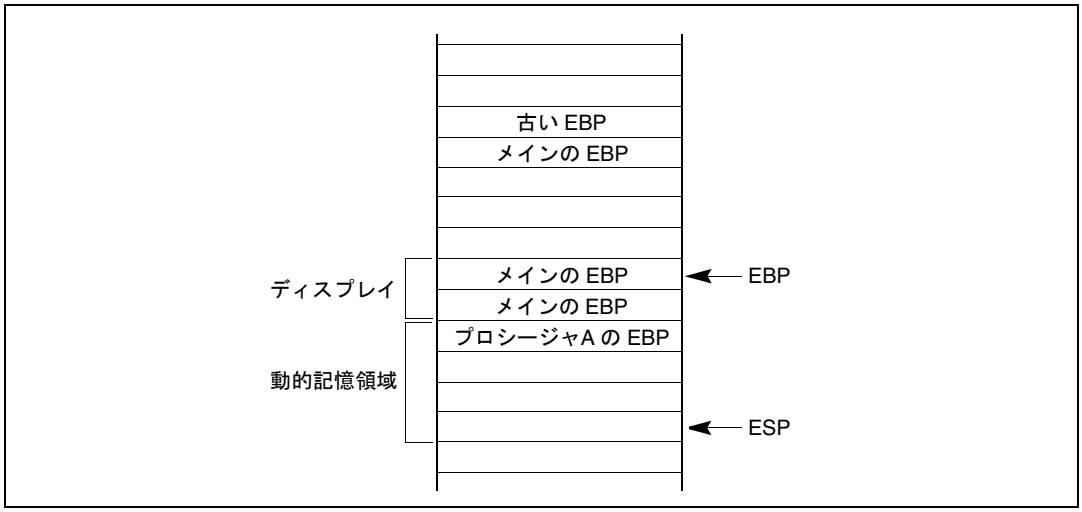


図 6-8. プロシージャ A に移行後のスタックフレーム

プロシージャ A がプロシージャ B をコールすると、ENTER 命令によって新しいディスプレイが作成される（図 6-9. を参照）。最初のダブルワードは、プロシージャ A の EBP レジスタ内にある最後の値のコピーを保持する。2 番目と 3 番目のダブルワードは、プロシージャ A のディスプレイ内にある 2 つのスタック・フレーム・ポインタのコピーである。プロシージャ B は、自身のディスプレイ内のスタック・フレーム・ポインタを使用すれば、プロシージャ A とメイン内の変数にアクセスできる。

プロシージャ B がプロシージャ C をコールすると、ENTER 命令によってプロシージャ C 用に新しいディスプレイが作成される（図 6-10. を参照）。最初のダブルワードは、プロシージャ B の EBP レジスタ内にある最後の値のコピーを保持する。この値は、プロシージャ B のスタックフレームをリストアする際に LEAVE 命令が使用する。2 番目と 3 番目のダブルワードは、プロシージャ A のディスプレイ内にある 2 つのスタック・

フレーム・ポインタのコピーである。プロシージャ C がプロシージャ B の次のレキシカル・レベルにある場合は、4 番目のダブルワードがコピーされる。これは、プロシージャ B のローカル変数に対するスタック・フレーム・ポインタである。

プロシージャ B とプロシージャ C は同じレベルにあるため、プロシージャ C にはプロシージャ B の変数にアクセスする必要がないことに注意しなければならない。これは、プロシージャ C がプロシージャ B から完全に分離されているということではない。プロシージャ C はプロシージャ B によってコールされたので、リターン用のスタックフレームのポインタはプロシージャ B のスタックフレームのポインタになる。また、プロシージャ B は、スタックを介するか、両プロシージャに対してグローバルな変数（すなわち、両プロシージャの有効範囲内にある変数）を介してプロシージャ C にパラメータを渡すこともできる。

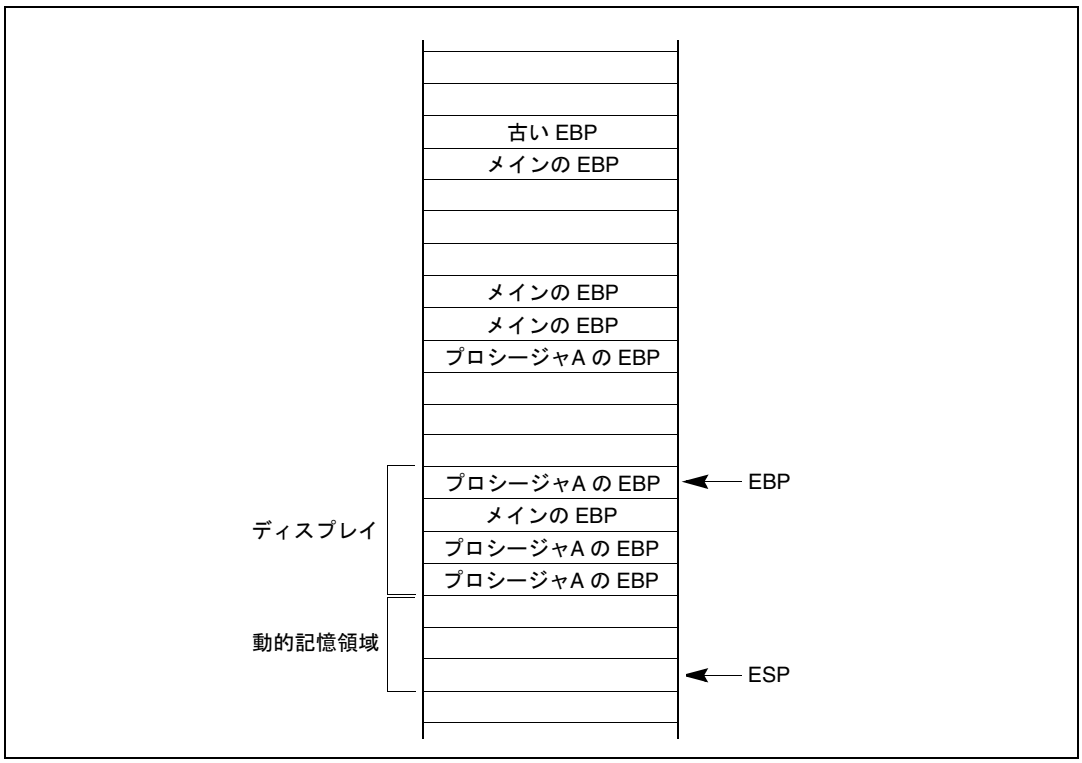


図 6-9. プロシージャ B に移行後のスタックフレーム

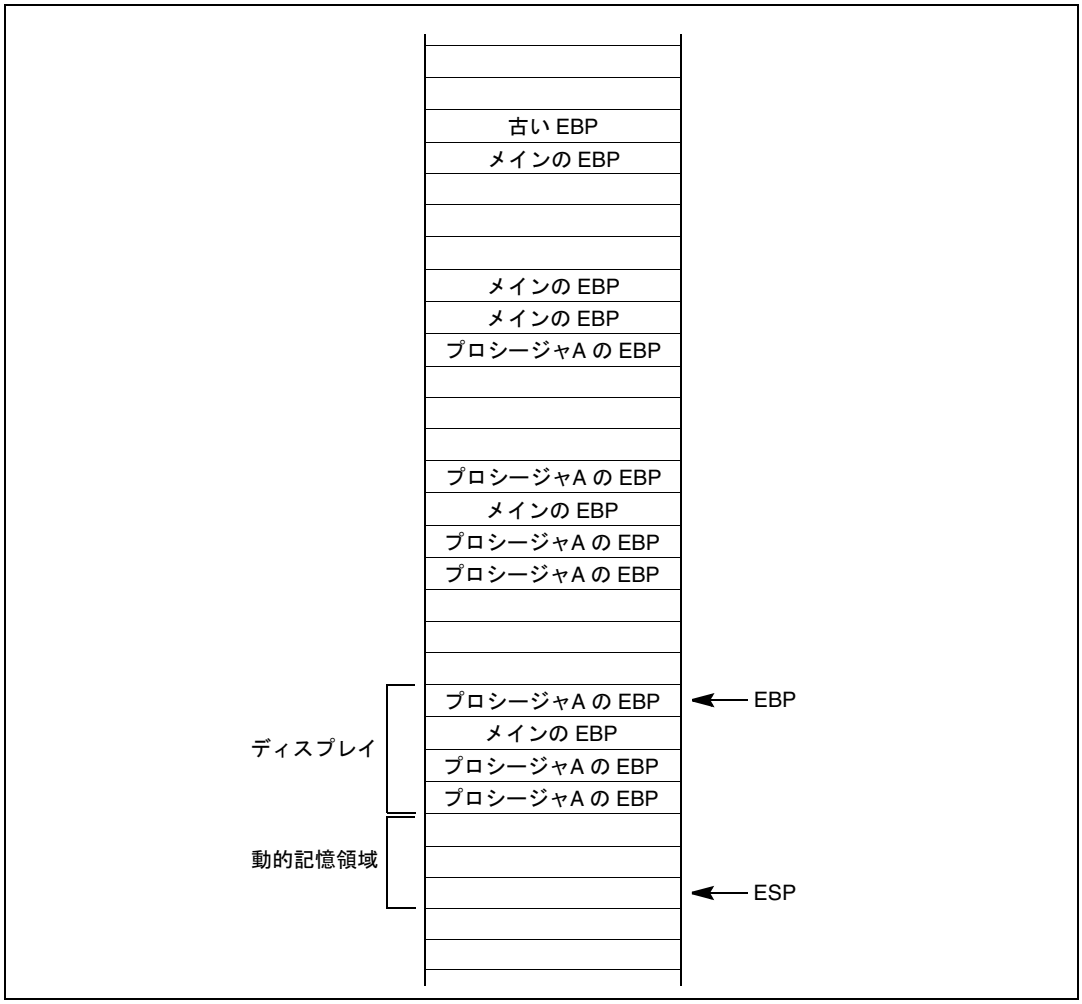


図 6-10. プロシージャ C に移行後のスタックフレーム

6.5.2. LEAVE 命令

LEAVE 命令は、直前の ENTER 命令と逆の動作を行う。LEAVE 命令はオペランドは持たず、EBP レジスタの内容を ESP レジスタにコピーし、プロシージャに割り当てられたスタック空間すべてを開放する。次に、スタックから EBP レジスタの古い値をリストアする。このとき同時に、ESP レジスタも元の値にリストアされる。したがって、LEAVE 命令の後で RET 命令を使用すれば、プロシージャで使用するためにコール元プログラム上でスタックにプッシュしておいた引き数とリターンアドレスを削除できる。

7

汎用命令による プログラミング

第 7 章
汎用命令による
プログラミング

7

汎用命令は、IA-32 命令のうち、インテル® IA-32 プロセッサ向けの基本命令セットに相当する。これらの命令は、最初の IA-32 プロセッサ（インテル® 8086 とインテル® 8088）で IA-32 アーキテクチャに導入された。これ以降の IA-32 プロセッサ・ファミリ（インテル® 286 プロセッサ、Intel386™ プロセッサ、Intel486™ プロセッサ、インテル® Pentium® プロセッサ、インテル® Pentium® Pro プロセッサ、インテル® Pentium® II プロセッサ）で、この汎用命令セットに対して新しい命令が追加された。

汎用命令は、整数データ型、ポインタデータ型、BCD データ型に対して、基本的なデータ転送、メモリアドレス指定、算術演算と論理演算、プログラム・フロー制御、入出力、およびストリング操作を実行する。

本章では、汎用命令の概要について説明する。これらの命令についての詳細は、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」を参照のこと。

7.1. 汎用命令のプログラミング環境

汎用命令のプログラミング環境は、IA-32 アーキテクチャの基本実行環境を構成する一連のレジスタおよびアドレス空間（図 7-1. を参照）と、一連のデータ型で構成される。基本実行環境には、以下の項目が含まれる。

- **汎用レジスタ。** 8 つの 32 ビット汎用レジスタ（図 3-4. を参照）と既存の IA-32 アドレス指定モードを組み合わせて、メモリ内のオペランドをアドレス指定する。これらのレジスタは、EAX、EBX、ECX、EDX、EBP、ESI、EDI、ESP の名前で参照される。
- **セグメント・レジスタ。** 6 つの 16 ビット・セグメント・レジスタが、メモリへのアクセスに使用されるセグメント・ポインタを格納する。これらのレジスタは、CS、DS、SS、ES、FS、GS の名前で参照される。
- **EFLAGS レジスタ。** この 32 ビット・レジスタ（図 3-7. を参照）は、基本的な算術演算、比較演算、およびシステム操作を制御し、各操作のステータスを示す。
- **EIP レジスタ。** この 32 ビット・レジスタは、現在の命令ポインタを格納する。

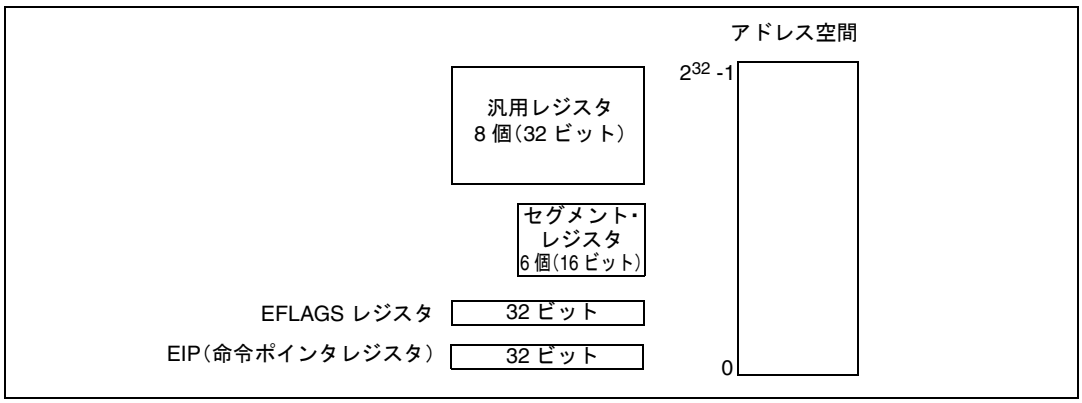


図 7-1. 汎用命令の基本実行環境

汎用命令は、以下のデータ型を操作する。

- バイト、ワード、ダブルワード（図4-1.を参照）
- 符号付きおよび符号なしバイト、ワード、ダブルワード整数（図4-3.を参照）
- nearポインタとfarポインタ（図4-4.を参照）
- ビット・フィールド（図4-5.を参照）
- BCD整数（図4-8.を参照）

7.2. 汎用命令の概要

汎用命令は、以下のサブグループに分けられる。

- データ転送命令
- 2進算術命令
- 10進算術命令
- 論理演算命令
- シフト命令とローテート命令
- ビット命令とバイト命令
- 制御転送命令
- ストリング命令
- I/O 命令
- ENTER 命令と LEAVE 命令
- フラグ制御命令

- セグメント・レジスタ命令
- その他の命令

汎用命令のサブグループの簡単な一覧は、5.1.節「汎用命令」を参照のこと。

7.2.1. データ転送命令

データ転送命令は、メモリとプロセッサ・レジスタの間およびレジスタ同士の間で、バイト、ワード、ダブルワード、またはクワッドワードを転送する。説明のため、これらの命令は次の下位のサブグループに分けられる。

- 汎用データ転送命令
- 交換命令
- スタック操作命令
- 型変換命令

7.2.1.1. 汎用データ転送命令

転送命令。MOV（move）命令と CMOVcc（conditional move）命令は、メモリとレジスタの間またはレジスタ同士の間でデータを転送する。

MOV 命令は、メモリとプロセッサ・レジスタの間の基本的なデータロード/データストア操作と、レジスタ間のデータ転送操作を実行する。この命令は、表 7-1. に示したパス上のデータ転送を処理する（コントロール・レジスタおよびデバッグレジスタとの間のデータの転送については、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「MOV - コントロール・レジスタとの間の転送」と「MOV - デバッグレジスタとの間の転送」を参照）。

MOV 命令は、あるメモリ・ロケーションから他のメモリ・ロケーションにデータを転送することや、あるセグメント・レジスタから他のセグメント・レジスタにデータを転送することはできない。メモリからメモリへの転送は、MOVS（string move）命令で実行する必要がある（7.2.8. 項「ストリングの操作」を参照）。

条件付き転送命令。CMOVcc 命令は、EFLAGS レジスタ内のステータス・フラグの状態をチェックし、フラグが指定の状態（または条件）である場合に転送動作を実行する命令グループである。これらの命令を使用して、メモリから汎用レジスタに、またはある汎用レジスタから他の汎用レジスタに、16 ビットまたは 32 ビットの値を転送できる。各命令でテストされるフラグの状態は、その命令に関連する条件コード（cc ）で指定される。指定の条件が満たされない場合は、転送は実行されず、CMOVcc 命令の次の命令からプログラムの実行が再開される。

表 7-1. 転送命令の動作

データ転送のタイプ	ソース → デスティネーション
メモリからレジスタへ	メモリ・ロケーション → 汎用レジスタ メモリ・ロケーション → セグメント・レジスタ
レジスタからメモリへ	汎用レジスタ → メモリ・ロケーション セグメント・レジスタ → メモリ・ロケーション
レジスタ同士の間	汎用レジスタ → 汎用レジスタ 汎用レジスタ → セグメント・レジスタ セグメント・レジスタ → 汎用レジスタ 汎用レジスタ → コントロール・レジスタ コントロール・レジスタ → 汎用レジスタ 汎用レジスタ → デバッグレジスタ デバッグレジスタ → 汎用レジスタ
即値データからレジスタへ	即値 → 汎用レジスタ
即値データからメモリへ	即値 → メモリ・ロケーション

表 7-2. は、CMOVcc 命令のニーモニックと、各命令でテストされる条件を示している。CMOVcc 命令のニーモニックは、"CMOV" に条件コード・ニーモニックを付加したものである。表 7-2. にペアで示した命令（例えば、CMOVA/CMOVNBE）は、同じ命令の別名である。アセンブラは、プログラム・リストが読みやすくなるように、これらの別名を用意している。

CMOVcc 命令によって、小さな IF 構造を削減するのに便利である。また、CMOVcc 命令を使用して、IF 文による分岐のオーバーヘッドとプロセッサによる分岐の予測ミスの可能性を減らすことができる。

これらの条件付き転送命令は、P6 ファミリ・プロセッサ、インテル® Pentium® 4 プロセッサ、インテル® Xeon™ プロセッサでのみサポートされている。ソフトウェアは、CPUID 命令を使用してプロセッサの機能情報をチェックすることによって、CMOVcc 命令がサポートされているかどうかを確認できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「CPUID - CPU の識別」を参照）。

7.2.1.2. 交換命令

交換命令は、1 つ以上のオペランドの内容を入れ替える。場合によっては、LOCK 信号のアサートや、EFLAGS レジスタ内のフラグの変更などの追加の操作も実行する。

XCHG（exchange）命令は、2 つのオペランドの内容を入れ替える。この命令は、3 つの MOV 命令と同じ効果を持つが、一方のオペランドをロードする間に他方のオペランドの内容を保存するための一時的なロケーションを必要としない。XCHG 命令でメモリ・オペランドを処理するときは、プロセッサの LOCK 信号が自動的にアサートされる。この命令は、プロセスの同期をとるためにセマフォまたは同様のデータ構造を実装するのに便利である。バスロックについての詳細は、『IA-32 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、下巻』の第7章の「バスロック」を参照。

BSWAP（byte swap）命令は、32 ビット・レジスタ・オペランドのバイト・オーダを反転する。ビット位置0～7は24～31で置き換えられ、ビット位置8～15は16～23で置き換えられる。この命令を2回続けて実行すると、レジスタは前と同じ値になる。BSWAP 命令は、「ビッグ・エンディアン」データ・フォーマットと「リトル・エンディアン」データ・フォーマットの変換に便利である。また、この命令によって、10進算術演算の実行を高速化できる（XCHG 命令を使用して、ワード内の上位バイトと下位バイトを入れ替えられる）。

表 7-2. 条件付き転送命令

命令ニーモニック	ステータス・フラグの状態	条件の説明
符号なし条件付き転送		
CMOVA/CMOVNBE	(CF or ZF)=0	より大きい / より小さくなく等しくない
CMOVAE/CMOVNB	CF=0	より大きいか等しい / より小さくない
CMOVNC	CF=0	キャリーなし
CMOVB/CMOVNAE	CF=1	より小さい / より大きくなく等しくない
CMOVC	CF=1	キャリー
CMOVBE/CMOVNA	(CF or ZF)=1	より小さいか等しい / より大きくない
CMOVE/CMOVZ	ZF=1	等しい / ゼロ
CMOVNE/CMOVNZ	ZF=0	等しくない / ゼロでない
CMOVP/CMOVPE	PF=1	パリティ / 偶数パリティ
CMOVNP/CMOVPO	PF=0	パリティなし / 奇数パリティ
符号付き条件付き転送		
CMOVGE/CMOVNL	(SF xor OF)=0	より大きいか等しい / より小さくない
CMOVL/CMOVNGE	(SF xor OF)=1	より小さい / より大きくなく等しくない
CMOVLE/CMOVNG	((SF xor OF) or ZF)=1	より小さいか等しい / より大きくない
CMOVO	OF=1	オーバーフロー
CMOVNO	OF=0	オーバーフローなし
CMOVS	SF=1	符号（負）
CMOVNS	SF=0	符号なし（負でない）

XADD（exchange and add）命令は、2つのオペランドを入れ替えて、2つのオペランドの和をデスティネーション・オペランドに格納する。EFLAGS レジスタ内のステータスフラグは、加算の結果を示す。マルチプロセッサ・システムでは、この命令とLOCK プリフィックス（『IA-32 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、中巻 A』の第3章「命令セット・リファレンス A-M」の「LOCK - LOCK# 信号アサート・プリフィックス」を参照）を組み合わせて、複数のプロセッサに1つの DO ループを実行できる。

CMPXCHG（compare and exchange）命令とCMPXCHG8B（compare and exchange 8 bytes）命令を使用して、複数のプロセッサを使用するシステム内で、動作の同期をとることができる。CMPXCHG命令は、3つのオペランド（レジスタ内のソース・オペランド、EAXレジスタ内のもう1つのソース・オペランド、およびデスティネーション・オペランド）を必要とする。デスティネーション・オペランド内の値とEAXレジスタの値が等しい場合は、デスティネーション・オペランドは他のソース・オペランドの値（EAXレジスタ内にない値）で置き換えられる。それ以外の場合は、デスティネーション・オペランドの元の値がEAXレジスタにロードされる。EFLAGSレジスタのステータスフラグは、EAXレジスタの値からデスティネーション・オペランドを引くことによって得られる結果を反映する。

CMPXCHG命令は、セマフォのテストと変更によく使用される。この命令は、セマフォが空いているかどうかをチェックする。セマフォが空いている場合は、そのセマフォは割り当て済みとしてマークされる。セマフォが空いていない場合は、そのセマフォは現在のオーナのIDを取得する。これらはすべて、割り込みをかけられない1つの動作として実行される。シングルプロセッサ・システムでは、CMPXCHG命令を使用すると、複数の命令を実行してセマフォのテストと変更を行う前に、保護レベル0に切り替えて割り込みを無効にする必要がなくなる。

マルチプロセッサ・システムでは、CMPXCHG命令とLOCKプリフィックスを組み合わせて、比較操作と交換操作をアトミックに実行できる（アトミック操作についての詳細は、『IA-32 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、下巻』の第7章の「ロックされたアトミック操作」を参照）。

CMPXCHG8B命令も、3つのオペランド（EDX:EAXレジスタ内の64ビット値、ECX:EBXレジスタ内の64ビット値、メモリ内のデスティネーション・オペランド）を必要とする。この命令は、EDX:EAXレジスタ内の64ビット値とデスティネーション・オペランドを比較する。2つの値が等しい場合は、ECX:EBXレジスタ内の64ビット値がデスティネーション・オペランドに格納される。EDX:EAXレジスタとデスティネーションが等しくない場合は、デスティネーションがEDX:EAXレジスタにロードされる。CMPXCHG8B命令とLOCKプリフィックスを組み合わせれば、この操作をアトミックに実行できる。

7.2.1.3. スタック操作命令

PUSH、POP、PUSHA（push all registers）、POPA（pop all registers）命令は、スタックとの間でデータを転送する。PUSH命令は、（ESPレジスタ内の）スタックポインタをデクリメントし、ソース・オペランドをスタックのトップにコピーする（図7-2.を参照）。この命令は、メモリ・オペランド、即値オペランド、レジスタ・オペランド（セグメント・レジスタを含む）を操作する。PUSH命令は、通常は、プロシージャを呼び出す前にスタック上にパラメータを置くのに使用される。また、PUSH命令を使用して、スタック上に一時的変数用の空間を確保できる。

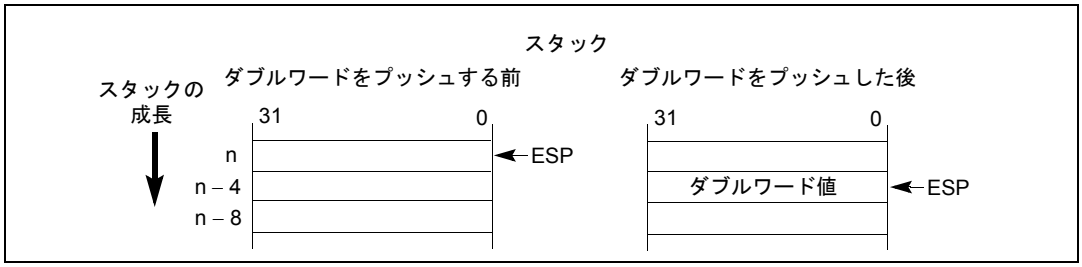


図 7-2. PUSH 命令の動作

PUSHA 命令は、8 つの汎用レジスタの内容をスタック上に保存する（図 7-3. を参照）。この命令によって、汎用レジスタの内容の保存に必要な命令の数が減り、プロシージャ・コールが簡単になる。レジスタは、EAX、ECX、EDX、EBX、EAX がプッシュされる前の ESP の初期値、EBP、ESI、EDI の順に、スタック上にプッシュされる。

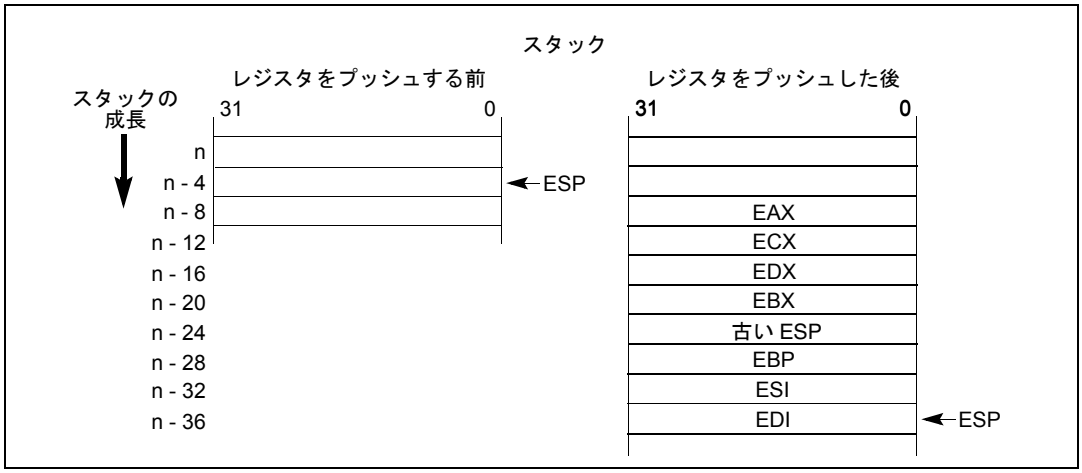


図 7-3. PUSHA 命令の動作

POP 命令は、(ESP レジスタによって指定される）スタックの現在のトップにあるワードまたはダブルワードを、デスティネーション・オペランドによって指定される位置にコピーする。次に、ESP レジスタをインクリメントして、スタックの新しいトップを指定する（図 7-4. を参照)。デスティネーション・オペランドは、汎用レジスタ、セグメント・レジスタ、またはメモリ・ロケーションを指定できる。

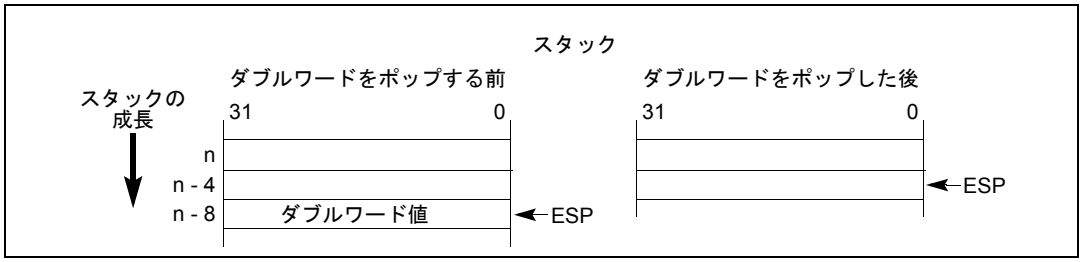


図 7-4. POP 命令の動作

POPA 命令は、PUSHA 命令の効果を逆にしたものである。この命令は、スタックのトップから汎用レジスタ（ESP レジスタを除く）に、8 つのワードまたはダブルワードをポップする（図 7-5. を参照）。オペランド・サイズ属性が 32 の場合は、スタック上のダブルワードが、EDI、ESI、EBP、無視されるダブルワード、EBX、EDX、ECX、EAX の順に、レジスタに転送される。ESP レジスタは、スタックをポップする動作によってリストアされる。オペランド・サイズ属性が 16 の場合は、スタック上のワードが、DI、SI、BP、無視されるワード、BX、DX、CX、AX の順に、レジスタに転送される。

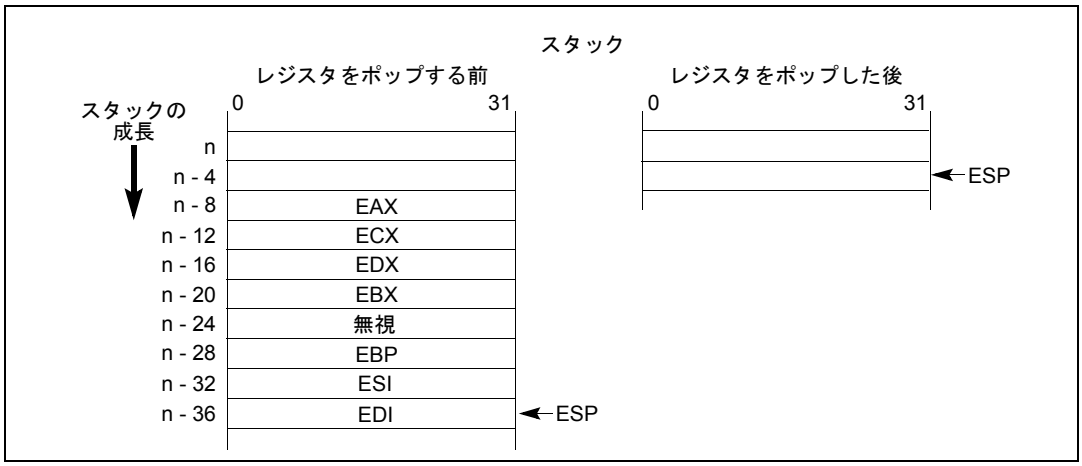


図 7-5. POPA 命令の動作

7.2.1.4. 型変換命令

型変換命令は、バイトからワードへ、ワードからダブルワードへ、ダブルワードからクワッドワードへのデータ型変換を実行する。これらの命令は、符号拡張を実行する。したがって、整数をより大きな整数フォーマットに変換するのに特に便利である（図 7-6. を参照）。

型変換命令には、単純変換と、転送後変換の 2 種類がある。

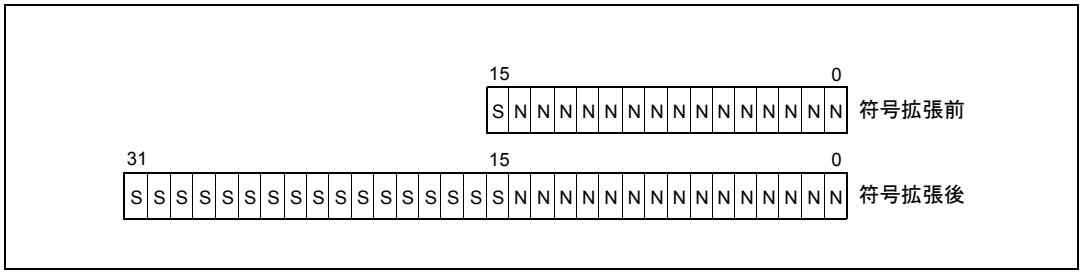


図 7-6. 符号拡張

単純変換。CBW（convert byte to word）、CWDE（convert word to doubleword extended）、CWD（convert word to doubleword）、CDQ（convert doubleword to quadword）命令は、符号拡張を実行して、ソース・オペランドのサイズを2倍にする。

CBW命令は、ALレジスタ内のバイトの符号（ビット7）を、AXレジスタの上位バイトの各ビット位置にコピーする。CWDE命令は、AXレジスタ内のワードの符号（ビット15）を、EAXレジスタの上位ワードの各ビット位置にコピーする。

CWD命令は、AXレジスタ内のワードの符号（ビット15）を、DXレジスタの各ビット位置にコピーする。CWQ命令は、EAXレジスタ内のダブルワードの符号（ビット31）を、EDXレジスタの各ビット位置にコピーする。CWD命令を使用して、ワード除算の前に、ワードからダブルワード被除数を作成できる。CDQ命令を使用して、ダブルワード除算の前に、ダブルワードからクワッドワード被除数を作成できる。

転送と符号拡張またはゼロ拡張。MOVSX（move with sign extension）命令とMOVZX（move with zero extension）命令は、ソース・オペランドをレジスタ内に転送し、符号拡張を実行する。

MOVSX命令は、図7-6.に示すように、ソース・オペランドを符号で拡張することによって、8ビット値を16ビット値に拡張したり、8ビット値または16ビット値を32ビット値に拡張したりする。MOVZX命令は、ソース・オペランドをゼロで拡張することによって、8ビット値を16ビット値に拡張したり、8ビット値または16ビット値を32ビット値に拡張したりする。

7.2.2. 2進算術命令

2進算術命令は、符号付きまたは符号なし2進整数としてコード化された、8ビット、16ビット、32ビットの数値データを操作する。2進算術命令は、10進（BCD）値を操作するアルゴリズムにも使用される。

説明のため、これらの命令は次の下位のサブグループに分けられる。

- 加算命令と減算命令

- インクリメント命令とデクリメント命令
- 比較命令と符号変更命令
- 乗算命令と除算命令

7.2.2.1. 加算命令と減算命令

ADD（add integers）、ADC（add integers with carry）、SUB（subtract integers）、および SBB（subtract integers with borrow）命令は、符号付きまたは符号なし整数オペランドの加算と減算を実行する。

ADD 命令は、2 つの整数オペランドの和を計算する。

ADC 命令は、2 つの整数オペランドの和を計算し、CF フラグがセットされている場合は 1 を加える。この命令は、数値を段階的に加算するとき、キャリーを伝搬するのに使用される。

SUB 命令は、2 つの整数オペランドの差を計算する。

SBB 命令は、2 つの整数オペランドの差を計算し、CF フラグがセットされている場合は 1 を引く。この命令は、数値を段階的に減算するとき、ボローを伝搬するのに使用される。

7.2.2.2. インクリメント命令とデクリメント命令

INC（increment）命令は、符号なし整数オペランドに 1 を加える。DEC（decrement）命令は、符号なし整数オペランドから 1 を引く。これらの命令は、主にカウンタを実装するときに使用される。

7.2.2.3. 比較命令と符号変更命令

CMP（compare）命令は、2 つの整数オペランドの差を計算し、その結果に基づいて、OF、SF、ZF、AF、PF、CF フラグを更新する。ソース・オペランドは変更されず、結果も保存されない。CMP 命令は、通常は、J*cc*（jump）命令または SET*cc*（byte set on condition）命令と組み合わせて使用される。この場合、J*cc* 命令と SET*cc* 命令は、CMP 命令の結果に基づいて処理を実行する。

NEG（negate）命令は、符号付き整数オペランドをゼロから引く。NEG 命令によって、2 の補数オペランドの絶対値を変えずに、符号だけを変更できる。

7.2.2.4. 乗算命令と除算命令

IA-32プロセッサは、MUL（unsigned multiply）とIMUL（signed multiply）の2つの乗算命令と、DIV（unsigned divide）とIDIV（signed divide）の2つの除算命令を備えている。

MUL命令は、2つの符号なし整数オペランドを乗算する。計算の結果は、ソース・オペランドの2倍のサイズになる（例えば、ワード・オペランドを乗算した結果はダブルワードになる）。

IMUL命令は、2つの符号付き整数オペランドを乗算する。計算の結果は、ソース・オペランドの2倍のサイズになるが、ソース・オペランドのサイズに合わせて切り捨てられる場合もある（『IA-32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、中巻 A』の第3章「命令セット・リファレンス A-M」の「IMUL - 符号付き乗算」を参照）。

DIV命令は、1つの符号なしオペランドをもう1つの符号なしオペランドで除算し、商と余りを返す。

IDIV命令は、DIV命令と同じ処理を実行するが、符号付き除算を実行する点が異なる。

7.2.3. 10進算術命令

10進算術演算は、2進算術命令ADD、SUB、MUL、DIV（7.2.2.項「2進算術命令」を参照）と、10進算術命令を組み合わせて実行される。10進算術命令は、以下の操作を実行する。

- 直前の2進算術演算の結果を調整して、有効なBCDの結果を求める。
- 有効なBCDの結果が得られるように、次の2進算術演算のオペランドを調整する。

10進算術命令は、パックドBCD値とアンパックBCD値を操作する。説明のため、これらの命令は次の下位のサブグループに分けられる。

- パックドBCD調整命令
- アンパックBCD調整命令

7.2.3.1. パックドBCD調整命令

DAA（decimal adjust after addition）命令とDAS（decimal adjust after subtraction）命令は、パックドBCD整数に対して実行された演算の結果を調整する（4.7.節「BCDおよびパックドBCD整数」を参照）。2つのパックドBCD値を加算するには、2つの命令が必要である。つまり、ADD命令を実行し、その後にDAA命令を実行する必要がある。ADD命令は、2つの値を加算し（2進加算）、その結果をALレジスタに格納する。

DAA 命令は、AL レジスタ内の値を調整して有効な 2 ケタのパックド BCD 値を求め、加算によって 10 進キャリーが発生した場合は、CF フラグをセットする。

同様に、パックド BCD 値からパックド BCD 値を引くには、SUB 命令を実行し、その後に DAS 命令を実行する必要がある。SUB 命令は、BCD 値から BCD 値を引き（2 進減算）、その結果を AL レジスタに格納する。DAS 命令は、AL レジスタ内の値を調整して有効な 2 ケタのパックド BCD 値を求め、減算によって 10 進ボローが発生した場合は、CF フラグをセットする。

7.2.3.2. アンパック BCD 調整命令

AAA（ASCII adjust after addition）、AAS（ASCII adjust after subtraction）、AAM（ASCII adjust after multiplication）、AAD（ASCII adjust before division）命令は、アンパック BCD 値に対して実行された算術演算の結果を調整する（4.7. 節「BCD およびパックド BCD 整数」を参照）。これらの命令はすべて、調整される値が AL レジスタ（AAD 命令では、AL レジスタと AH レジスタ）に格納されているものとする。

AAA 命令は、2 つのアンパック BCD 値の加算後に、AL レジスタの内容を調整する。この命令は、AL レジスタ内の 2 進値を 10 進値に変換し、結果をアンパック BCD フォーマットで AL レジスタに格納する（AL レジスタの下位 4 ビットに 10 進数が格納され、上位 4 ビットはクリアされる）。加算によって 10 進キャリーが発生した場合は、CF フラグがセットされ、AH レジスタの内容が 1 だけインクリメントされる。

AAS 命令は、2 つのアンパック BCD 値の減算後に、AL レジスタの内容を調整する。この場合も、2 進値がアンパック BCD 値に変換される。10 進減算によってボローが発生した場合は、CF フラグがセットされ、AH レジスタの内容が 1 だけデクリメントされる。

AAM 命令は、2 つのアンパック BCD 値の乗算後に、AL レジスタの内容を調整する。この命令は、AL レジスタ内の 2 進値を 10 進値に変換し、結果の最下位の桁を（アンパック BCD フォーマットで）AL レジスタに格納し、最上位の桁（存在する場合）を（アンパック BCD フォーマットで）AH レジスタに格納する。

AAD 命令は、DIV 命令を使って 2 ケタの BCD 値を除算したとき、有効なアンパック BCD の結果が得られるように、その 2 ケタの BCD 値を調整する。この命令は、レジスタ AH（最上位の桁）およびレジスタ AL（最下位の桁）内の BCD 値を 2 進値に変換し、結果をレジスタ AL に格納する。レジスタ AL 内の値をアンパック BCD 値で除算すると、その商と余りは自動的にアンパック BCD フォーマットでコード化される。

7.2.4. 論理演算命令

論理演算命令 AND、OR、XOR（exclusive or）、NOT は、それぞれの名前に対応する標準的なブール演算を実行する。AND、OR、XOR 命令は、2 つのオペランドを必要とする。NOT 命令は、1 つのオペランドを操作する。

7.2.5. シフト命令とローテート命令

シフト命令とローテート命令は、オペランド内のビットを移動する。説明のため、これらの命令は次の下位のサブグループに分けられる。

- ビットのシフト命令
- ビットのダブルシフト命令（オペランド間の転送）
- ビットのローテート命令

7.2.5.1. シフト命令

SAL（shift arithmetic left）、SHL（shift logical left）、SAR（shift arithmetic right）、SHR（shift logical right）命令は、バイト、ワード、またはダブルワード内のビットの算術シフトまたは論理シフトを実行する。

SAL 命令と SHL 命令は、同じ操作を実行する（図 7-7. を参照）。これらの命令は、ソース・オペランドを 1 ～ 31 ビット位置だけ左にシフトする。空いたビット位置はクリアされる。オペランドの外にシフトされた最後のビットは、CF フラグにロードされる。

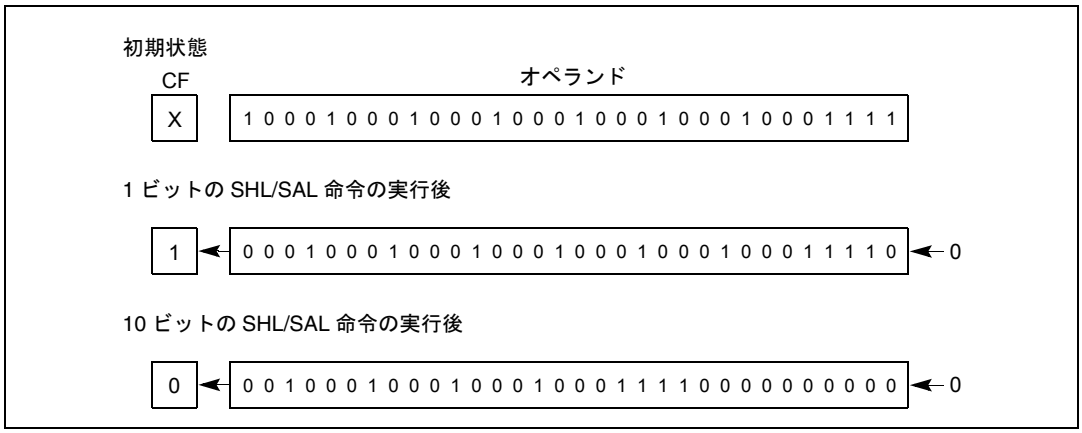


図 7-7. SHL/SAL 命令の動作

SHR 命令は、ソース・オペランドを 1 ～ 31 ビット位置だけ右にシフトする（図 7-8. を参照）。SHL/SAL 命令と同じように、空いたビット位置はクリアされ、オペランドの外にシフトされた最後のビットは CF フラグにロードされる。

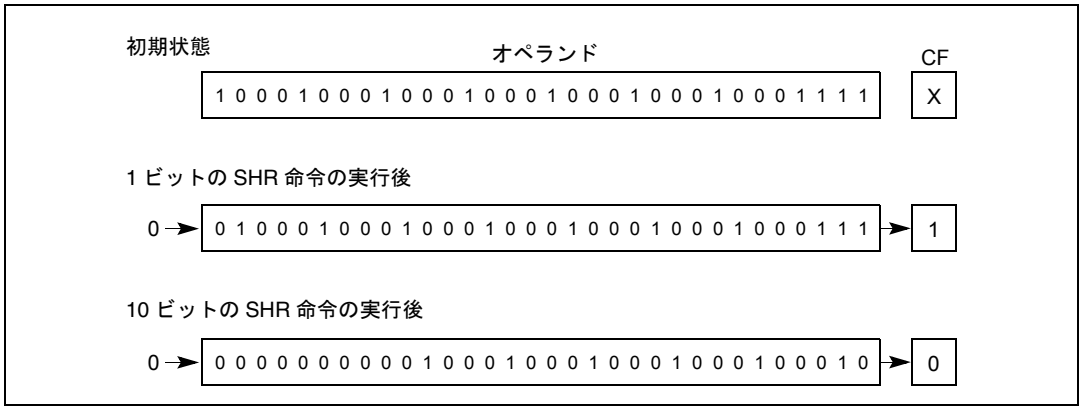


図 7-8. SHR 命令の動作

SAR 命令は、ソース・オペランドを 1 ～ 31 ビット位置だけ右にシフトする（図 7-9. を参照）。SHR 命令との相違点は、SAR 命令は、オペランドが正の場合は空いたビット位置をクリアし、オペランドが負の場合は空いたビットをセットすることによって、ソース・オペランドの符号を維持することである。この場合も、オペランドの外にシフトされた最後のビットが CF フラグにロードされる。

また、SAR 命令と SHR 命令を使用して、2 を累乗した値で除算を実行することができる（『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」の「SAL/SAR/SHL/SHR - Shift」を参照のこと）。

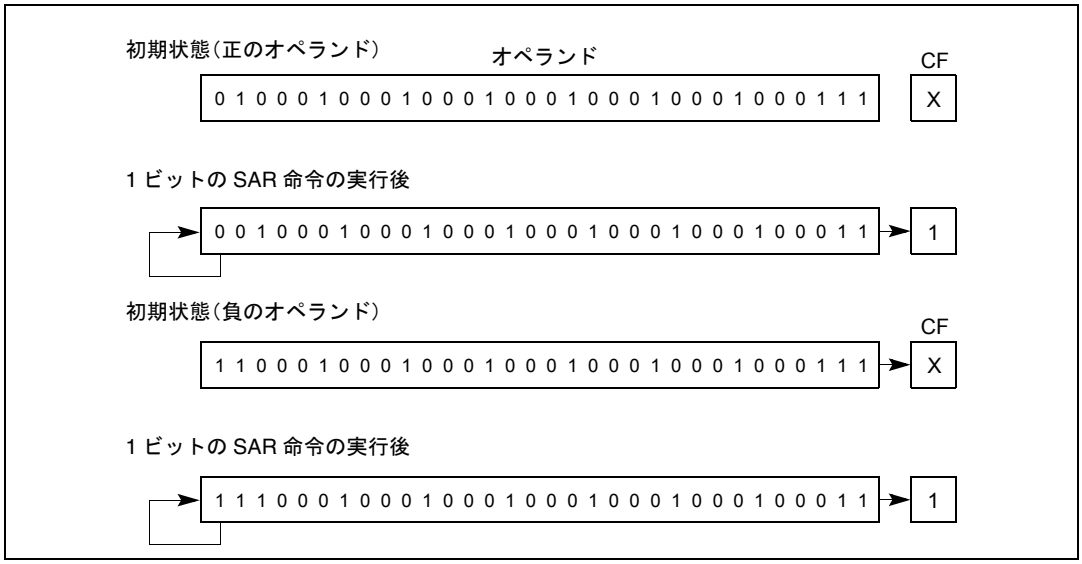


図 7-9. SAR 命令の動作

7.2.5.2. ダブルシフト命令

SHLD（shift left double）命令と SHRD（shift right double）命令は、1 つのオペランドからもう 1 つのオペランドへ指定したビット数をシフトする（図 7-10. を参照）。これらの命令は、アライメントの合っていないビット・ストリングの操作を容易にするために用意されている。これらの命令を使用して、各種のビット・ストリング転送操作も実行できる。

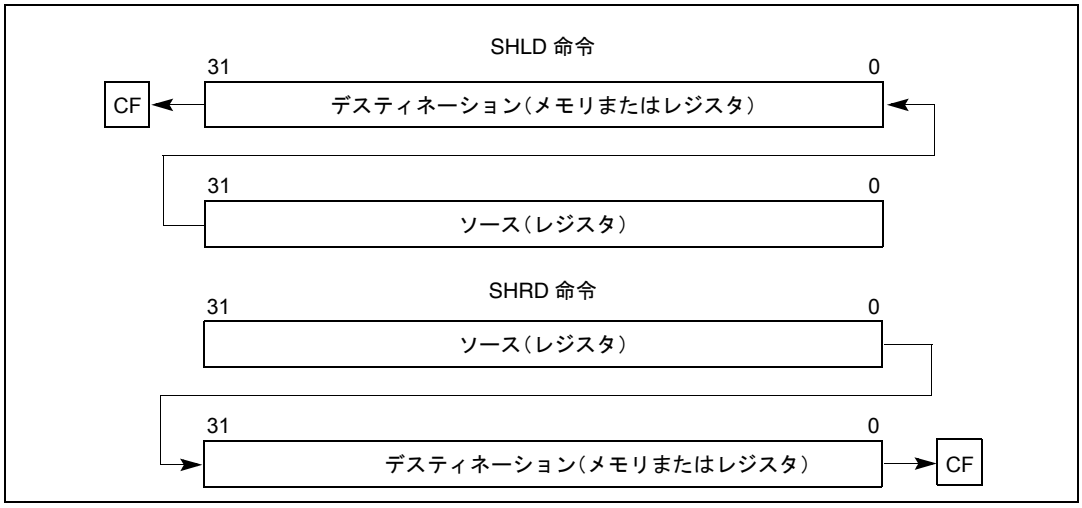


図 7-10. SHLD 命令と SHRD 命令の動作

SHLD 命令は、デスティネーション・オペランド内の各ビットを左にシフトし、（デスティネーション・オペランド内の）空いたビット位置を、ソース・オペランドからシフトされたビットで埋める。デスティネーション・オペランドとソース・オペランドは、同じ長さ（ワードまたはダブルワード）でなければならない。シフトするビット数の範囲は、0 ～ 31 である。このシフト操作の結果はデスティネーション・オペランドに格納され、ソース・オペランドは変更されない。デスティネーション・オペランドの外にシフトされた最後のビットは、CF フラグにロードされる。

SHRD 命令の動作は SHLD 命令と同じであるが、デスティネーション・オペランド内のビットが右にシフトされる点が異なる。空いたビット位置は、ソース・オペランドからシフトされたビットで埋められる。

7.2.5.3. ローテート命令

ROL（rotate left）、ROR（rotate right）、RCL（rotate through carry left）、RCR（rotate through carry right）命令は、デスティネーション・オペランド内の各ビットを、一方の端からもう一方の端に循環させる（図 7-11. を参照）。シフト命令とは異なり、ローテート命令ではビットは失われない。循環させるビット数の範囲は、0 ～ 31 である。

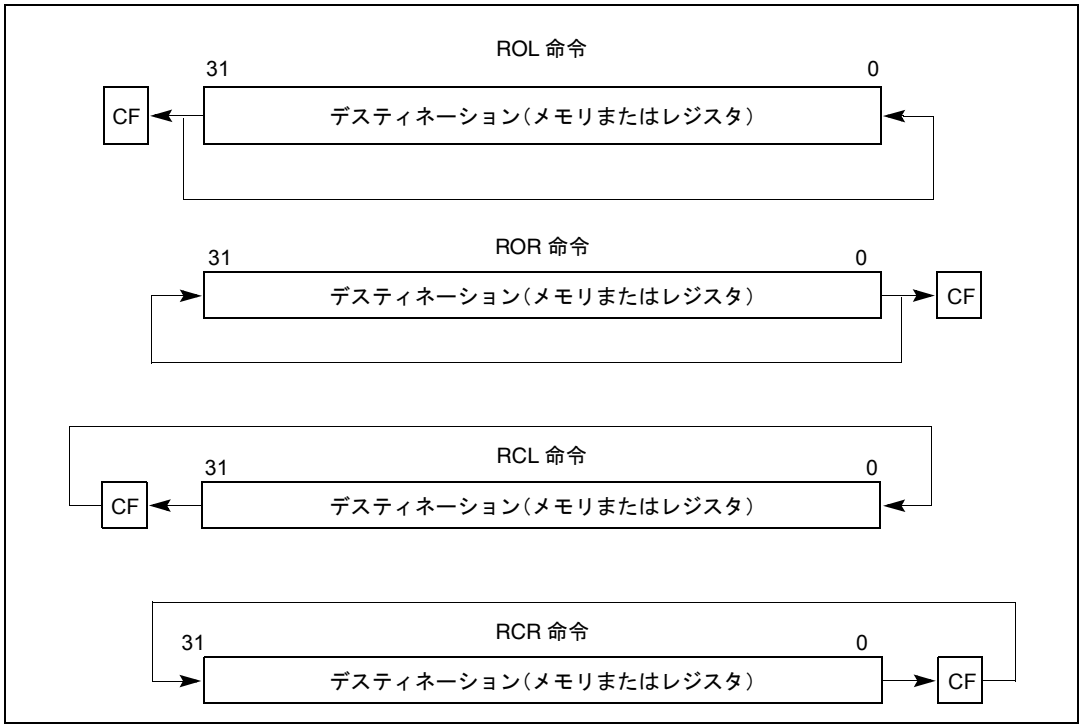


図 7-11. ROL、ROR、RCL、および RCR 命令の動作

ROL 命令は、オペランド内の各ビットを左に（最上位ビットの方向に）循環させる。ROR 命令は、オペランドを右に（最下位ビットの方向に）循環させる。

RCL 命令は、オペランド内の各ビットを、CF フラグを通して左に循環させる。この命令は、オペランドの上位側を 1 ビット拡張したものとして、CF フラグを扱う。オペランドの最上位ビットの位置から押し出された各ビットは、CF フラグ内に入る。同時に、CF フラグ内にあったビットは、オペランドの最下位ビットの位置に入る。

RCR 命令は、オペランド内の各ビットを、CF フラグを通して右に循環させる。

すべてのローテート命令で、CF フラグは、常に（その命令が CF フラグをオペランドの拡張として使用しない場合でも）、オペランドから押し出された最後のビットの値を格納する。このフラグの値は、条件付きジャンプ命令（JC または JNC）によってテストできる。

7.2.6. ビット命令とバイト命令

ビット命令とバイト命令は、ビット・ストリングまたはバイト・ストリングを操作する。説明のため、これらの命令は次の下位のサブグループに分けられる。

- 単一ビットのテストおよび変更命令
- ビット・ストリングのスキャン命令
- 条件付きバイトセット命令
- オペランドのテストと結果の報告の命令

7.2.6.1. ビットテストおよび変更命令

ビットテストおよび変更命令（表 7-3. を参照）は、オペランド内の 1 つのビットを操作する。このビットの位置は、オペランドの最下位ビットからのオフセットで指定される。プロセッサは、テストされ変更されるビットを特定すると、最初にそのビットの現在値を CF フラグにロードする。次に、この命令の変更操作の指定にしたがって、選択されたビットに新しい値を割り当てる。

表 7-3. ビットテストおよび変更命令

命令	CF フラグに対する影響	選択されたビットに対する影響
BT（Bit Test）	CF フラグ←選択されたビット	影響なし
BTS（Bit Test and Set）	CF フラグ←選択されたビット	選択されたビット ← 1
BTR（Bit Test and Reset）	CF フラグ←選択されたビット	選択されたビット ← 0
BTC（Bit Test and Complement）	CF フラグ←選択されたビット	選択されたビット ← NOT（選択されたビット）

7.2.6.2. ビットスキャン命令

BSF（bit scan forward）命令と BSR（bit scan reverse）命令は、ソース・オペランド内のビット・ストリングをスキャンしてセットされたビットを探し、最初に見つかったセットされたビットのビット・インデックスをデスティネーション・レジスタに格納する。このビット・インデックスは、ビット・ストリング内の最下位ビット（ビット 0）から最初のセットされたビットまでのオフセットである。BSF 命令は、ソース・オペランドを下位から上位の方向に（ソース・オペランドのビット 0 から最上位ビットに向かって）スキャンする。BSR 命令は、上位から下位の方向に（最上位ビットから最下位ビットに向かって）スキャンする。

7.2.6.3. 条件付きバイトセット命令

SET*cc*（set byte on condition）命令は、EFLAGS レジスタ内の選択されたステータス・フラグ（CF、OF、SF、ZF、PF）の状態に基づいて、デスティネーション・オペランド・バイトを 0 または 1 にセットする。SET ニーモニックに付加されるサフィックス（*cc*）は、テストされる条件を指定する。

例えば、SETO 命令は、オーバーフローがないかどうかをテストする。OF フラグがセットされている場合は、デスティネーション・バイトは 1 にセットされる。OF がクリアされている場合は、デスティネーション・バイトは 0 にクリアされる。付録 B「EFLAGS 条件コード」に、この命令でテストできる条件の一覧を示す。

7.2.6.4. テスト命令

TEST 命令は、2 つのオペランドの論理積（AND）演算を実行し、その結果に基づいて SF、ZF、PF フラグをセットする。セットされたフラグは、条件付きジャンプ命令、条件付きループ命令、または SET*cc* 命令によってテストできる。AND 命令との相違点は、TEST 命令はいずれのオペランドも変更しないことである。

7.2.7. 制御転送命令

IA-32 プロセッサは、プログラムの実行フローを指示するために、条件付き制御転送命令と無条件制御転送命令を備えている。条件付き転送は、EFLAGS レジスタのステータス・フラグが指定した状態である場合にのみ実行される。無条件制御転送は、常に実行される。

説明のため、これらの命令は次の下位のサブグループに分けられる。

- 無条件転送命令
- 条件付き転送命令
- ソフトウェア割り込み命令

7.2.7.1. 無条件転送命令

JMP、CALL、RET、INT、IRET 命令は、プログラムの制御を、命令ストリーム内の他の位置（デスティネーション・アドレス）に転送する。デスティネーションは、同じコード・セグメント内（near 転送）であっても、異なるコード・セグメント内（far 転送）であってもかまわない。

ジャンプ命令。JMP（jump）命令は、無条件に、プログラムの制御をデスティネーション命令に転送する。これは一方向の転送であり、リターンアドレスは保存されない。

デスティネーション・オペランドは、デスティネーション命令のアドレス（命令ポインタ）を指定する。アドレスは、**相対アドレス**でも**絶対アドレス**でもよい。

相対アドレスは、EIP レジスタ内のアドレスを基準とするディスプレースメント（オフセット）である。デスティネーション・アドレス（near ポインタ）は、EIP レジスタ内のアドレスにこのディスプレースメントを加算することによって得られる。このディスプレースメントは、符号付き整数で指定されるため、命令ストリーム内で順方向にジャンプすることも、逆方向にジャンプすることもできる。

絶対アドレスは、セグメントのアドレス 0 からのオフセットである。このアドレスは、次のいずれかの方法で指定される。

- **汎用レジスタ内のアドレス。**このアドレスは near ポインタとして扱われ、EIP レジスタにコピーされる。プログラムの実行は、現在のコード・セグメント内の新しいアドレスから再開される。
- **プロセッサの標準アドレス指定モードで指定されたアドレス。**このアドレスは、near ポインタまたは far ポインタである。アドレスが near ポインタの場合は、アドレスはオフセットに変換され、EIP レジスタにコピーされる。アドレスが far ポインタの場合は、アドレスはセグメント・セレクタとオフセットに変換される。セグメント・セレクタ部は CS レジスタにコピーされ、オフセット部は EIP レジスタにコピーされる。

プロテクト・モードでは、JMP 命令によって、コールゲート、タスクゲート、タスクステートの各セグメントへのジャンプも可能である。

コール命令とリターン命令。CALL（call procedure）命令は、あるプロシージャ（またはサブルーチン）から他のプロシージャへのジャンプを実行する。RET（return from procedure）命令は、呼び出し元プロシージャに戻るジャンプ（リターン）を実行する。

CALL 命令は、現在のプロシージャ（呼び出し元プロシージャ）から他のプロシージャ（呼び出し先プロシージャ）に、プログラムの制御を転送する。呼び出し元プロシージャに戻れるように、CALL 命令は、呼び出し先プロシージャにジャンプする前に、EIP レジスタの現在の内容をスタック上に保存する。プログラムの制御を転送する前に、EIP レジスタに、CALL 命令に続く命令のアドレスが格納される。このアドレスは、スタック上にプッシュされると、**リターン命令ポインタ**または**リターンアドレス**と呼ばれる。

呼び出し先プロシージャのアドレス（ジャンプ先のプロシージャ内の最初の命令のアドレス）は、JMP 命令の場合と同じ方法で、CALL 命令内で指定される（7-18 ページの「ジャンプ命令」を参照）。このアドレスは、相対アドレスで指定することも、絶対アドレスで指定することもできる。絶対アドレスで指定する場合は、near ポインタでも far ポインタでもよい。

RET 命令は、現在実行中のプロシージャ（呼び出し先プロシージャ）から、それを呼び出したプロシージャ（呼び出し元プロシージャ）に、プログラムの制御を戻す。プログラムの制御を戻すために、リターン命令ポインタがスタックから EIP レジスタにコピーされる。プログラムの実行は、EIP レジスタによって指定される命令から再開される。

RET 命令にはオプションのオペランドが 1 つある。リターン動作の際に、このオペランドの値が ESP レジスタの内容に加算される。このオペランドによってスタックポインタをインクリメントし、呼び出し元プロシージャがスタック上にプッシュしたパラメータを、スタックから削除することができる。

CALL 命令と RET 命令を使用してプロシージャ・コールを実行する機構については、6.3. 節「CALL と RET によるプロシージャのコール」を参照のこと。

割り込みからのリターン命令。プロセッサは、割り込みを処理するとき、割り込み処理プロシージャに対する暗黙的なコールを実行する。IRET（return from interrupt）命令は、プログラムの制御を、割り込みハンドラから、割り込みをかけられたプロシージャ（すなわり、割り込みの発生時に実行されていたプロシージャ）に戻す。IRET 命令は、RET 命令と同様の操作を実行するが、スタックから EFLAGS レジスタもリストアする点が異なる（7-19 ページの「コール命令とリターン命令」を参照）。プロセッサが割り込みを処理するとき、EFLAGS レジスタの内容は、リターン命令ポインタと一緒に、スタック上に自動的に格納される。

7.2.7.2. 条件付き転送命令

条件付き転送命令は、指定された条件が満たされる場合に、命令ストリーム内の他の命令にプログラムの制御を転送するジャンプまたはループを実行する。制御転送の条件は、EFLAGS レジスタ内のステータス・フラグ（CF、ZF、OF、PF、SF）の各種の状態を定義する、一連の条件コードによって指定される。

条件付きジャンプ命令。J*cc*（conditional jump）命令は、命令の条件コード（*cc*）で指定された条件が満たされる場合に、プログラムの制御をデスティネーション命令に転送する（表 7-4. を参照）。この条件が満たされない場合は、J*cc* 命令の次の命令からプログラムの実行が再開される。JMP 命令の場合と同じように、これは一方向の転送であり、リターンアドレスは保存されない。

表 7-4. 条件付きジャンプ命令

命令ニーモニック	条件（フラグの状態）	説明
符号なし条件付きジャンプ		
JA/JNBE	(CF or ZF)=0	より大きい / より小さくなく等しくない
JAE/JNB	CF=0	より大きいか等しい / より小さくない
JB/JNAE	CF=1	より小さい / より大きくなく等しくない
JBE/JNA	(CF or ZF)=1	より小さいか等しい / より大きくない
JC	CF=1	キャリー
JE/JZ	ZF=1	等しい / ゼロ
JNC	CF=0	キャリーなし
JNE/JNZ	ZF=0	等しくない / ゼロでない
JNP/JPO	PF=0	パリティなし / 奇数パリティ
JP/JPE	PF=1	パリティ / 偶数パリティ
JCXZ	CX=0	レジスタ CX がゼロ
JECXZ	ECX=0	レジスタ ECX がゼロ
符号付き条件付きジャンプ		
JG/JNLE	((SF xor OF) or ZF) =0	より大きい / より小さくなく等しくない
JGE/JNL	(SF xor OF)=0	より大きいか等しい / より小さくない
JL/JNGE	(SF xor OF)=1	より小さい / より大きくなく等しくない
JLE/JNG	((SF xor OF) or ZF)=1	より小さいか等しい / より大きくない
JNO	OF=0	オーバーフローなし
JNS	SF=0	符号なし（負でない）
JO	OF=1	オーバーフロー
JS	SF=1	符号（負）

デスティネーション・オペランドは、現在のコード・セグメント内の命令を指す相対アドレス（EIP レジスタ内のアドレスを基準とする符号付きオフセット）を指定する。J*cc* 命令は far 転送をサポートしないが、J*cc* 命令と JMP 命令を組み合わせれば、far 転送を実行できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「Jcc - Jump if Condition Is Met」を参照）。

表 7-4. は、J*cc* 命令のニーモニックと、各命令でテストされる条件を示している。J*cc* 命令のニーモニックは、"J" に条件コード・ニーモニックを付加したものである。

J*cc* 命令は、符号なし条件付きジャンプと符号付き条件付きジャンプの 2 つのグループに分けられる。符号なし条件付きジャンプ命令は、符号なし整数に対して実行された演算の結果を条件とする。符号付き条件付きジャンプ命令は、符号付き整数に対して実行された演算の結果を条件とする。ペアで示した命令（例えば、JA/JNBE）は、同

じ命令の別名である。アセンブラは、プログラム・リストが読みやすくなるように、これらの別名を用意している。

JCXZ 命令は、1 つ以上のステータス・フラグの代わりに CX レジスタをテストする。JECXZ 命令は、1 つ以上のステータス・フラグの代わりに ECX レジスタをテストする。これらの命令についての詳細は、7-22 ページの「Jump If Zero 命令」を参照のこと。

ループ命令。LOOP、LOOPE（loop while equal）、LOOPZ（loop while zero）、LOOPNE（loop while not equal）、および LOOPNZ（loop while not zero）命令は、ECX レジスタの値をループの実行回数カウンタとして使用する、条件付きジャンプ命令である。すべてのループ命令は、実行されるたびに ECX レジスタの値をデクリメントし、ゼロに到達したときにループを終了する。LOOPE、LOOPZ、LOOPNE、LOOPNZ 命令は、ZF フラグの状態によって、カウンタがゼロに到達する前にループを終了する。

LOOP 命令は、ECX レジスタ（アドレスサイズ属性が 16 の場合は CX レジスタ）の内容をデクリメントし、ECX レジスタがループ終了条件を満たすかどうかテストする。ECX レジスタ内のカウンタがゼロでない場合は、プログラムの制御は、デスティネーション・オペランドによって指定される命令アドレスに転送される。デスティネーション・オペランドは相対アドレス（すなわち、EIP レジスタの内容を基準とするオフセット）であり、一般的に、ループ内で実行されるコードブロック内の最初の命令を指す。ECX レジスタ内のカウンタがゼロに到達すると、プログラムの制御は LOOP 命令の直後の命令に戻され、その命令がループを終了する。LOOP 命令を初めて実行するとき、ECX レジスタ内のカウンタがゼロになっていると、レジスタは FFFFFFFFH にデクリメントされるため、ループは 2^{32} 回実行される。

LOOPE 命令と LOOPZ 命令は、同じ操作を実行する（これらは同じ命令のニーモニックである）。これらの命令は、LOOP 命令と同じように動作するが、ZF フラグもテストする点が異なる。

ECX レジスタ内のカウンタがゼロでなく、ZF フラグがセットされている場合は、プログラムの制御はデスティネーション・オペランドに転送される。カウンタがゼロに到達するか、ZF フラグがクリアされると、プログラムの制御は LOOPE/LOOPZ 命令の直後の命令に戻され、ループは終了する。

LOOPNE 命令と LOOPNZ 命令（同じ命令のニーモニック）は、LOOPE/LOOPZ 命令と同じように動作するが、ZF フラグがセットされた場合にループを終了する点が異なる。

Jump If Zero 命令。JECXZ（jump if ECX zero）命令は、ECX レジスタの値がゼロの場合に、デスティネーション・オペランドで指定された位置にジャンプする。この命令とループ命令（LOOP、LOOPE、LOOPZ、LOOPNE、または LOOPNZ）を組み合わせれば、ループを開始する前に ECX レジスタをテストすることができる。ループ命令は、ECX レジスタの内容をデクリメントしてから、ECX レジスタがゼロかどうかをテ

ストする。7-22 ページの「ループ命令」を参照のこと。したがって、ECX レジスタの値が最初からゼロになっている場合は、最初のループ命令でカウンタが FFFFFFFFH にデクリメントされるため、ループが 2^{32} 回実行されてしまう。この問題を防ぐために、JECXZ 命令をループのコードブロックの始まりに挿入しておき、ECX レジスタの初期値がゼロの場合に、ループの外にジャンプさせることができる。反復されるストリング・スキャン命令および比較命令と合わせて使用した場合、JECXZ 命令は、カウンタがゼロに到達したためにループが終了したのか、それともスキャン条件または比較条件が満たされたためにループが終了したのかを判断できる。

JCXZ（jump if CX is zero）命令は、16 ビットのアドレスサイズ属性の使用時に、JECXZ 命令と同じように動作する。この命令は、CX レジスタの値がゼロかどうかをテストする。

7.2.7.3. ソフトウェア割り込み命令

INT *n*（software interrupt）、INTO（interrupt on overflow）、BOUND（detect value out of range）命令によって、プログラムは、指定された割り込みまたは例外を直接に発生させ、その割り込みまたは例外用のハンドラルーチンを呼び出すことができる。

INT *n* 命令は、命令内にベクタ番号または割り込み/例外をコード化することによって、IA-32 プロセッサのすべての割り込みまたは例外を生成することができる。この命令を使用して、ソフトウェア生成割り込みをサポートしたり、割り込み / 例外ハンドラの動作をテストすることができる。

IRET（return from interrupt）命令は、プログラムの制御を、割り込みハンドラから、割り込みをかけられたプロシージャに戻す。IRET 命令は、RET 命令と同様の操作を実行するが、スタックから EFLAGS レジスタもリストアする点が異なる。

CALL（call procedure）命令は、あるプロシージャから他のプロシージャへのジャンプを実行する。RET（return from procedure）命令は、呼び出し元プロシージャに戻るジャンプを実行する。プロセッサが割り込みを処理するとき、EFLAGS レジスタの内容は、リターン命令ポインタと一緒に、スタック上に自動的に格納される。

INTO 命令は、OF フラグがセットされている場合に、オーバーフロー例外を発生させる。OF フラグがクリアされている場合は、例外を生成せずに実行を続ける。この命令によって、ソフトウェアは、オーバーフロー例外ハンドラに直接アクセスして、オーバーフロー条件が発生していないかどうかをチェックできる。

BOUND 命令は、符号付きの値を許容範囲の上限および下限と比較し、値が下限より小さいか、上限より大きい場合は、「BOUND 範囲超過」例外を生成する。この命令は、配列インデックスが、その配列について定義された有効範囲内に入るかどうかを確認するような操作に便利である。

7.2.8. ストリングの操作

MOVS（Move String）、CMPS（Compare string）、SCAS（Scan string）、LODS（Load string）、STOS（Store string）命令は、英数字文字列などの大きなデータ構造を、メモリ内で転送し、チェックすることができる。これらの命令は、ストリング内の個々の要素（バイト、ワード、またはダブルワード）を操作する。操作対象となるストリング要素は、ESI（ソース・ストリング要素）レジスタとEDI（デスティネーション・ストリング要素）レジスタで指定される。これらのレジスタは、ストリング要素を指す絶対アドレス（セグメント内のオフセット）を格納する。

デフォルトでは、ESIレジスタは、DSセグメント・セレクタで指定されるセグメントをアドレス指定する。セグメント・オーバライド・プリフィックスによって、ESIレジスタを、CS、SS、ES、FS、またはGSセグメント・レジスタに関連付けることができる。EDIレジスタは、ESセグメント・レジスタで指定されるセグメントをアドレス指定する。EDIレジスタにセグメント・オーバライドを適用することはできない。ストリング命令内で2つの異なるセグメント・レジスタを使用すると、異なるセグメント内のストリングに対して操作を実行できる。また、ESIレジスタをESセグメント・レジスタに関連付ければ、ソース・ストリングとデスティネーション・ストリングを同じセグメント内に置くことができる（同じセグメント・セレクタを使用してDSセグメント・レジスタとESセグメント・レジスタをロードし、ESIレジスタをデフォルトのままDSレジスタに関連付けても、この状態を実現できる）。

MOVS命令は、ESIレジスタによってアドレス指定されるストリング要素を、EDIレジスタによってアドレス指定される位置に転送する。アセンブラは、この命令の3つの「短縮形式」として、MOVSB（move byte string）、MOVSW（move word string）、MOVSD（move doubleword string）を認識する。これらの命令は、転送するストリングのサイズを指定する。

CMPS命令は、ソース・ストリング要素からデスティネーション・ストリング要素を引き、その結果に基づいて、EFLAGSレジスタ内のステータス・フラグ（CF、ZF、OF、SF、PF、AF）を更新する。どちらのストリング要素も、メモリに書き戻されない。アセンブラは、CMPS命令の3つの「短縮形式」として、CMPSB（compare byte strings）、CMPSW（compare word strings）、CMPSD（compare doubleword strings）を認識する。

SCAS命令は、EAX、AX、またはALレジスタ（オペランドの長さによる）の内容からデスティネーション・ストリング要素を引き、その結果に基づいてステータス・フラグを更新する。ストリング要素とレジスタの内容は変更されない。SCASB（scan byte string）、SCASW（scan word string）、SCASD（scan doubleword string）は、SCAS命令の「短縮形式」であり、オペランドの長さを指定する。

LODS命令は、ESIレジスタによって指定されるソース・ストリング要素を、EAXレジスタ（ダブルワード・ストリングの場合）、AXレジスタ（ワード・ストリングの場

合）、または AL レジスタ（バイト・ストリングの場合）にロードする。この命令の「短縮形式」は、LODSB（load byte string）、LODSW（load word string）、LODSD（load doubleword string）である。この命令は、通常はループ内で使用される。ストリング要素がターゲット・レジスタにロードされた後、他の命令がストリングの各要素を処理する。

STOS 命令は、EAX（ダブルワード・ストリング）、AX（ワード・ストリング）、または AL（バイト・ストリング）レジスタから、EDI レジスタで指定されるメモリ・ロケーションに、ソース・ストリング要素をストアする。この命令の「短縮形式」は、STOSB（store byte string）、STOSW（store word string）、STOSD（store doubleword string）である。この命令も、通常はループ内で使用される。ストリングは、通常は LODS 命令によってレジスタにロードされ、他の命令によって操作された後、STOS 命令によって再びメモリにストアされる。

I/O 命令（7.2.9. 項「I/O 命令」を参照）も、メモリ内のストリングを操作する。

7.2.8.1. ストリング操作の反復

7.2.8. 項「ストリングの操作」で説明したストリング命令は、ストリング操作を 1 回だけ実行する。ダブルワードより長いストリングを操作するには、ストリング命令とリピート・プリフィックス（REP）を組み合わせて反復命令を作成するか、ストリング命令をループ内に置けばよい。

ESI レジスタと EDI レジスタは、ストリング命令で使用される場合、命令が反復されるたびに自動的にインクリメントまたはデクリメントされ、ストリング内の次の要素（バイト、ワード、またはダブルワード）を指す。この方法で、ストリング操作は、上位のアドレスから下位のアドレスに向かって処理を進めることも、下位のアドレスから上位のアドレスに向かって処理を進めることもできる。EFLAGS レジスタ内の DF フラグは、レジスタがインクリメントされるか（DF=0）、デクリメントされるか（DF=1）を制御する。STD 命令は、このフラグをセットする。CLD 命令は、このフラグをクリアする。

以下のリピート・プリフィックスと ECX レジスタ内のカウンタを組み合わせて、ストリング命令を反復できる。

- REP - ECX レジスタがゼロでない間、命令を反復する。
- REPE/REPZ - ECX レジスタがゼロでなく、ZF フラグがセットされている間、命令を反復する。
- REPNE/REPNZ - ECX レジスタがゼロでなく、ZF フラグがクリアされている間、命令を反復する。

ストリング命令にリピート・プリフィックスがある場合は、プリフィックスによって指定された終了条件のうち1つが満たされるまで、処理が続行される。REPE/REPZ プリフィックスと REPNE/REPNZ プリフィックスは、CMPS 命令と SCAS 命令にのみ使用される。また、大きなメモリブロックを初期化する最も早い方法は、REP STOS 命令を実行することである。

7.2.9. I/O 命令

IN（input from port to register）、INS（input from port to string）、OUT（output from register to port）、OUTS（output string to port）命令は、プロセッサの I/O ポートとレジスタまたはメモリの間でデータを転送する。

レジスタ I/O 命令（IN と OUT）は、I/O ポートと、EAX レジスタ（32 ビット I/O）、AX レジスタ（16 ビット I/O）、または AL レジスタ（8 ビット I/O）の間でデータを転送する。読み取り元または書き込み先の I/O ポートは、即値オペランドまたは DX レジスタ内のアドレスで指定される。

ブロック I/O 命令（INS と OUTS）は、I/O ポートとメモリの間で、データブロック（ストリング）を転送する。これらの命令は、ストリング命令と同様の動作をする（7.2.8. 項「ストリングの操作」を参照）。メモリ内のストリング要素は、ESI レジスタと EDI レジスタによって指定される。また、リピート・プリフィックス（REP）を使用して、ブロック転送を実行する命令を反復することができる。INS 命令と OUTS 命令について、アセンブラは、INSB（input byte）、INSW（input word）、および INSD（input doubleword）と、OUTB（output byte）、OUTW（output word）、OUTD（output doubleword）の各ニーモニックを認識する。

INS 命令と OUTS 命令は、DX レジスタ内のアドレスを使用して、読み取り元または書き込み先の I/O ポートを指定する。

7.2.10. ENTER 命令と LEAVE 命令

ENTER 命令と LEAVE 命令は、C および Pascal などのブロック構造言語内のプロシージャ・コールのためのマシン語をサポートする。これらの命令と、これらの命令がサポートするコールおよびリターン機構については、6.5. 節「ブロック構造言語でのプロシージャ・コール」を参照のこと。

7.2.11. フラグ制御（EFLAGS）命令

フラグ制御（EFLAGS）命令によって、EFLAGS レジスタ内の選択したフラグの状態の読み取りや変更が行える。説明のため、これらの命令は次の下位のサブグループに分けられる。

- キャリーフラグおよび方向フラグ命令
- EFLAGS 転送命令
- 割り込みフラグ命令

7.2.11.1. キャリーフラグおよび方向フラグ命令

STC（set carry flag）、CLC（clear carry flag）、CMC（complement carry flag）命令は、EFLAGS レジスタ内の CF フラグを直接変更できる。これらの命令は、通常は、CF フラグを使用する命令を実行する前に、CF フラグを確認済みの状態に初期化するために使用される。これらの命令は、キャリー付きローテート命令（RCL および RCR）と組み合わせられる。

STD（set direction flag）命令と CLD（clear direction flag）命令は、EFLAGS レジスタ内の DF フラグを直接変更できる。DF フラグは、ストリング処理命令の実行時に、インデックス・レジスタ ESI と EDI をインクリメントするか、デクリメントするかを指定する。DF フラグがクリアされている場合は、ストリング命令を 1 回実行するたびに、インデックス・レジスタはインクリメントされる。DF フラグがセットされている場合は、レジスタはデクリメントされる。

7.2.11.2. EFLAGS 転送命令

EFLAGS 転送命令は、EFLAGS レジスタ内のフラグのグループを、レジスタまたはメモリにコピーしたり、レジスタまたはメモリからロードできる。

LAHF（load AH from flags）命令と SAHF（store AH into flags）命令は、5 つの EFLAGS ステータス・フラグ（SF、ZF、AF、PF、CF）を操作する。LAHF 命令は、これらのステータス・フラグを、それぞれ AH レジスタのビット 7、6、4、2、0 にコピーする。AH レジスタのその他のビット（ビット 5、3、1）の内容は未定義であり、EFLAGS レジスタの内容は変更されない。SAHF 命令は、AH レジスタのビット 7、6、4、2、0 を、それぞれ EFLAGS レジスタの SF、ZF、AF、PF、CF フラグにコピーする。

PUSHF（push flags）、PUSHFD（push flags double）、POPF（pop flags）、POPFD（pop flags double）命令は、EFLAGS レジスタ内のフラグを、スタックとの間でコピーする。PUSHF 命令は、EFLAGS レジスタの下位ワードをスタック上にプッシュする（図 7-12. を参照）。PUSHFD 命令は、EFLAGS レジスタ全体をスタック上にプッシュする（RF フラグと VM フラグはクリアされているものとして読み取られる）。

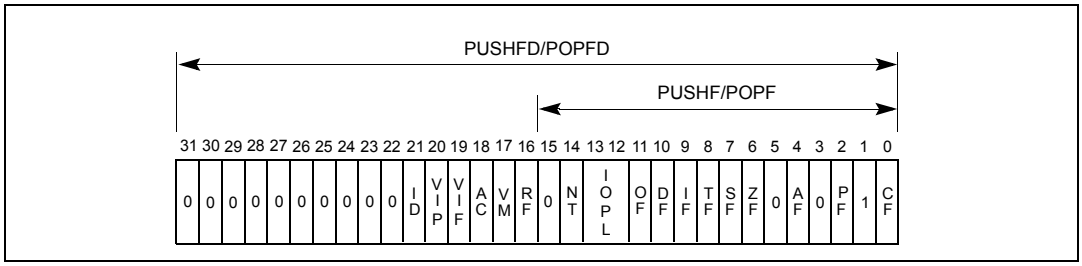


図 7-12. PUSHF、POPF、PUSHFD、POPFD 命令の影響を受けるフラグ

POPF 命令は、スタックから 1 ワードを EFLAGS レジスタにポップする。この命令の影響を受けるビットは、EFLAGS レジスタのビット 11、10、8、7、6、4、2、0 だけである。ただし、現在のコード・セグメントの現行特権レベル（CPL）が 0（最高の特権レベル）の場合は、IOPL ビット（ビット 13 とビット 12）も影響を受ける。I/O 特権レベルの数字が CPL より大きいか、CPL に等しい場合は、IF フラグ（ビット 9）も影響を受ける。

POPFD 命令は、1 ダブルワードを EFLAGS レジスタにポップする。この命令は、POPF 命令の影響を受けるビット以外に、AC ビット（ビット 18）と ID ビット（ビット 21）の状態を変更できる。POPF 命令について説明した、IOPL ビットと IF フラグの変更に関する制限は、POPFD 命令にも適用される。

7.2.11.3. 割り込みフラグ命令

STI（set interrupt flag）命令と CTI（clear interrupt flag）命令は、EFLAGS レジスタ内の割り込み IF フラグを直接変更できる。IF フラグは、ハードウェア生成割り込み（プロセッサの INTR ピンで受信される割り込み）の処理を制御する。IF フラグがセットされている場合は、プロセッサはハードウェア割り込みを処理する。IF フラグがクリアされている場合は、ハードウェア割り込みはマスクされる。

これらの命令を実行できるかどうかは、プロセッサの動作モードと、これらの命令を実行しようとするプログラムまたはタスクの現行特権レベル（CPL）によって決まる。

7.2.12. セグメント・レジスタ命令

IA-32 プロセッサは、プロセッサのセグメント・レジスタを直接アドレス指定する各種の命令を備えている。これらの命令は、オペレーティング・システムまたはエグゼクティブがセグメント化モードまたは実アドレスモードのメモリモデルを使用している場合にのみ使用される。

説明のため、これらの命令は次の下位のサブグループに分けられる。

- セグメント・レジスタ・ロードおよびストア命令

- far 制御転送命令
- ソフトウェア割り込み命令
- far ポインタロード命令

7.2.12.1. セグメント・レジスタ・ロードおよびストア命令

MOV 命令（7.2.1.1. 項「汎用データ転送命令」を参照）と PUSH 命令および POP 命令（7.2.1.3. 項「スタック操作命令」を参照）を使用して、セグメント・レジスタ（DS、ES、FS、GS、SS）との間で 16 ビット・セグメント・セレクタを転送できる。この転送は、常に、セグメント・レジスタと、汎用レジスタまたはメモリの間で行われる。セグメント・レジスタ同士の間の転送はサポートしていない。

POP 命令と MOV 命令は、CS レジスタに値を入れることはできない。far 制御転送を行う JMP、CALL、RET 命令（7.2.12.2. 項「far 制御転送命令」を参照）だけが、CS レジスタに直接影響を与えることができる。

7.2.12.2. far 制御転送命令

JMP 命令と CALL 命令（7.2.7. 項「制御転送命令」を参照）は、far ポインタをソース・オペランドとして受け入れて、CS レジスタによって現在指定されているセグメント以外のセグメントにプログラムの制御を転送できる。CALL 命令を使用して far コールを実行すると、EIP レジスタと CS レジスタの現在値がスタック上にプッシュされる。

RET 命令を使用して、far リターンを実行することができる（7-19 ページの「コール命令とリターン命令」を参照）。この場合、プログラムの制御は、呼び出し先プロシージャを格納しているコード・セグメントから、呼び出し元プロシージャを格納していたコード・セグメントに戻される。RET 命令は、呼び出し元プロシージャの CS レジスタと EIP レジスタの値を、スタックからリストアする。

7.2.12.3. ソフトウェア割り込み命令

ソフトウェア割り込み命令 INT、INTO、BOUND、IRET（7.2.7.3. 項「ソフトウェア割り込み命令」を参照）は、（現在のコード・セグメント以外のコード・セグメント内にある）割り込みプロシージャおよび例外ハンドラ・プロシージャのコールと、そこからのリターンを実行できる。ただし、これらの命令では、コード・セグメントの切り替えは、アプリケーション・プログラムから見て透過的に処理される。

7.2.12.4. far ポインタロード命令

far ポインタロード命令 LDS（load far pointer using DS）、LES（load far pointer using ES）、LFS（load far pointer using FS）、LGS（load far pointer using GS）、LSS（load far pointer using SS）は、メモリからセグメント・レジスタと汎用レジスタに far ポインタをロー

ドする。far ポインタのセグメント・セレクタ部は、選択されたセグメント・レジスタにロードされ、オフセットは、選択された汎用レジスタにロードされる。

7.2.13. その他の命令

以下の命令は、アプリケーション・プログラマが利用できる操作を実行する。

説明のため、これらの命令は次の下位のサブグループに分けられる。

- アドレス計算命令
- テーブル・ルックアップ命令
- プロセッサ識別命令
- ノー・オペレーション命令と未定義命令

7.2.13.1. アドレス計算命令

LEA（load effective address）命令は、ソース・オペランドのメモリ内での実効アドレス（セグメント内のオフセット）を計算し、その結果を汎用レジスタに入れる。この命令は、プロセッサの任意のアドレス指定モードを解釈でき、必要な任意のインデックス操作やスケール操作を実行できる。この命令は、ストリング命令を実行する前に ESI レジスタまたは EDI レジスタを初期化する場合や、XLAT 命令の前に EBX レジスタを初期化する場合に特に便利である。

7.2.13.2. テーブル・ルックアップ命令

XLAT および XLATB（table lookup）命令は、AL レジスタの内容を、メモリ内のトランスレーション・テーブルから読み取った 1 バイトで置き換える。AL レジスタの初期値は、トランスレーション・テーブルへの符号なしインデックスとして解釈される。このインデックスが、（トランスレーション・テーブルのベースアドレスを格納する）EBX レジスタの内容に加算されて、テーブル・エントリのアドレスが計算される。これらの命令は、特定のアルファベットから他のアルファベットに文字コードを変換するアプリケーションなどに使用される（例えば、ASCII コードを使用して、テーブル内でそれに相当する EBCDIC コードを参照できる）。

7.2.13.3. プロセッサ識別命令

CPUID（processor identification）命令は、この命令の実行対象となるプロセッサに関する情報を返す。

7.2.13.4. ノー・オペレーション命令と未定義命令

NOP（no operation）命令は、EIP レジスタをインクリメントして次の命令を指定するが、それ以外には何も影響を与えない。

UD2（undefined）命令は、無効オペコード例外を生成する。インテルでは、この機能のために、この命令のオペコードを予約している。この命令の目的は、ソフトウェア上で無効オペコード例外ハンドラをテストできるようにすることである。

8

x87 FPU による プログラミング

第 8 章
x87 FPU による プログラミング

8

x87 浮動小数点ユニット（FPU）は、画像処理、科学計算、工学計算、ビジネスなどのアプリケーション向けに、高性能の浮動小数点処理を可能にする。x87 FPU は、浮動小数点、整数、パックド BCD 整数の各データ型に対応し、2 進浮動小数点演算に関する IEEE 規格 754 に定義された浮動小数点処理アルゴリズムと例外処理アーキテクチャをサポートしている。

本章では、x87 FPU の実行環境と命令セットについて説明する。また、x87 FPU に固有の例外処理についても説明する。x87 FPU 命令と浮動小数点演算についての詳細は、以下の個所を参照のこと。

- x87 FPU 命令についての詳細は、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」に記載されている。
- x87 FPU の操作対象となる浮動小数点データ型、整数データ型、BCD データ型については、4.2.2. 項「浮動小数点データ型」、4.2.1.2. 項「符号付き整数」、4.7. 節「BCD およびパックド BCD 整数」に記載されている。
- x87 FPU が検出し、報告する浮動小数点例外の概要は、4.9. 節「浮動小数点例外の概要」、4.9.1. 項「浮動小数点例外条件」、4.9.2. 項「浮動小数点例外の優先順位」に記載されている。

8.1. x87 FPU の実行環境

x87 FPU は、IA-32 アーキテクチャ内の独立した実行環境である（図 8-1. を参照）。この実行環境は、8 つのデータレジスタ（x87 FPU データレジスタと呼ばれる）と、以下の汎用レジスタで構成される。

- ステータス・レジスタ
- コントロール・レジスタ
- タグ・ワード・レジスタ
- ラスト命令ポインタレジスタ
- ラスト・データ（オペランド）ポインタ・レジスタ
- オペコード・レジスタ

これらのレジスタについては、以下の各項で説明する。

x87 FPU は、プロセッサの通常の命令ストリーム内の命令を実行する。x87 FPU のステートは、（第 7 章で説明した）基本実行環境のステートや、（第 10 章、第 11 章、第 12 章で個別に説明する）SSE、SSE2、SSE3 のステートに依存しない。ただし、x87 FPU とインテル® MMX® テクノロジはステートを共有する。これは、MMX テクノロジ・レジスタは、x87 FPU データレジスタを別名で定義したものだからである。したがって、プログラマは、x87 FPU 命令と MMX 命令を使用するコードを作成する場合、x87 FPU のステートと MMX のステートを明確に管理する必要がある（9.5. 節「x87 FPUアーキテクチャとの互換性」を参照）。

8.1.1. x87 FPU データレジスタ

x87 FPU は、プロセッサの通常の命令ストリーム内の命令を実行する。x87 FPU のステートは、（第 7 章で説明した）基本実行環境のステートや、（第 10 章、第 11 章、第 12 章で個別に説明する）SSE、SSE2、SSE3 のステートに依存しない。ただし、x87 FPU とインテル® MMX® テクノロジはステートを共有する。これは、MMX テクノロジ・レジスタは、x87 FPU データレジスタを別名で定義したものだからである。したがって、プログラマは、x87 FPU 命令と MMX 命令を使用するコードを作成する場合、x87 FPU のステートと MMX のステートを明確に管理する必要がある（9.5. 節「x87 FPUアーキテクチャとの互換性」を参照）。

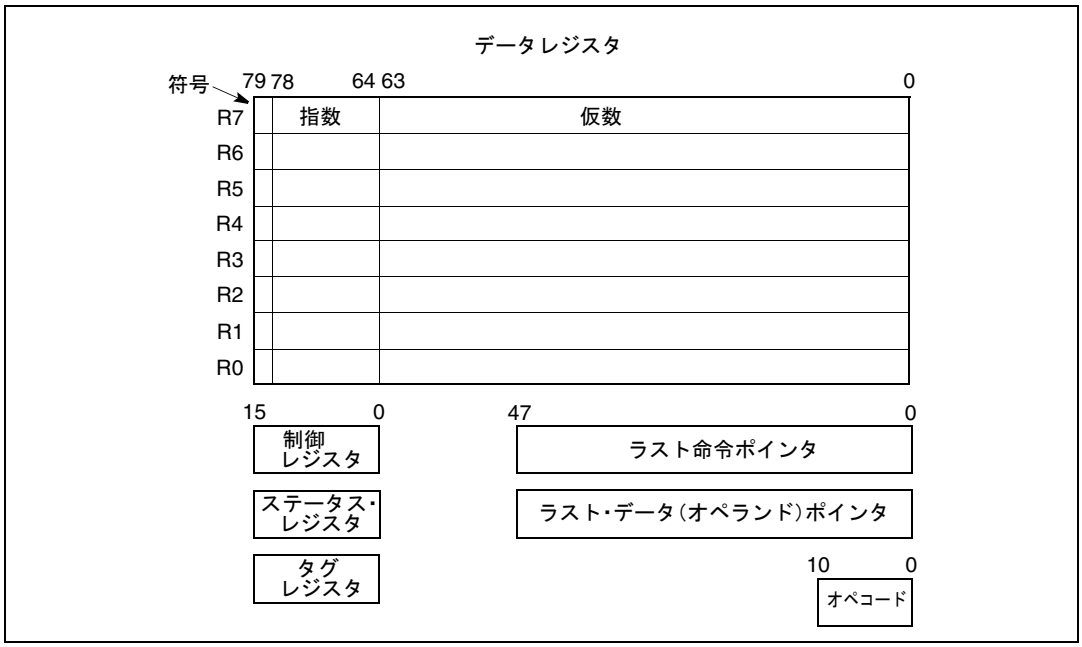


図 8-1. x87 FPU 実行環境

x87 FPU 命令は、8 つの x87 FPU データレジスタをレジスタスタックとして扱う（図 8-2. を参照）。データレジスタのアドレス指定はすべて、スタックのトップにあるレジスタに対して相対的になる。現在のスタックのトップにあるレジスタのレジスタ番号は、x87 FPU ステータス・ワード内の TOP（スタックのトップ）フィールド内に格納される。ロード操作では、TOP が 1 だけデクリメントされて、新しくスタックのトップになったレジスタに値がロードされる。また、ストア操作では、現在の TOP レジスタからメモリに値が格納され、その後で TOP が 1 だけインクリメントされる。（x87 FPU にとっては、ロード操作はプッシュに相当し、またストア操作はポップに相当する。）ただし、スタックのプッシュとポップを行わないロード操作とストア操作も利用可能である。

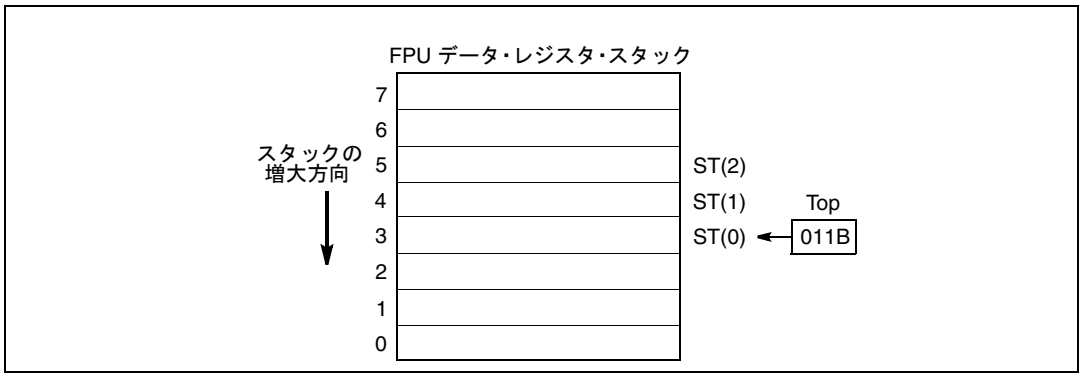


図 8-2. x87 FPU データ・レジスタ・スタック

TOP が 0 の場合にロード操作が実行されると、レジスタはラップアラウンドし、TOP の新しい値が 7 にセットされる。ラップアラウンドによってセーブされていない値が上書きされる可能性がある場合は、浮動小数点スタック・オーバーフロー例外によって示される（8.5.1.1. 項「スタック・オーバーフロー例外またはスタック・アンダーフロー例外（#IS）」を参照）。

浮動小数点命令の多くではいくつかのアドレス指定モードが用意されていて、プログラマはスタックのトップに対して暗黙的に操作するか、特定のレジスタに対しては TOP に相対させて明示的に操作することができる。アセンブラはこれらのレジスタアドレス指定モードをサポートしており、ST(0)（あるいは単に ST）という表現を使って現在のスタックのトップを表し、ST(i) という表現を使ってスタック内の TOP から *i* 番目（$0 \leqq i \leqq 7$）のレジスタを指定する。例えば、TOP に 011B が格納されている場合（スタックのトップがレジスタ 3）、次の命令は、スタック内の 2 つのレジスタ（レジスタ 3 と 5）の内容を加算する。

```
FADD ST, ST(2);
```

図 8-3. に、一連の計算を実行する場合に、x87 FPU レジスタのスタック構造や命令が一般的にどのように使用されるか、例を挙げて示す。この例では、2 次元のドット積が次のように計算される。

1. 最初の命令（FLD value1）が、スタック・レジスタ・ポインタ（TOP）をデクリメントし、値 5.6 をメモリから ST(0) にロードする。この操作の結果をスナップショット（a）に示す。
2. 2 番目の命令が、ST(0) 内の値をメモリからロードした値 2.4 で乗算し、その結果を ST(0) に格納する。この操作の結果をスナップショット（b）に示す。
3. 3 番目の命令が TOP をデクリメントし、値 3.8 を ST(0) にロードする。
4. 4 番目の命令が ST(0) 内の値をメモリからロードした値 10.3 で乗算し、その結果を ST(0) に格納する。この操作の結果をスナップショット（c）に示す。
5. 5 番目の命令が、この ST(0) の値と ST(1) の値を加算し、その結果を ST(0) に格納する。この操作の結果をスナップショット（d）に示す。

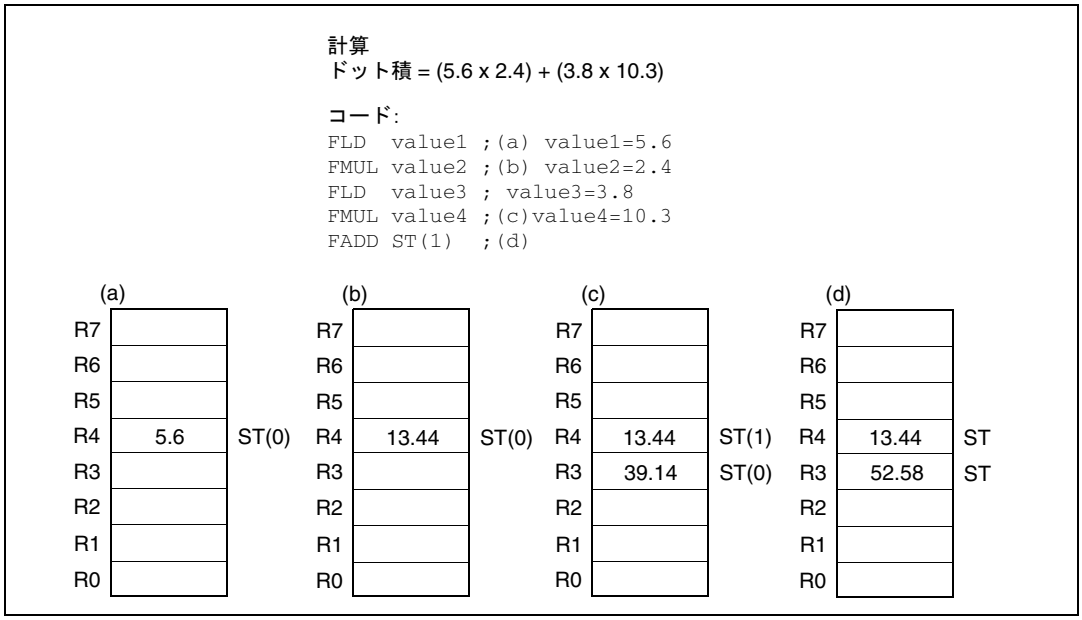


図 8-3. x87 FPU によるドット積の計算例

この例に示したプログラミング・スタイルは、浮動小数点命令セットによってサポートされる。スタック構造が計算上のボトルネックとなるような場合は、FXCH（Exchange x87 FPU register contents）命令を使用して計算を一本化できる。

8.1.1.1. x87 FPU レジスタスタックとのパラメータの受け渡し

汎用レジスタと同じように、x87 FPU データレジスタの内容もプロシージャ・コールの影響を受けない。すなわち、これらのレジスタの値はプロシージャの境界を越えて保持される。このため、コール元プロシージャは、x87 FPU データレジスタ（およびプロシージャ・スタック）を使用することで、プロシージャ間でパラメータを受け渡すことができる。コールされたプロシージャがレジスタスタックを介して渡されるパラメータを参照する場合は、現在のスタック・レジスタ・ポインタ（TOP）と、ST(0) ならびに ST(i) の表現を使用できる。さらに、一般的に行われている技法として、コール元のプロシージャやプログラムに実行を戻す際に、コールされたプロシージャがリターン値や結果をレジスタ ST(0) に残しておくこともできる。

プロシージャまたはコード・シーケンス内で MMX 命令と x87 FPU 命令を混在させる場合、プログラマは、x87 FPU データレジスタ内で渡されるパラメータの整合性を維持する責任を負う。x87 FPU データレジスタ内のパラメータが他のプロシージャに渡される前に MMX 命令が実行されると、それらのパラメータは失われる（9.5. 節「x87 FPU アーキテクチャとの互換性」を参照）。

8.1.2. x87 FPU ステータス・レジスタ

16 ビットの x87 FPU ステータス・レジスタ（図 8-4. を参照）は、x87 FPU の現在のステートを示す。x87 FPU ステータス・レジスタ内のフラグには、x87 FPU ビジーフラグ、スタック・トップ（TOP）ポインタ、条件コードフラグ、エラー・サマリ・ステータス・フラグ、スタック・フォルト・フラグ、例外フラグが含まれる。x87 FPU は、このレジスタ内の各フラグを設定することで、演算の結果を示す。

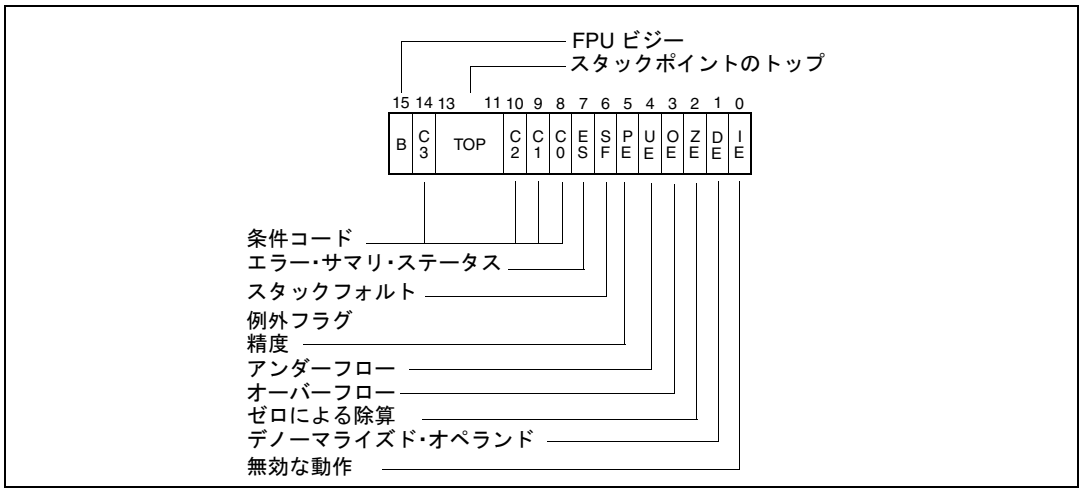


図 8-4. x87 FPU ステータス・ワード

x87 FPU ステータス・レジスタの内容（x87 FPU ステータス・ワードと呼ばれる）は、FSTSW/FNSTSW、FSTENV/FNSTENV、FSAVE/FNSAVE の各命令を使用してメモリに格納できる。また、FSTSW/FNSTSW 命令を使用して整数ユニットの AX レジスタに格納することもできる。

8.1.2.1. スタック・トップ（TOP）ポインタ

現時点で x87 FPU レジスタスタックのトップにある x87 FPU データレジスタを指すポインタは、x87 FPU ステータス・ワードのビット 11 ～ 13 に格納される。一般に TOP（スタックのトップを表す）と呼ばれるこのポインタは、0 ～ 7 の 2 進値である。TOP ポインタの詳細については、8.1.1. 項「x87 FPU データレジスタ」を参照のこと。

8.1.2.2. 条件コードフラグ

浮動小数点の比較演算や算術演算の結果は、4 つの条件コードフラグ（C0 ～ C3）に示される。表 8-1. に、浮動小数点命令が条件コードフラグを設定する方法をまとめて示す。これらの条件コードビットは、基本的には、条件付き分岐や例外処理で使用される情報を格納するのに使用される（8.1.3. 項「条件コードに基づく分岐と条件付き移動」を参照）。

表 8-1. に示すように、C1 条件コードフラグは各種の機能で使用される。x87 FPU ステータス・ワード内の IE フラグと SF フラグが共にセットされている（スタックのオーバーフロー例外またはアンダーフロー例外（# IS）を示す）場合は、C1 フラグでオーバーフロー（C1=1）かアンダーフロー（C1=0）のいずれであるかを識別する。ステータス・ワード内の PE フラグがセットされている（結果が丸められて不正確であることを示す）場合は、命令による最後の丸めが切り上げであった場合に C1 フラグが 1 にセットされる。C1 は、FXAM 命令によって、現在チェックされている値の符号に設定される。

C2 条件コードフラグは、FPREM 命令と FPREM1 命令が剰余計算の未完了（部分剰余）を示すために使用する。剰余計算が正常に完了している場合は、C0、C3、C1 の各条件コードフラグがそれぞれ、商の 3 つの最下位ビット（Q2、Q1、Q0）に対してセットされる。これらの命令が条件コードフラグを使用する方法については、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」にある「FPREM1 － Partial Remainder」を参照のこと。

FPTAN、FSIN、FCOS、FSINCOS の各命令は、ソース・オペランドが許容範囲である $\pm 2^{63}$ を超えたことを示す場合に C2 フラグを 1 にセットし、ソース・オペランドが許容範囲内の場合は、C2 フラグをクリアする。

表 8-1. で条件コードフラグのステートが「未定義」と記されている場合は、それらのフラグのどの特定値にも依存してはならない。

8.1.2.3. x87 FPU 浮動小数点例外フラグ

x87 FPU ステータス・ワードの 6 つの x87 FPU 浮動小数点例外フラグ（ビット 0 ～ 5）は、これらのビットが最後にクリアされてから 1 つ以上の浮動小数点例外が検出されたことを示す。個々の例外フラグ（IE、DE、ZE、OE、UE、PE）については、8.4. 節「x87 FPU 浮動小数点例外処理」で詳しく説明する。それぞれの例外フラグは、x87 FPU 制御ワードの例外マスクビットでマスクできる（8.1.4. 項「x87 FPU 制御ワード」を参照）。マスクされていない例外フラグのいずれかがセットされると、例外サマリ・ステータス（ES）フラグ（ビット 7）がセットされる。ES フラグがセットされると、8.7. 節「ソフトウェア内での x87 FPU 例外の処理」で説明する技法のいずれかを使用して x87 FPU 例外ハンドラが呼び出される。（例外フラグがマスクされている場合、そのフラグに関連付けられている例外が発生すると、x87 FPU は適切なフラグをセットするが ES フラグはセットしないので注意すること。）

例外フラグは、「スティッキー（頑固）」なビットである。すなわち、いったんセットされると、明示的にクリアされるまではセットされたままになる。例外フラグをクリアするには、FCLEX/FNCLEX（Clear exceptions）命令を実行する、FINIT/FNINIT 命令か FSAVE/FNSAVE 命令を使用して x87 FPU を再初期化する、FRSTOR 命令か FLDENV 命令を使用してフラグを上書きする、のいずれかの方法を用いる。

B ビット（ビット 15）は、8087 との互換性を得るためだけに含まれている。このビットは、ES フラグの内容を反映する。

表 8-1. 条件コードの解釈

命令	C0	C3	C2	C1
FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FTST, FUCOM, FUCOMP, FUCOMPP	比較の結果		オペランドが比較できない	0 または #IS
FCOMI, FCOMIP, FUCOMI, FUCOMIP	未定義。（これらの命令は EFLAGS レジのステータス・フラグをセットする。）			#IS
FXAM	オペランド・クラス			符号
FPREM, FPREM1	Q2	Q1	0= 余剰計算完了 1= 余剰計算未完了	Q0 または #IS
F2XM1, FADD, FADDP, FBSTP, FCMOVcc, FIADD, FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR, FIMUL, FIST, FISTP, FISUB, FISUBR,FMUL, FMULP, FPATAN, FRNDINT, FSCALE, FST, FSTP, FSUB, FSUBP, FSUBR, FSUBRP, FSQRT, FYL2X, FYL2XP1	未定義			切り上げまたは #IS

表 8-1. 条件コードの解釈（続き）

命令	C0	C3	C2	C1
FCOS, FSIN, FSINCOS, FPTAN	未定義		0= ソース・オペランドが範囲内 1= ソース・オペランドが範囲外	切り上げまたは #IS（C2=1 の場合は未定義）
FABS, FBLD, FCHS, FDECSTP, FILD, FINCSTP, FLD, Load Constants, FSTP (ext. real), FXCH, FXTRACT	未定義			0 または #IS
FLDENV, FRSTOR	メモリからロードされた各ビット			
FFREE, FLDCW, FCLEX/FNCLEX, FNOP, FSTCW/FNSTCW, FSTENV/FNSTENV, FSTSW/FNSTSW	未定義			
FINIT/FNINIT, FSAVE/FNSAVE	0	0	0	0

8.1.2.4. スタック・フォルト・フラグ

スタック・フォルト・フラグ（x87 FPU ステータス・ワードのビット 6）は、x87 FPU データ・レジスタ・スタック内のデータに、スタック・オーバーフローまたはスタック・アンダーフローが発生したことを示す。x87 FPU は、スタックのオーバーフロー条件またはアンダーフロー条件を検出した場合にこの SF フラグを明示的にセットするが、無効演算オペランド条件を検出した場合には、SF フラグを明示的にはクリアしない。このフラグがセットされている場合は、フォルトの性質は条件コードフラグ C1 が示す（すなわち、C1 =1 であればオーバーフロー、C1 = 0 であればアンダーフロー）。SF フラグは「スティッキー」なフラグであり、いったんセットされると、FINIT/FNINIT、FCLEX/FNCLEX、FSAVE/FNSAVE などの命令で明示的にクリアしない限り、プロセッサがこのフラグをクリアすることはない。

x87 FPU スタックフォルトの詳細については、8.1.6. 項「x87 FPU タグワード」を参照のこと。

8.1.3. 条件コードに基づく分岐と条件付き移動

P6 ファミリ・プロセッサ以降の x87 FPU では、2 つの浮動小数点値の比較結果を基づいて、分岐や条件付き移動を実行するためのメカニズムが 2 つ用意されている。本書では、これらのメカニズムを「旧メカニズム」と「新メカニズム」と呼ぶ。

旧メカニズムは、インテル® Pentium® Pro プロセッサより前の x87 FPU と P6 ファミリ・プロセッサで利用できる。このメカニズムは、浮動小数点比較命令（FCOM、FCOMP、FCOMPP、FTST、FUCOMPP、FICOM、および FICOMP）を使用して 2 つの浮動小数

点値を比較し、その結果にしたがって条件コードフラグ（C0 ～ C3）を設定する。次に、条件コードフラグの内容を、次に挙げる 2 ステップの処理（図 8-5. を参照）によって EFLAGS レジスタのステータス・フラグにコピーする。

1. FSTSW AX 命令で、x87 FPU ステータス・ワードを AX レジスタに移動する。
2. SAHF 命令で、AX レジスタの上位 8 ビット（条件コードフラグが含まれる）を EFLAGS レジスタの下位 8 ビットにコピーする。

条件コードフラグを EFLAGS レジスタにロードした後は、EFLAGS レジスタ内のステータス・フラグの新しい設定に基づいて、条件付きジャンプや条件付き移動が実行できる。

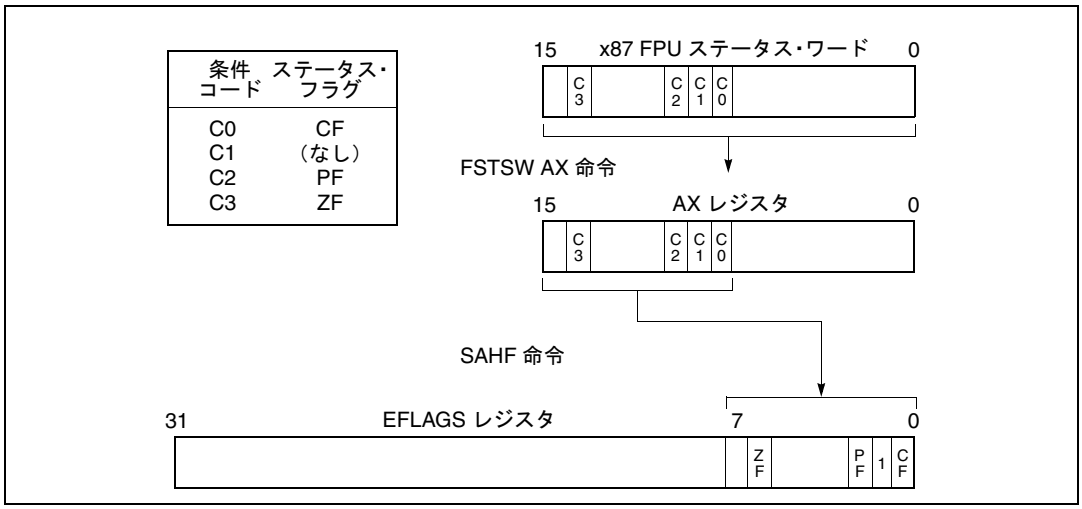


図 8-5. 条件コードの EFLAGS レジスタへの移動

新メカニズムは、P6 ファミリ・プロセッサでしか使用できない。このメカニズムでは、新しい浮動小数点比較命令と EFLAGS 設定命令（FCOMI、FCOMIP、FUCOMI、FUCOMIP）を使用して 2 つの浮動小数点値を比較し、EFLAGS レジスタの ZF、PF、CF フラグを直接設定する。このメカニズムでは、旧メカニズムで必要だった 3 つの命令を 1 つの命令に置き換えられる。

また、P6 ファミリ・プロセッサで新たに導入された FCMOV*cc* 命令を使用しても、EFLAGS レジスタのステータス・フラグ（ZF、PF、CF）の設定に基づいて浮動小数点値（x87 FPU データレジスタの値）の条件付き移動ができるので注意する。これらの命令を使用すれば、浮動小数点値の条件付き移動を実行する際に IF ステートメントが不要になる。

8.1.4. x87 FPU 制御ワード

16 ビットの x87 FPU 制御ワード（図 8-6. を参照）は、使用する x87 FPU の精度と丸めの方法を制御するためのものである。このワードはまた、x87 FPU 浮動小数点例外マスクビットも格納する。x87 FPU 制御ワードは、x87 FPU 制御レジスタにキャッシュされる。x87 FPU 制御レジスタの内容は、FLDCW 命令を使用してロードし、FSTCW/FNSTCW 命令を使用してメモリに格納できる。

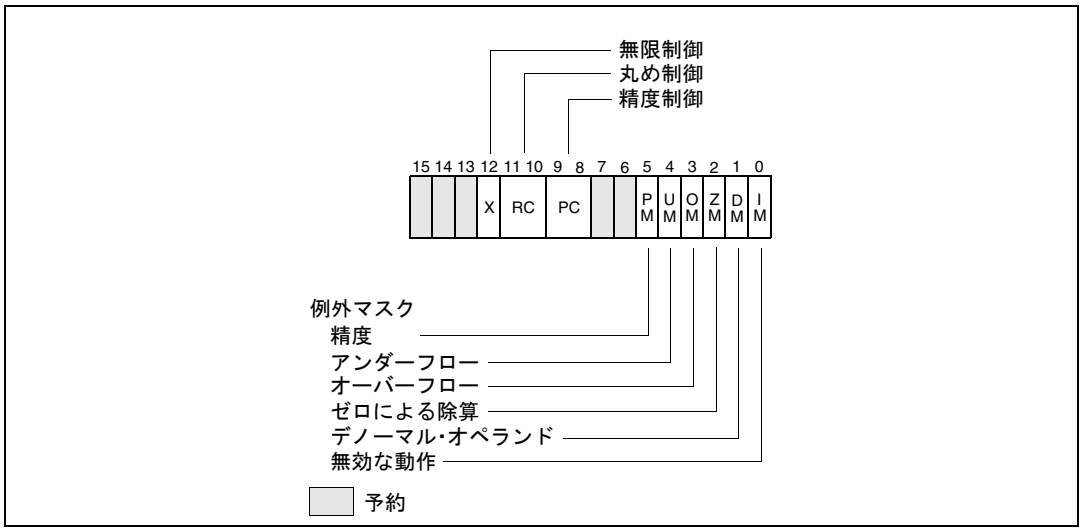


図 8-6. x87 FPU 制御ワード

FINIT/FNINIT か FSAVE/FNSAVEFPU のいずれかの命令を使用して x87 FPU を初期化すると、x87 FPU 制御ワードは 037FH に設定される。この場合、すべての浮動小数点例外がマスクされ、丸めモードは最近値に設定され、x87 FPU の精度は 64 ビットに設定される。

8.1.4.1. x87 FPU 浮動小数点例外フラグマスク

例外フラグ・マスク・ビット（x87 FPU 制御ワードのビット 0 ～ 5）は、x87 FPU ステータス・ワードの 6 つの浮動小数点例外フラグをマスクする。これらのマスクビットのいずれかがセットされると、それに対応する x87 FPU 浮動小数点例外の生成が阻止される。

8.1.4.2. 精度制御フィールド

精度制御（PC）フィールド（x87 FPU 制御ワードのビット 8 ～ 9）は、x87 FPU が行う浮動小数点計算の精度（64 ビット、53 ビット、または 24 ビット）を決定する（表 8-2. を参照）。デフォルトの精度は、拡張倍精度である。デフォルトの精度は拡張精度である。この精度では、x87 FPU データレジスタの拡張倍精度浮動小数点フォーマットで使用可能なフル 64 ビットの仮数部が使用される。この設定にすれば、アプリケーションは、x87 FPU データレジスタで可能な最大の精度をフルに利用できる。したがって、この設定は、ほとんどのアプリケーションに最適である。

表 8-2. 精度制御フィールド（PC）

精度	PC フィールド
単精度（24-Bits*）	00B
予約	01B
倍精度（53-Bits*）	10B
拡張倍精度（64-Bits）	11B

倍精度や単精度の設定では、仮数部のサイズがそれぞれ 53 ビットと 24 ビットに縮小される。これらの設定が用意されているのは、IEEE 規格をサポートすると共に、既存のプログラミング言語の仕様との互換性を保つためである。これらの設定を使用すると、拡張倍精度浮動小数点フォーマットが持つ 64 ビット長の仮数部ならではのメリットは失われる。低い精度を指定した場合、仮数部の値の丸めによって、右側の使用されないビットがゼロにクリアされる。

精度制御ビットは、FADD、FADDP、FIADD、FSUB、FSUBP、FISUB、FSUBR、FSUBRP、FISUBR、FMUL、FMULP、FIMUL、FDIV、FDIVP、FIDIV、FDIVR、FDIVRP、FIDIVR、FSQRT の各浮動小数点命令の結果に対してのみ有効である。

8.1.4.3. 丸め制御フィールド

x87 FPU 制御レジスタの丸め制御（RC）フィールド（ビット 10 ～ 11）は、x87 FPU 浮動小数点命令の結果を丸める方法を制御する。浮動小数点値の丸めについては、4.8.4. 項「丸め」を参照のこと。RC フィールドのエンコーディングについては、4.8.4.1. 項「丸め制御（RC）フィールド」を参照のこと。

8.1.5. 無限大制御フラグ

無限大制御フラグ（x87 FPU 制御ワードのビット 12）は、インテル® 287 数値演算コプロセッサとの互換性を維持するために用意されたフラグである。したがって、後発バージョンの x87 FPU コプロセッサ、または IA-32 プロセッサに対しては意味を持たない。x87 FPU が無限大値を処理する方法については、4.8.3.3. 項「符号付き無限大」を参照のこと。

8.1.6. x87 FPU タグワード

この 16 ビット・ビットのタグワード（図 8-7. を参照）は、x87 FPU データ・レジスタ・スタックの 8 つのレジスタそれぞれの内容を示す（レジスタごとに 2 ビット・タグが 1 つずつ対応）。これらのタグコードは、レジスタに、有効な数値、ゼロ、特殊な浮動小数点数値（NaN、無限大、デノーマル、またはサポートされていないフォーマット）のどれが格納されているか、または空であるかどうかを示す。x87 FPU タグワードは、x87 FPU の x87 FPU タグ・ワード・レジスタにキャッシュされる。FINIT/FNINIT か FSAVE/FNSAVE のいずれかの命令で x87 FPU を初期化すると、x87 FPU タグワードは FFFFH に設定される。この結果、すべての x87 FPU データレジスタが空としてマーク付けされる。

図 8-7. x87 FPU タグワード

x87 FPU タグワード内の各タグは、物理レジスタ（番号 0 ～ 7）に対応する。タグは、x87 FPU ステータス・ワードに格納されている現在のスタックのトップ（TOP）ポインタを使用して ST(0) に相対させてレジスタに関連付けることができる。

x87 FPU は、これらのタグ値を使用してスタックのオーバーフロー条件とアンダーフロー条件を検出する（8.5.1.1. 項「スタック・オーバーフロー例外またはスタック・アンダーフロー例外（#IS）」を参照）。

アプリケーション・プログラムや例外ハンドラでは、このタグ情報を使用することで、レジスタ内の実際のデータの複雑なデコーダを行わないでも、x87 FPU データレジスタの内容をチェックできる。タグレジスタを読み取るには、FSTENV/FNSTENV か FSAVE/FNSAVE のいずれかの命令を使用して、レジスタの内容をメモリに格納しなければならない。これらの命令のいずれかでセーブした後のメモリ内のタグワードの位置を、図 8-9. ～図 8-12. に示す。

タグレジスタ内のタグは、ソフトウェア上で直接ロードしたり、変更することはできない。FLDENV 命令と FRSTOR 命令では、タグレジスタのイメージを x87 FPU にロードするが、x87 FPU がそれらのタグの値を使用するのは、データレジスタが空である（11B）か、空でない（00B、01B、または 10B）かを確認するためだけである。

タグレジスタのイメージがデータレジスタが空であることを示している場合は、そのデータレジスタに対するタグレジスタのタグは空（11B）とマーク付けされる。タグレジスタのイメージがデータレジスタが空でないことを示している場合は、x87 FPU はデータレジスタ内の実際の値を読み取り、その値にしたがってレジスタのタグを設定する。この動作により、プログラム上でタグレジスタの値を設定して、空でないデータレジスタの実際の内容を間違って示さないようにできる。

8.1.7. x87 FPU 命令とデータ（オペランド）ポインタ

x87 FPU は、最後に実行された非制御型の命令に対し、その命令とデータ（オペランド）に対するポインタを、2 つの 48 ビットレジスタ（x87 FPU 命令ポインタレジスタと x87 FPU オペランド（データ）ポインタ・レジスタ）に格納する（図 8-1. を参照）。（これらのポインタをセーブするのは、例外ハンドラにステート情報を提供するためである。）

ただし、x87 FPU データ・ポインタ・レジスタの値は、常にメモリ・オペランドへのポインタである。最後に実行された非制御命令がメモリ・オペランドを使用しない場合は、データ・ポインタ・レジスタの値は未定義（予約済み）である。

x87 FPU 命令ポインタ・レジスタとデータ・ポインタ・レジスタの内容は、制御命令（FINIT/FNINIT、FCLEX/FNCLEX、FLDCW、FSTCW/FNSTCW、FSTSW/FNSTSW、FSTENV/FNSTENV、FLDENV、FSAVE/FNSAVE、FRSTOR、WAIT/FWAIT）のどれが実行されても変更されることはない。

x87 FPU 命令ポインタレジスタとデータ・ポインタ・レジスタに格納されるポインタは、オフセット（ビット 0 ～ 31 に格納される）とセグメント・セレクタ（ビット 32 ～ 47 に格納される）で構成される。

これらのレジスタには、FSTENV/FNSTENV、FLDENV、FINIT/FNINIT、FSAVE/FNSAVE、FRSTOR、FXSAVE、FXRSTOR の各命令を使ってアクセスできる。これらのレジスタをクリアするには、FINIT/FNINIT 命令と FSAVE/FNSAVE 命令を使用する。

8087 以外のすべての x87 FPU や NPX では、命令の前にプリフィックスがあれば x87 FPU 命令ポインタはそのプリフィックスをポイントする。8087 の場合は、x87 FPU 命令ポインタは実際のオペコードだけをポイントする。

8.1.8. 最後の命令オペコード

x87 FPUは、最後に実行された非制御命令のオペコードを11ビットのx87 FPUオペコード・レジスタに格納する。（この情報は、例外ハンドラにステート情報を提供するためのものである。）x87 FPU オペコード・レジスタには、（すべてのプリフィックスの後に続く）オペコードの 1 番目のバイトと 2 番目のバイトだけが格納される。図 8-8. に、これら 2 つのバイトのエンコーディングを示す。オペコードの 1 番目のバイトの上位 5 ビットは、すべての浮動小数点オペコード（11011B）に対して同じであるため、オペコード・レジスタにはこのバイトの下位 3 ビットだけが格納される。

8.1.8.1. fopcode 互換モード

インテル® Pentium® 4 プロセッサとインテル® Xeon™ プロセッサから、IA-32 アーキテクチャは、最後の命令オペコード（fopcode と呼ばれる）の格納方法をプログラムによって制御できるようになった。IA32_MISC_ENABLE MSR のビット 2 は、fopcode 互換モードを有効（セット）または無効（クリア）にする。

FOP コード互換モードが有効になっている場合、FOP は以前の IA32 アーキテクチャと同じように定義される（常に、FSAVE/FSTENV/FXSAVE の前に実行された最後の非透過的な FP 命令の FOP として定義）。FOP コード互換モードが無効になっている場合（デフォルト）、FOP は、FSAVE/FSTENV/FXSAVE の前に実行された最後の非透過的な FP 命令に、マスクされていない例外があったときのみ有効である。

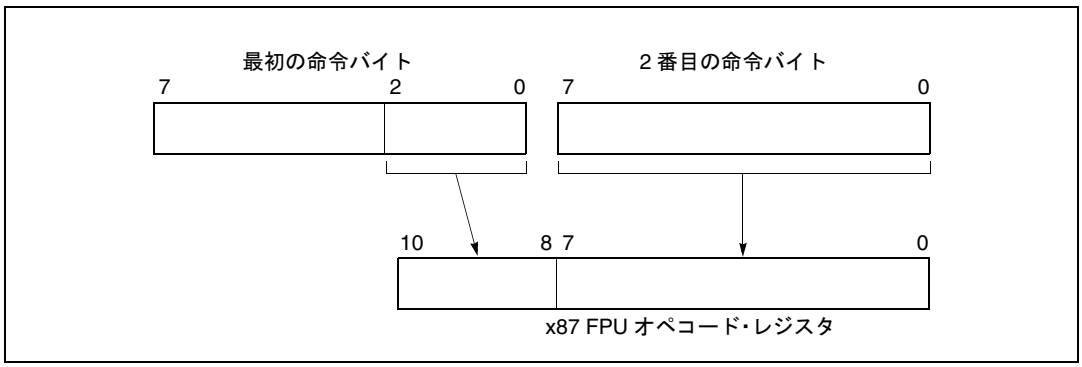


図 8-8. x87 FPU オペコード・レジスタの内容

fopcode 互換モードは、x87 FPU 浮動小数点例外ハンドラが、fopcode を使用してプログラムのパフォーマンスを分析したり、例外の処理後にプログラムを再起動する場合にのみ、有効にすることをお勧めする。

8.1.9. FSTENV/FNSTENV 命令および FSAVE/FNSAVE 命令による x87 FPU のステートのセーブ

FSTENV/FNSTENV 命令と FSAVE/FNSAVE 命令は、例外ハンドラや、他のシステム・ソフトウェア、アプリケーション・ソフトウェア上で使用できるように、x87 FPU のステート情報をメモリに格納する。FSTENV/FNSTENV 命令は、ステータス、制御、タグ、x87 FPU 命令ポインタ、FPU オペランド・ポインタ、オペコードの各レジスタの内容をセーブする。FSAVE/FNSAVE 命令は、それらの情報に加え、x87 FPU データレジスタの内容を格納する。FSAVE/FNSAVE 命令は、（FINIT/FNINIT 命令と同じように）x87 FPU の元のステートをセーブしてから、x87 FPU をデフォルト値に初期化する点に注意する。

これらの情報がどのようにメモリに格納されるかは、プロセッサの動作モード（保護モードか実アドレスモード）と、有効なオペランド・サイズ属性（32 ビットか 16 ビット）によって決まる。図 8-9. ～図 8-12. を参照のこと。仮想 8086 モードまたは SMM では、図 8-12. に示す実アドレスモード形式が使用される。SMM において FPU を使用する場合の注意点については、『IA-32 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、下巻』の第 13 章「システム管理モード（SMM）」を参照のこと。

x87 FPU ステート情報は、FLDENV 命令または FRSTOR 命令を使ってメモリから x87 FPU にロードできる。この場合、FLDENV 命令を使用すると、ステータス、制御、タグ、x87 FPU 命令ポインタ、x87 FPU オペランド・ポインタ、オペコードの各レジスタだけがロードされる。FRSTOR 命令を使用した場合は、x87 FPU スタック・レジスタを含むすべての x87 FPU レジスタがロードされる。

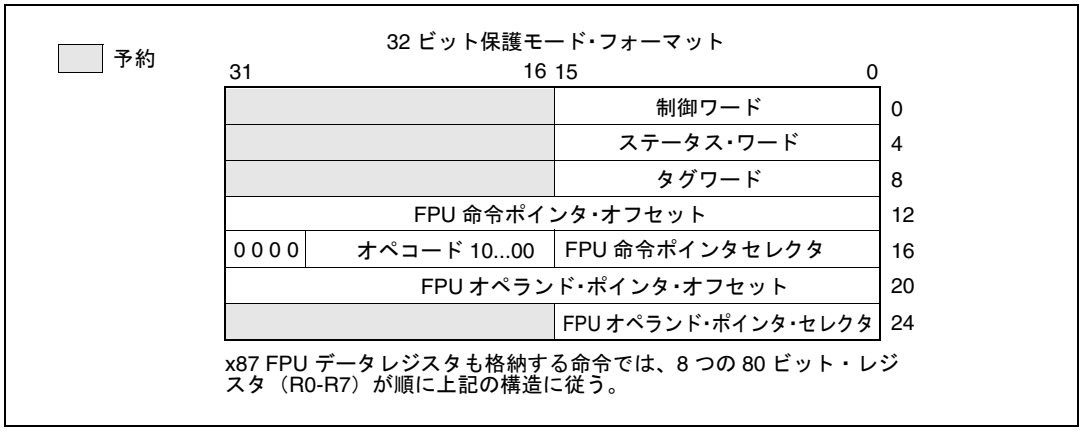


図 8-9. 保護モードにおけるメモリ内の x87 FPU ステートイメージ（32 ビット・フォーマット）

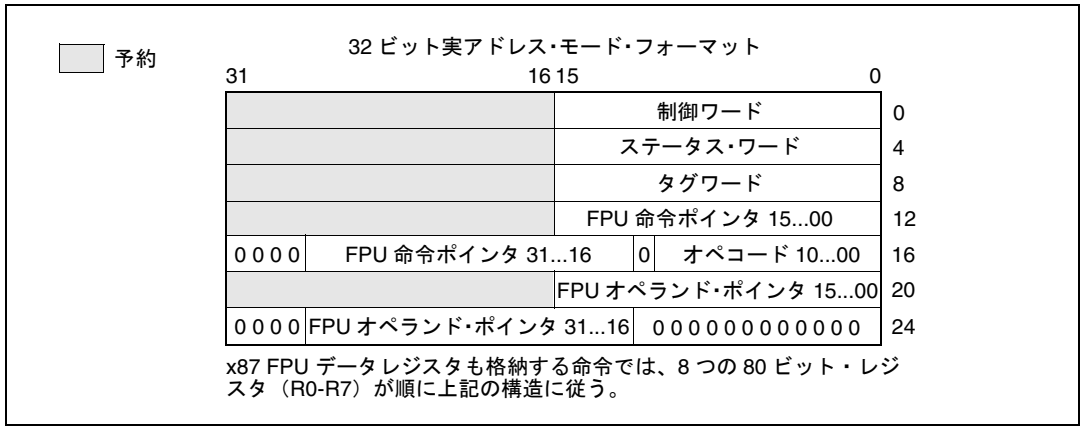


図 8-10. 実アドレスモードにおけるメモリ内の x 87 FPU ステートイメージ
（32 ビット・フォーマット）

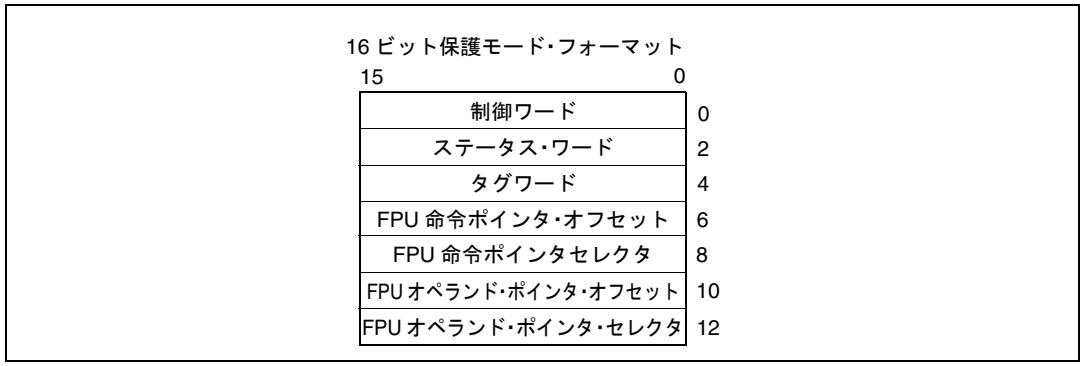


図 8-11. 保護モードにおけるメモリ内の x87 FPU ステートイメージ
（16 ビット・フォーマット）

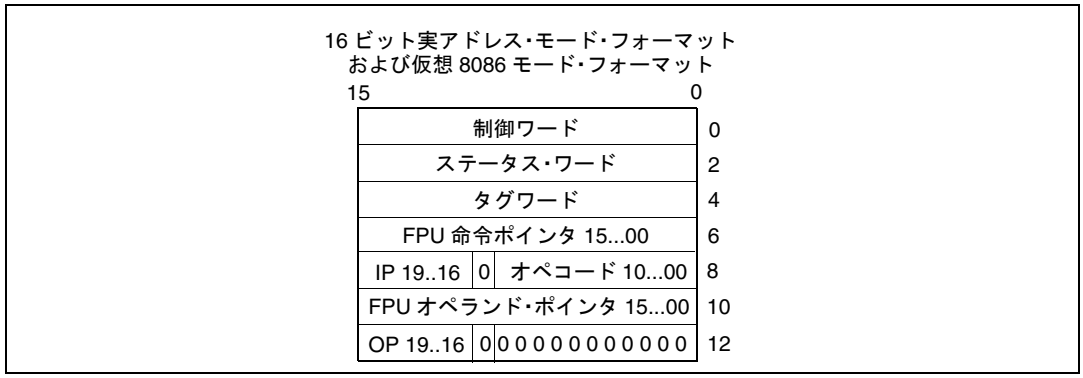


図 8-12. 実アドレスモードにおけるメモリ内の x87 FPU ステートイメージ
（16 ビット・フォーマット）

8.1.10. FXSAVE 命令による x87 FPU ステートの保存

FXSAVE 命令は、x87 FPU ステートと、XMM レジスタおよびMXCSR レジスタの状態を保存する。FXRSTOR 命令は、これらの状態をリストアする。FXSAVE 命令を使用してx87 FPUステートを保存すると、次の2つのメリットがある。(1) FXSAVEはFSAVEより高速で実行される。(2) FXSAVEは、1回の操作でx87 FPU、MMX、XMMステート全体を保存する。これらの命令についての詳細は、10.5.節「FXSAVE命令とFXRSTOR命令」を参照のこと。

8.2. x87 FPU データ型

x87 FPU は、単精度浮動小数点、倍精度浮動小数点、拡張倍精度浮動小数点、符号付きワード整数、符号付きダブルワード整数、符号付きクワッドワード整数、およびパックド BCD 10 進整数の 7 種類のデータ型を認識し、それらのデータ型を操作する(図 8-13. を参照)。

これらのデータ型についての詳細は、4.2.2. 項「浮動小数点データ型」、4.2.1.2. 項「符号付き整数」、4.7. 節「BCD およびパックド BCD 整数」を参照のこと。

これらのデータ型は、80 ビットの拡張倍精度浮動小数点フォーマットを除き、すべてメモリ内でだけの表現である。x87 FPU データレジスタにロードされると、これらのデータ型は拡張倍精度浮動小数点フォーマットに変換され、そのフォーマットで操作が行われる。

各浮動小数点型では、IEEE 規格 754 の規定にしたがって、デノーマル値もサポートされる。単精度または倍精度浮動小数点フォーマットのデノーマル数がソース・オペランドとして使用されたとき、デノーマル例外がマスクされている場合は、x87 FPU は、この数値を拡張倍精度フォーマットに変換するときに自動的に**正規化**する。

メモリに格納されるときは、x87 FPU データ型の値の最下位バイトが、その値に対して指定されている先頭アドレスに格納される。これ以後、この後に続くバイトがメモリ内の高いアドレスに向かって順番に格納される。浮動小数点命令では、オペランドの先頭アドレスだけを使用してメモリ・オペランドのロードやストアを行う。

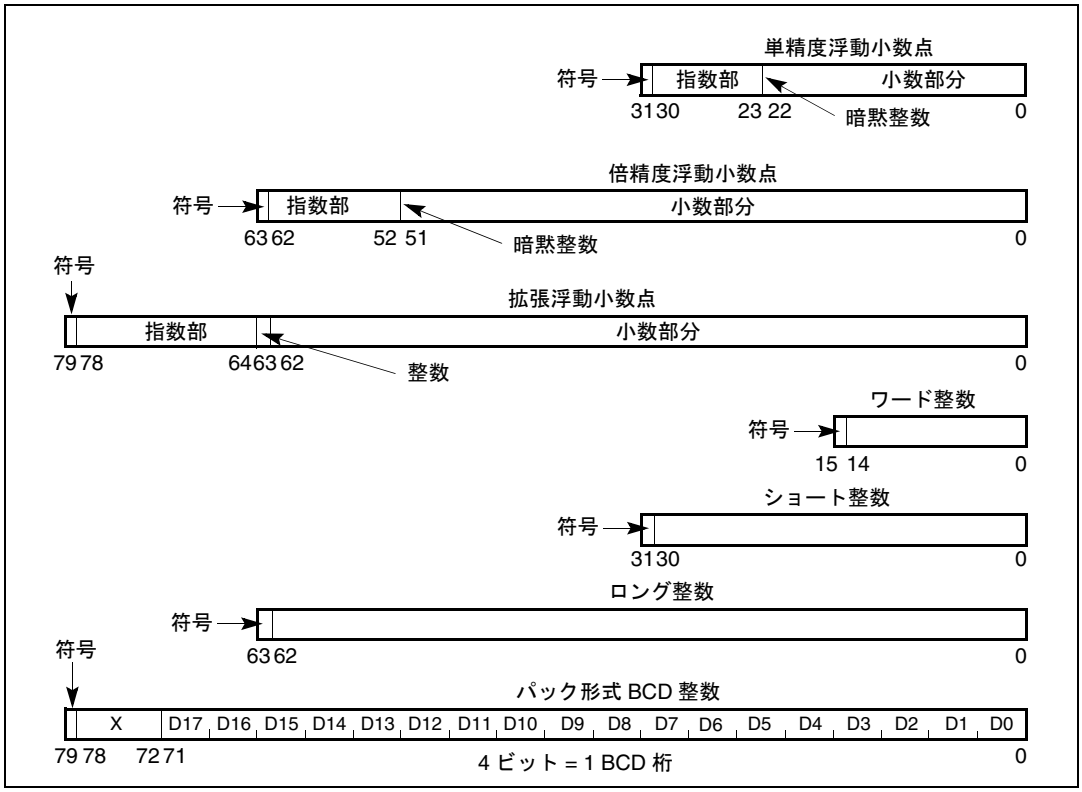


図 8-13. x87 FPU データ型のフォーマット

原則として、数値は倍精度フォーマットでメモリに格納しなければならない。このフォーマットは、プログラマによる作業を最小限に抑えながら、正しい結果を得るのに十分な範囲と精度を提供する。単精度フォーマットでは丸めによる問題が早い段階で明らかになるため、アルゴリズムをデバッグする上では効果的である。拡張倍精度フォーマットは、通常はx87 FPUのレジスタの中間結果や定数を保持する目的で使用される。このフォーマットは特に大きなビット数を持つため、中間段階での計算における丸めやオーバーフロー / アンダーフローの影響が最終結果に及ぶのを防ぐことができる。ただし、アプリケーションがデータの格納、計算、結果に対してx87 FPUが持つ最大の範囲と精度を必要とする場合は、数値を拡張倍精度形式でメモリに格納してもよい。

8.2.1. 不定値

x87 FPUの各データ型について、**不定値**と呼ばれる特殊な値を表現するために、1つの独自のエンコーディングが予約されている。x87 FPU命令は、一部のマスクされている浮動小数点無効操作例外に対する応答として、不定値を返す。整数不定値、QNaN

浮動小数点不定値、パックドBCD整数不定値のエンコーディングについては、それぞれ表4-1、表4-3、表4-4を参照のこと。

2進整数のコード100..00Bは、コードが使用される状況によって次のいずれかを表す。

- -2^{15}、-2^{31}、または-2^{63}のいずれかのフォーマットでサポートされる負の最大数。
- **整数不定**値。

このコードが（整数ロード命令や整数算術命令などの）ソース・オペラ ンドとして使用された場合は、x87 FPUはそれを、使用されているフォーマットで表現可能な負の最大数として解釈する。FIST/FISTP命令で整数値をメモリに格納する際に、x87 FPUが無効操作を検出した場合で、しかも無効操作例外がマスクされていた場合は、x87 FPUは例外に対するマスク応答として、デスティネーション・オペランドに整数不定値のエンコーディングを格納する。このエンコーディングだけではこの値が格納された理由が不明確な場合は、無効操作例外フラグを調べれば、この値が例外に対する応答として生成されたものかどうかを確認できる。

8.2.2. サポートされない拡張倍精度浮動小数点のエンコーディングと疑似デノーマル

拡張倍精度浮動小数点フォーマットでは、表4-4.に示すカテゴリのいずれにも分類されない多くのエンコーディングが可能になる。表8-3.に、これらのサポートされていないエンコーディングを示す。これらのエンコーディングの一部はインテル® 287数値演算コプロセッサでサポートされていたが、インテル® 387数値演算コプロセッサや、後発のIA-32プロセッサではその大部分がサポートされていない。これらのエンコーディングは、IEEE規格754の最終バージョンで行われた変更によって削除されたため、今後はサポートされない。

特に、これまで疑似NaN、疑似無限大、非ノーマル数とされてきたエンコーディングのカテゴリもサポートされない。これらはオペランド値として使用してはならない。インテル® 387数値演算コプロセッサと、後発のIA-32プロセッサは、オペランドとしてこれらを検出した時点で無効操作例外を生成する。

これまで疑似デノーマル数とされてきたエンコーディングは、インテル387数値演算コプロセッサ以降のIA-32プロセッサでは生成が行われない。ただし、オペランドとして検出された場合は正しく処理される。つまり、デノーマルとして処理され、デノーマル例外が生成される。疑似デノーマル数は、オペランド値として使用してはならない。これら現行のIA-32プロセッサでは、レガシーコードに対処することを目的にサポートされている。

表 8-3. サポートされていない拡張倍精度浮動小数点のエンコーディングと疑似デノーマル

クラス		符号	バイアス付き指数	仮数部	
				整数部分	小数部分
正の疑似 NaN	クワイエット型	0 . 0	11..11 . 11..11	0	11..11 . 10..00
	シグナル型	0 . 0	11..11 . 11..11	0	01..11 . 00..01
正の浮動小数点	疑似無限大	0	11..11	0	00..00
	アンノーマル	0 . 0	11..10 . 00..01	0	11..11 . 00..00
	疑似デノーマル	0 . 0	00..00 . 00..00	1	11..11 . 00..00
負の浮動小数点	疑似デノーマル	1 . 1	00..00 . 00..00	1	11..11 . 00..00
	アンノーマル	1 . 1	11..10 . 00..01	0	11..01 . 00..00
	疑似無限大	1	11..11	0	00..00
負の疑似 NaN	シグナル型	1 . 1	11..11 . 11..11	0	01..11 . 00..01
	クワイエット型	1 . 1	11..11 . 11..11	0	11..11 . 10..00
			← 15 ビット →		← 63 ビット →

8.3. x87 FPU 命令セット

x87 FPU がサポートする浮動小数点命令は、機能によって次の 6 つのグループに分類できる。

- データ転送命令
- 基本算術命令
- 比較命令
- 超越関数命令
- 定数ロード命令
- FPU 制御命令

浮動小数点命令のカテゴリ別一覧は、5.2. 節「x87 FPU 命令」に掲載している。

以降の各項では、それぞれのカテゴリの命令を簡単に説明する。浮動小数点命令の詳しい説明については、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」を参照のこと。

8.3.1. エスケープ（ESC）命令

x87 FPU 命令セットの命令はすべて、エスケープ（ESC）命令として知られている命令クラスに分類される。これらの命令はすべて、共通のオペコード・フォーマットを持つ。このオペコードの最初のバイトは、D8H ～ DFH の範囲内の数値になる。

8.3.2. x87 FPU 命令のオペランド

大部分の浮動小数点命令は、x87 FPU のデータ・レジスタ・スタックまたはメモリに配置されたオペランドを 1 つ、または 2 つ必要とする。（浮動小数点命令はいずれも、即値オペランドは受け入れない。）

オペランドがデータレジスタにある場合は、オペランドは物理レジスタ番号によってではなく、ST(0) レジスタ（レジスタスタックのトップにあるレジスタ）に対して相対的に参照される。ST(0) レジスタは暗黙のオペランドであることが多い。

メモリ内のオペランドは、3.7. 節「オペランドのアドレス指定」のオペランド・アドレス指定方法と同じ方法で参照できる。

8.3.3. データ転送命令

データ転送命令（表 8-4. を参照）は、次の操作を実行する。

- 浮動小数点、整数、またはパック形式 BCD のオペランドを、メモリから ST(0) レジスタにロードする。
- ST(0) レジスタの値を、浮動小数点、整数、またはパック形式 BCD フォーマットでメモリに格納する。
- x87 FPU レジスタスタックのレジスタ間で値を移動する。

FLD（Load floating point）命令は、浮動小数点オペランドをメモリから x87 FPU データ・レジスタ・スタックのトップにプッシュする。オペランドが単精度浮動小数点または倍精度浮動小数点のフォーマットの場合は、オペランドは自動的に拡張倍精度浮動小数点フォーマットに変換される。この命令はまた、指定の x87 FPU データレジスタの値をレジスタスタックのトップにプッシュする場合にも使用できる。

FILD（Load integer）命令は、メモリ内の整数オペランドを拡張倍精度浮動小数点フォーマットに変換し、その値をレジスタスタックのトップにプッシュする。

表 8-4. データ転送命令

実数		整数		パック形式 10 進	
FLD	Load Floating Point	FILD	Load Integer	FBLD	Load Packed Decimal
FST	Store Floating Point	FIST	Store Integer		
FSTP	Store Floating Point and Pop	FISTP	Store Integer and Pop	FBSTP	Store Packed Decimal and Pop
FXCH	Exchange Register Contents				
FCMOV*cc*	Conditional Move				

FBLD（Load packed decimal）命令は、これと同じロード操作を、メモリ内のパック形式 BCD オペランドに対して実行する。

FST（Store floating point）命令と FIST（Store integer）命令は、レジスタ ST(0) の値をデスティネーション・フォーマット（それぞれ浮動小数点または整数）でメモリに格納する。この場合も、フォーマットの変換が自動的に行われる。

FSTP（Store floating point and pop）、FISTP（Store integer and pop）、および FBSTP（Store packed decimal and pop）の各命令は、ST(0) レジスタの値をデスティネーション・フォーマット（浮動小数点、整数、またはパック形式 BCD）でメモリに格納し、その後でレジスタスタックに対して**ポップ**操作を実行する。ポップ操作の結果、ST(0) レジスタが空としてマーク付けされ、x87 FPU 制御ワードのスタックポインタ（TOP）が 1 だ

けインクリメントされる。FSTP 命令はまた、ST(0) レジスタの値を別の x87 FPU レジスタ [ST(i)] にコピーする場合にも使用できる。

FXCH（Exchange register contents）命令は、選択されたスタックのレジスタ [ST(i)] の値と ST(0) の値を交換する。

条件コード（cc）で指定された条件が満たされた場合、FCMOV*cc*（Conditional move）命令は、選択されたスタックのレジスタ [ST (i)] の値をレジスタ ST (0) に移動する（表 8-5. を参照）。テストされる条件は、EFLAGS レジスタのステータス・フラグで表される。FCMOVcc 命令のニーモニックは、文字 "FCMOV" の後に条件コード・ニーモニックを付け加えたものである。

表 8-5. 浮動小数点条件付き移動命令

命令ニーモニック	ステータス・フラグのステート	条件の説明
FCMOVB	CF=1	より小
FCMOVNB	CF=0	より小でない
FCMOVE	ZF=1	等しい
FCMOVNE	ZF=0	等しくない
FCMOVBE	CF または ZF=1	より小または等しい
FCMOVNBE	CF または ZF=0	より小でも等しくもない
FCMOVU	PF=1	順序化不可能
FCMOVNU	PF=0	順序化不可能でない

CMOV*cc* 命令と同様、FCMOV*cc* 命令は小規模な IF 文構造を最適化するのに便利である。これらの命令はまた、IF 文操作の分岐によるオーバーヘッドや、プロセッサによる分岐の予測ミスを排除する上でも有効である。

ソフトウェア上で、CPUID 命令でプロセッサの機能情報をチェックすれば、FCMOV*cc* 命令がサポートされているかどうかを確認できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「CPUID - CPU Identification」を参照）。

8.3.4. 定数ロード命令

次に挙げる命令は、よく使用される定数を x87 FPU レジスタスタックのトップ [ST(0)] にプッシュする。

FLDZ Load +0.0
FLD1 Load +1.0
FLDPI Load π
FLDL2T Load $\log_2 10$
FLDL2E Load $\log_2 e$

FLDLG2	Load $\log_{10}2$
FLDLN2	Load $\log_e2$

定数値は、フルの拡張倍精度浮動小数点精度（64 ビット）を持ち、これは 10 進のほぼ 19 ケタの精度に相当する。定数値は、内部的には拡張倍精度浮動小数点より精度の高いフォーマットで格納される。定数をロードするときは、x87 FPU はこの高精度の内部定数を、x87 FPU 制御ワードの RC（丸め制御）フィールドにしたがって丸める。この丸めの結果として不正確結果例外（#P）は発生せず、値が丸められても x87 FPU ステータス・ワードに C1 フラグはセットされない。π 定数については、8.3.8. 項「π」を参照のこと。

8.3.5. 基本算術命令

次に挙げる浮動小数点命令は、実数に関して基本的な算術演算を実行する。これらの命令は、IEEE 規格 754 に準拠している。

FADD/FADDP	Add floating point
FIADD	Add integer to floating point
FSUB/FSUBP	Subtract floating point
FISUB	Subtract integer from floating point
FSUBR/FSUBRP	Reverse subtract floating point
FISUBR	Reverse subtract floating point from integer
FMUL/FMULP	Multiply floating point
FIMUL	Multiply integer by floating point
FDIV/FDIVP	Divide floating point
FIDIV	Divide floating point by integer
FDIVR/FDIVRP	Reverse divide
FIDIVR	Reverse divide integer by floating point
FABS	Absolute value
FCHS	Change sign
FSQRT	Square root
FPREM	Partial remainder
FPREM1	IEEE partial remainder
FRNDINT	Round to integral value
FXTRACT	Extract exponent and significand

加算、減算、乗算、および除算の各命令は、次に挙げるタイプのオペランドに対して演算を行う

- 2 つの x87 FPU レジスタ値。
- x87 FPU データレジスタ値と、メモリ内の浮動小数点値または整数値。

(データ・レジスタ・スタック上でオペランドがどのように参照されるかについては、8.1.1. 項「x87 FPU データレジスタ」を参照)。

メモリ内のオペランドは、単精度浮動小数点、倍精度浮動小数点、ワード整数、またはダブルワード整数のフォーマットになる。これらのオペランドは、自動的に拡張倍精度浮動小数点フォーマットに変換される。

減算命令と除算命令の逆バージョン（FSUBR と FDIVR）を使用して、効率的なコーディングが行える。例えば、指定した x87 FPU データレジスタ ST(*i*) と ST(0) レジスタの値を操作する場合、FSUB 命令と FSUBR 命令には以下のオプションを利用できる。

FSUB:
ST(0) ← ST(0) – ST(*i*)
ST(*i*) ← ST(*i*) – ST(0)

FSUBR:
ST(0) ← ST(*i*) – ST(0)
ST(*i*) ← ST(0) – ST(*i*)

これらの命令によって、減算や除算を実行する際に、レジスタ ST(0) と他の x87 FPU レジスタの間で値を交換する必要がなくなる。

加算、減算、乗算、除算命令のポップ版は、算術演算の後に x87 FPU レジスタスタックをポップする。これらの命令は、ST(*i*) レジスタと ST(0) レジスタの値を操作して、その結果を ST(*i*) レジスタに格納し、ST(0) レジスタをポップする。

FPREM 命令は、インテル 8087 やインテル 287 の数値演算コプロセッサが使用する方法で、2 つのオペランドの除算の剰余を計算する。一方、FPREM1 命令は、IEEE 754 規格で定義されている方法で剰余を計算する。

FSQRT 命令はソース・オペランドの平方根を計算する。

FRNDINT 命令は、x87 FPU 制御ワードの RC フィールドで指定されている丸めモードにしたがって、浮動小数点値を整数の最近値に丸める。

FABS、FCHS、および FXTRACT の各命令は、便利な算術演算を実行する。FABS 命令は、ソース・オペランドの絶対値を生成する。FCHS 命令は、ソース・オペランドの符号を変更する。FXTRACT 命令は、ソース・オペランドを指数部と小数部分に分け、それぞれの値を浮動小数点フォーマットでレジスタに格納する。

8.3.6. 比較命令と分類命令

次に挙げる命令は、実数値の比較または分類を行う。

FCOM/FCOMP/FCOMPP	Compare floating point and set x87 FPU condition code flags.
FUCOM/FUCOMP/FUCOMPP	Unordered compare floating point and set x87 FPU condition code flags.
FICOM/FICOMP	Compare integer and set x87 FPU condition code flags.
FCOMI/FCOMIP	Compare floating point and set EFLAGS status flags.
FUCOMI/FUCOMIP	Unordered compare floating point and set EFLAGS status flags.
FTST	Test (compare floating point with 0.0).
FXAM	Examine.

浮動小数点値の比較は、整数の比較とは異なる。これは、浮動小数点値が、より小さい、等しい、より大きい、順序化不可能という 4 つ（3 つではなく）の互いに排他的な関係を持つためである。

順序化不可能という関係は、比較の対象となる 2 つの値の少なくとも一方が NaN であるか、またはサポートされていないフォーマットである場合に真になる。この追加的な関係が必要になるのは、定義上 NaN が数値ではなく、したがって他の浮動小数点値との間で、より小、等しい、またはより大などの関係を持てないためである。

FCOM、FCOMP、および FCOMPP の各命令は、レジスタ ST(0) の値を浮動小数点のソース・オペランドと比較し、その結果にしたがって x87 FPU ステータス・ワードの条件コードフラグ（C0、C2、および C3）を設定する（表 8-6. を参照）。

順序化不可能条件（比較対象の 2 つの値の一方または両方が NaN であるか、未定義フォーマットである）が検出されても、浮動小数点無効操作例外が生成される。

これらの命令のポップ・バージョンは、比較操作の終了後、x87 FPU レジスタスタックを 1 回または 2 回ポップする。

FUCOM、FUCOMP、および FUCOMPP の各命令の操作は、それぞれ FCOM、FCOMP、FCOMPP 命令の操作と同じになる。唯一の相違点は、FUCOM、FUCOMP、FCOMPP の各命令では、オペランドの一方または両方が QNaN であったために順序化不可能条件が検出された場合に、浮動小数点無効操作例外が生成されないことである。

表 8-6. 浮動小数点値比較における x87 FPU 条件コードフラグの設定

条件	C3	C2	C0
ST(0) > ソース・オペランド	0	0	0
ST(0) < ソース・オペランド	0	0	1
ST(0) = ソース・オペランド	1	0	0
順序化不可能	1	1	1

FICOM 命令と FICOMP 命令の操作も、ソース・オペランドがメモリ内の整数値である点を除けば、それぞれ FCOM 命令と FCOMP 命令の操作と同じである。整数値は、比較が行われる前に自動的に拡張倍精度浮動小数点値に変換される。FICOMP 命令は、比較操作後に x87 FPU レジスタスタックをポップする。

FTST 命令の操作は、ST(0) レジスタの値が常に値 0.0 と比較される点を除けば、FCOM 命令の操作と同じである。

FCOMI 命令と FCOMIP 命令は、P6 ファミリ・プロセッサで IA-32 アーキテクチャに導入された。これらの命令の操作は、比較の結果を示すために（表 8-7. を参照）、x87 FPU 条件コードフラグではなく EFLAGS レジスタのステータス・フラグ（ZF、PF、CF）を設定することを除けば、それぞれ FCOM 命令と FCOMP 命令と同じである。FCOMI 命令と FCOMIP 命令では、比較の結果から条件付き分岐命令（J*cc*）を直接実行することができる。

表 8-7. 浮動小数点値比較における EFLAGS ステータス・フラグの設定

比較結果	ZF	PF	CF
ST0 > ST(*i*)	0	0	0
ST0 < ST(*i*)	0	0	1
ST0 = ST(*i*)	1	0	0
順序化不可能	1	1	1

ソフトウェアは、CPUID 命令を使用してプロセッサの機能情報をチェックすることによって、プロセッサが FCOMI 命令と FCOMIP 命令をサポートしているかどうかを確認できる（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「CPUID - CPU Identification」を参照）。

FUCOMI 命令と FUCOMIP 命令の操作は、順序化不可能条件がオペランドの一方または両方が QNaN であったことに由来するものであっても浮動小数点無効操作例外を発生しない点を除けば、それぞれ FCOMI 命令と FCOMIP 命令の操作と同じである。FCOMIP 命令と FUCOMIP 命令は、比較操作後に x87 FPU レジスタスタックをポップする。

FXAM 命令は、ST(0) レジスタの浮動小数点値の分類クラス（ゼロ、デノーマル数、ノーマル有限数、∞、NaN、サポートされていないフォーマットのいずれか）、あるいはST(0) レジスタが空であるかどうかを判断する。この命令は、x87 FPU 条件コードフラグを設定することで、クラスを示す（『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の「FXAM - Examine」を参照）。この命令はまた、値の符号を示すために C1 フラグを設定する。

8.3.6.1. x87 FPU 条件コードに基づく分岐

プロセッサには、x87 FPU ステータス・ワードの条件コードフラグ（C0、C2、C3）の設定に基づいて分岐を行う制御フロー命令が用意されていない。これらのフラグのステートに基づいて分岐を行うためには、まず x87 FPU ステータス・ワードを整数ユニットの AX レジスタに移動しなければならない。このためには、FSTSW AX（Store status word）命令が使用できる。これらのフラグを AX レジスタに移したら、TEST 命令を使用して次のように条件付き分岐操作を制御できる。

1. 結果が順序化不可能かどうかを調べる。TEST 命令を使用して、AX レジスタの内容を定数 0400H と比較する（表 8-8. を参照）。この操作で、条件コードフラグが順序化不可能という結果を示した場合は、EFLAGS レジスタの ZF フラグがクリアされる。そうでなければ、ZF フラグがセットされる。この後、必要に応じて、JNZ 命令を使用して、順序化不可能オペランド処理用のプロシージャに制御を移すことができる。

表 8-8. TEST 命令の条件付き分岐用定数

順序	定数	分岐
ST(0) > ソース・オペランド	4500H	JZ
ST(0) < ソース・オペランド	0100H	JNZ
ST(0) = ソース・オペランド	4000H	JNZ
順序化不可能	0400H	JNZ

2. 順序化比較結果を調べる。TEST 命令に表 8-8. に示した値を使用して、結果と比較してより小、等しい、またはより大をテストする。次に、対応する条件付き分岐命令を使用して、該当するプロシージャまたはコード・セクションにプログラムの制御を転送する。

プログラムまたはプロシージャに対するテストが十分に行われ、また QNaN 結果の発生に対する定期的なチェックが組み込まれている場合は、比較を実行するたびに順序化不可能結果の有無をチェックする必要はない。

x87 FPU 条件コードに基づいて分岐を行う別の方法については、8.1.3. 項「条件コードに基づく分岐と条件付き移動」を参照のこと。

一部の非比較型 x87 FPU 命令では、x87 FPU ステータス・ワードの条件コードフラグを更新する。したがって、誤ってステータス・ワードが変更されることがないようにするため、比較操作を行ったら直ちに x87 FPU ステータス・ワードを格納するようにしなければならない。

8.3.7. 三角関数命令

次に挙げる命令は、一般的な 4 種類の三角関数を実行する。

FSIN	Sine（正弦）
FCOS	Cosine（余弦）
FSINCOS	Sine and cosine（正弦および余弦）
FPTAN	Tangent（正接）
FPATAN	Arctangent（逆正接）

これらの命令は、x87 FPU レジスタスタックの上から 1 つまたは 2 つのレジスタに対して演算を行い、それぞれの結果をスタックに返す。FSIN、FCOS、FSINCOS、FPTAN 命令のソース・オペランドは、ラジアンで指定しなければならない。FPATAN 命令のソース・オペランドは、直交座標単位で指定しなければならない。

FSINCOS 命令は、ソース・オペランド値の正弦と余弦を返す。この命令を使用した方が、FSIN 命令と FCOS 命令を続けて実行するよりも処理は高速になる。

FPATAN 命令は、ST(1) を ST(0) で割った結果の逆正接を計算し、その結果をラジアンで返す。この命令は、直交座標を極座標に変換するのに便利である。

8.3.8. π

三角関数の引き数（ソース・オペランド）が関数の範囲内にある場合、FPREM 命令や FPREM1 命令に使用されるのと同じ剰余計算機構によって、2π の整数倍で引き数が自動的に剰余計算される。x87 FPU が、引き数の剰余計算やその他の計算に使用する π の内部値は次のようになる。

$\pi = 0.f * 2^2$

ここで、

f = C90FDAA2 2168C234 C

（上記の小数部分内のスペースは、32 ビット境界を表す。）

この内部 π 値は、66 ビットの小数部分を持ち、これは拡張倍精度浮動小数点値の仮数部で許可されるビット数より 2 ビット多い。（66 ビットでは偶数の 16 進桁数にならないため、16 進フォーマットで表現できるよう、値にはゼロがさらに 2 つ追加されている。したがって、最下位 16 進桁（C）は 1100B になり、その最下位 2 ビットが小数部分のビット 67 と 68 を表す。）

この π 値は、ソース・オペランドが命令の仕様範囲内にある限り、オペランドの有効性が失われることがないように選択されたものである。

π を明示的に使用した計算の結果が、FSIN、FCOS、FSINCOS、または FPTAN のいずれかの命令で使用される場合は、π の 66 ビットの小数部分すべてを使用しなければならない。こうすることにより、計算結果とこれらの命令が使用する引き数剰余計算アルゴリズムとの整合性が保たれる。π を丸めて使用した場合は、結果ごとに不正確な値が生成される可能性がある。このような不正確な結果が計算から計算に伝播されると、無意味な結果を生じることにもなりかねない。

π を完全な 66 ビットの小数部分で表現するためには、一般には、この値を 2 つの数（上位 π と下位 π）に分割する。これらの値を合わせれば、本項の始めに示した完全な 66 ビットの小数部分を持つ π の値を得られる。

$\pi = high\pi + low\pi$

例えば、16 進の小数部分と 10 進の指数部分を持つ指数部付き表記法で与えられた次の 2 つ値は、小数部分の上位 33 ビットと下位 33 ビットを表す。

上位 π（非ノーマライズ）= $0.C90FDAA20 * 2^{+2}$

下位 π（非ノーマライズ）= $0.42D184698 * 2^{-31}$

これらの値は、IEEE の倍精度浮動小数点フォーマットでは次のようにコード化される。

上位 π = 400921FB 54400000

下位 π = 3DE0B461 1A600000

（IEEE の倍精度浮動小数点フォーマットでは、指数部分がバイアス付き（1023）になり、小数部分がノーマライズされるので注意すること。）

このような π の使い方は、拡張倍精度浮動小数点フォーマットでも記述できる。

この 2 つの部分からなる π の値を特定のアルゴリズムで使用する場合は、各部分に対して計算を並行して実行し、結果を別々に保持しなければならない。すべての計算が終わったら、2 つの結果を組み合わせれば、最終的な結果を得られる。

引き数の剰余計算を通して π の値の整合性を維持するのは複雑であるが、自動剰余計算機構の範囲内の引き数に対してだけ三角関数を適用するか、π/4 未満の絶対値までのすべての引き数の剰余計算をソフトウェアで明示的に実行すると、この複雑さを回避できる。

8.3.9. 対数、指数、スケーリング関数

次に挙げる命令は、2 つの異なる対数関数、指数関数、スケーリング関数を実行する。

FYL2X	Logarithm
FYL2XP1	Logarithm epsilon
F2XM1	Exponential
FSCALE	Scale

FYL2X 命令と FYL2XP1 命令は、2 を底とする 2 つの異なる対数演算を実行する。FYL2X 命令は、$(y * \log_2 x)$ を計算する。この演算では、次の式を使用することで任意の底の対数の計算が可能になる。

$\log_b x = (1/\log_2 b) * \log_2 x$

FYL2XP1 命令は、$(y * \log_2(x+1))$ を計算する。この演算では、0 に非常に近い x の値の最適精度が得られる。

F2XM1 命令は、(2^x-1) を計算する。この命令は、-1.0 ～ +1.0 の範囲のソース値に対してのみ演算を行う。

FSCALE 命令は、ソース・オペランドに 2 のべき乗を掛ける。

8.3.10. 超越関数命令の精度

新しい超越関数命令アルゴリズムは、インテル® Pentium® プロセッサから IA-32 アーキテクチャに組み込まれた。これらの新しいアルゴリズムは、超越関数命令 FSIN、FCOS、FSINCOS、FPTAN、FPATAN、F2XM1、FYL2X、FYL2XP1 に使用され、初期の IA-32 プロセッサおよび x87 数値演算コプロセッサより高いレベルの精度を実現するものである。これらの命令の精度は、**最後の桁位置のユニット数**（ulp）で測定できる。与えられた引き数 *x* に対し、f(*x*) と F(*x*) をそれぞれ正しい関数値と計算結果の（近似）関数値であるとすると、ulp での誤差は次のように定義できる。

$$error = \left| \frac{f(x) - F(x)}{2^{k-63}} \right|$$

k は、$1 \le 2^{-k} f(x) < 2$ の関係を満たすような整数である。

インテル Pentium プロセッサや IA-32 以降のプロセッサでは、超越関数におけるワーストケースの誤差は、偶数の最近値へ丸める場合で 1 ulp 未満、その他のモードで丸める場合で 1.5 ulp 未満になる。超越関数は、入力オペランドに関しては、命令がサポートする領域全体を通して単調関数であることが保証されている。

命令 FYL2X および FYL2XP1 は 2 オペランド命令であり、y = 1 の場合にのみ、1ulp 以内になることが保証されている。y ≠ 1 の場合の最大 ulp 誤差は、直近値へ丸める場合で常に 1.35ulp 以内になる（2 オペランド関数の場合、関数の単調性は、いずれか一方のオペランドを一定に保つことによって証明されている）。

8.3.11. x87 FPU 制御命令

次に挙げる命令は、x87 FPU の操作ステートと操作モードを制御する。これらの命令はまた、x87 FPU のステータスのチェックにも使用できる。

FINIT/FNINIT	Initialize x87 FPU
FLDCW	Load x87 FPU control word
FSTCW/FNSTCW	Store x87 FPU control word
FSTSW/FNSTSW	Store x87 FPU status word
FCLEX/FNCLEX	Clear x87 FPU exception flags
FLDENV	Load x87 FPU environment
FSTENV/FNSTENV	Store x87 FPU environment
FRSTOR	Restore x87 FPU state
FSAVE/FNSAVE	Save x87 FPU state
FINCSTP	Increment x87 FPU register stack pointer
FDECSTP	Decrement x87 FPU register stack pointer
FFREE	Free x87 FPU register
FNOP	No operation
WAIT/FWAIT	Check for and handle pending unmasked x87 FPU exceptions

FINIT/FNINIT 命令は、x87 FPU とその内部レジスタをデフォルト値に初期化する。

FLDCW 命令は、メモリから x87　FPU 制御ワードレジスタに値をロードする。FSTCW/FNSTCW 命令は、x87 FPU 制御ワードと x87 FPU ステータス・ワードをメモリに格納する。FSTSW/FNSTSW 命令は、x87 FPU 制御ワードと x87 FPU ステータス・ワードを汎用レジスタに格納する。

FSTENV/FNSTENV 命令と FSAVE/FNSAVE 命令はそれぞれ、x87 FPU の環境とステートをメモリに格納する。x87 FPU 環境には、x87 FPU のすべての制御レジスタとステータス・レジスタが含まれる。x87 FPU ステートには、x87 FPU 環境と x87 FPU レジスタスタック内のデータレジスタが含まれる。(FSAVE/FNSAVE 命令はまた、FINIT/FNINIT 命令と同じように、x87 FPU の元のステートをセーブした後で、FPU をデフォルト値に初期化する。)

FLDENV 命令と FRSTOR 命令はそれぞれ、x87 FPU 環境と x87 FPU ステートをメモリから x87 FPU にロードする。これらの命令は、一般的にタスクやコンテキストを切り替える際に使用される。

WAIT/FWAIT 命令は、同期をとるための命令である。(これらの命令は、実際には同じオペコードに対するニーモニックである。)これらの命令は、x87 FPU ステータス・ワードをチェックし、ペンディング状態のマスクされていない x87 FPU 例外の有無を確認する。ペンディング状態のマスクされていない x87 FPU 例外を検出した場合は、プロセッサはまずそれらの例外を処理し、その後で命令ストリーム上の命令(整数命令、浮動小数点命令、またはシステム命令)の実行を再開する。WAIT/FWAIT 命令が用意されているのは、x87 FPU とプロセッサの整数ユニットとの間で命令実行の同期をとるためである。WAIT/FWAIT 命令の使用法については、8.6. 節「x87 FPU 例外の同期」で詳しく説明する。

8.3.12. 同期型命令と非同期型命令

少数の特殊な制御命令を除き、すべての x87 FPU 命令は、WAIT/FWAIT 命令に似た同期操作を実行する。すなわち、命令はまずペンディング状態のマスクされていない x87 FPU 浮動小数点例外の有無を確認し、それらの例外を処理してから、それぞれの本来の操作(2 つの倍精度浮動小数点の加算など)を実行する。これらの命令は、**同期型**命令と呼ばれる。FSTSW/FNSTSW 命令などの一部の x87 FPU 制御命令には、同期型と非同期型の両バージョンが用意されている。同期型バージョン("F" プリフィックスが付く)が同期操作を実行してから本来の操作を実行するのに対し、非同期型バージョン("FN" プリフィックスが付く)はペンディング状態のマスクされていない例外を無視する。

非同期型命令を使用すれば、ソフトウェア上でペンディング状態の例外を処理せずに現在の x87 FPU ステートをセーブしたり、あるいはペンディング状態の例外に関係なく x87 FPU をリセットまたは初期化することができる。

注記

インテル® Pentium® プロセッサまたは Intel486™ プロセッサを MS-DOS* 互換モードで使用している場合は、（異常な状況の下では）ペンディング状態の x87 FPU 例外を処理するために非同期型命令の実行に割り込むことが可能である。このような事態が発生する状況と、その結果生じるプロセッサの動作については、D.2.1.3. 項「非同期型命令のウインドウ内の x87 FPU 割り込み」で詳しく説明している。P6 ファミリ・プロセッサ、インテル® Pentium® 4 プロセッサ、またはインテル® Xeon™ プロセッサを MS-DOS 互換モードで使用する場合は、この方法では非同期型命令に割り込むことはできない（D.2.2. 項「P6 ファミリおよびインテル ® Pentium® 4 プロセッサにおける MS-DOS* 互換モード」を参照）。

8.3.13. サポートされていない x87 FPU 命令

インテル ® 8087 の FENI 命令や FDISI 命令、インテル ® 287 数値演算コプロセッサの FSETPM 命令は、インテル® 387 数値演算コプロセッサや、IA-32 以降のプロセッサでは何の機能も実行しない。これらのオペコードが命令ストリーム上で検出されても、x87 FPU は特定の操作を実行せず、また x87 FPU の内部ステートも影響を受けない。

8.4. x87 FPU 浮動小数点例外処理

x87 FPU は、以下の 6 つのクラスの例外条件を検出する（4.9. 節「浮動小数点例外の概要」を参照）。

- 無効操作（#I）
- デノーマル型オペランド（#D）
- ゼロ除算（#Z）
- 数値オーバーフロー（#O）
- 数値アンダーフロー（#U）
- 不正確結果（精度）（#P）

また、無効操作例外クラスは、以下の 2 つのサブクラスに分けられる。

- スタック・オーバーフローまたはスタック・アンダーフロー（#IS）
- 無効算術演算（#IA）

6つの例外クラスそれぞれには、x87 FPUステータス・ワード内のフラグビットとx87 FPU制御ワード内のマスクビットが対応している（8.1.2.項「x87 FPUステータス・レジスタ」と8.1.4.項「x87 FPU制御ワード」を参照）。また、1つ以上のマスクされていない例外が検出されたかどうかはステータス・ワード内の例外サマリ（ES）フラグが示す。無効操作例外の2つのタイプを区別するにはステータス・ワード内のスタック・フォルト（SF）フラグで識別する。

マスクビットは、FLDCW、FRSTOR、またはFXRSTOR命令によってセットされる。これらのビットは、FSTCW/FNSTCW、FSAVE/FNSAVE、またはFXSAVE命令によって読み取られる。フラグビットは、FSTSW/FNSTSW、FSAVE/FNSAVE、またはFXSAVE命令によって読み取られる。

注記

4.9.1.項「浮動小数点例外条件」では、IA-32プロセッサが各種の浮動小数点例外を検出し、処理する機構の概要を説明している。4.9.1項の内容は、x87 FPUと、SSE、SSE2、SSE3に関連する。以下の各項では、x87 FPUに固有の浮動小数点例外の処理方法について説明する。

8.4.1. 算術命令と非算術命令

浮動小数点例外を処理する際は、**算術命令**と**非算術命令**を区別すると便利である。非算術命令は、オペランドを持たないか、持っていてもオペランドには実質的な変更は行わない。算術命令は、オペランドを有意に変更するばかりでなく、浮動小数点例外が発生するような変更を行うことが多い。表8-9.に、非算術命令と算術命令の一覧を示す。一部の非算術命令では浮動小数点スタック（フォルト）例外を通知できるが、この例外はオペランドに対する演算の結果ではない点に注意しなければならない。

表 8-9. 算術命令と非算術命令

非算術命令	算術命令
FABS	F2XM1
FCHS	FADD/FADDP
FCLEX	FBLD
FDECSTP	FBSTP
FFREE	FCOM/FCOMP/FCOMPP
FINCSTP	FCOS
FINIT/FNINIT	FDIV/FDIVP/FDIVR/FDIVRP
FLD (register-to-register)	FIADD
FLD (extended format from memory)	FICOM/FICOMP
FLD constant	FIDIV/FIDIVR
FLDCW	FILD
FLDENV	FIMUL
FNOP	FIST/FISTP[1]
FRSTOR	FISUB/FISUBR
FSAVE/FNSAVE	FLD (single and double)
FST/FSTP (register-to-register)	FMUL/FMULP
FSTP (extended format to memory)	FPATAN
FSTCW/FNSTCW	FPREM/FPREM1
FSTENV/FNSTENV	FPTAN
FSTSW/FNSTSW	FRNDINT
WAIT/FWAIT	FSCALE
FXAM	FSIN
FXCH	FSINCOS
	FSQRT
	FST/FSTP (single and double)
	FSUB/FSUBP/FSUBR/FSUBRP
	FTST
	FUCOM/FUCOMP/FUCOMPP
	FXTRACT
	FYL2X/FYL2XP1

注 :

1. SSE3 の FISTTP 命令は、算術 x87 FPU 命令である。

8.5. x87 FPU 浮動小数点例外条件

以降の各項では、x87 FPUによって、浮動小数点例外を発生させる各種の条件と、それらの例外が検出されたときのx87 FPUのマスク応答について説明する。それぞれの浮動小数点命令において通知可能な浮動小数点例外の一覧については、『IA-32 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、中巻 A』の第3章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第4章「命令セット・リファレンス N-Z」を参照のこと。

1 つの命令に対して複数の浮動小数点例外条件が検出された場合の例外の優先規則については、4.9.2. 項「浮動小数点例外の優先順位」 を参照のこと。

8.5.1. 無効操作例外

浮動小数点の無効操作例外は、次の2つのサブクラスの操作に対する応答として発生する。

- スタック・オーバーフロー またはスタック・アンダーフロー（#IS）
- 無効算術オペランド（#IA）

この例外（IE）に対するフラグはx87 FPU ステータス・ワードのビット0であり、そのマスクビット（IM）はx87 FPU制御ワードのビット0である。x87 FPU ステータス・ワードのスタック・フォルト・フラグ（SF）は、この例外の原因となった操作のタイプを示す。SF フラグが1にセットされている場合は、スタック操作によってスタックのオーバーフローまたはアンダーフローが生じたことを示す。また、このフラグが0にクリアされている場合は、算術命令に無効なオペランドがあったことを示す。x87 FPUは、スタックのオーバーフローまたはアンダーフロー条件を検出した場合にSF フラグを明示的にセットするが、無効算術オペランド条件を検出した場合にはこのフラグを明示的にはクリアしないので注意しなければならない。つまり、最後にスタックのオーバーフローまたはアンダーフロー条件が発生したときにフラグをクリアしなかった場合は、SF フラグのステートは無効算術演算例外の後でも1である可能性がある。SF フラグの詳細については、8.1.2.4. 項「スタック・フォルト・フラグ」を参照のこと。

8.5.1.1. スタック・オーバーフロー例外またはスタック・アンダーフロー例外（#IS）

x87 FPU タグワードは、x87 FPU レジスタスタック内のレジスタの内容を記録し続ける（8.1.6. 項「x87 FPU タグワード」を参照）。タグワードは、この情報を使用して、次の2つの異なるタイプのスタックフォルトを検出する。

- スタック・オーバーフロー - 命令が、空でない x87 FPU レジスタにメモリ内の値をロードしようとした。空でないレジスタは、0（タグ値 01）、valid の値（タグ値 00）、または特殊な値（タグ値 10）を保持するレジスタとして定義される。
- スタック・アンダーフロー - 命令が、空の x87 FPU レジスタをソース・オペランドとして参照した（空のレジスタの内容をメモリに書き込もうとする操作など）。空のレジスタのタグ値は 11 である。

注記

スタック・オーバーフローという用語は、プログラムがメモリから x87 FPU レジスタスタックに 8 つの値をすでにロード（プッシュ）しているため、次の値をスタックにプッシュすると、すでに値が入っているレジスタに対するスタックのラップアラウンドが発生する状況に由来する。スタック・アンダーフローという用語は、これとは逆の状況に由来する。この場合は、プログラムが x87 FPU レジスタスタックからメモリに 8 つの値をすでにストア（ポップ）しているため、次の値をスタックからポップすると、空のレジスタに対するスタックのラップアラウンドが発生する。

スタックのオーバーフローまたはアンダーフローを検出すると、x87 FPU は x87 FPU ステータス・ワードの IE フラグ（ビット 0）と SF フラグ（ビット 6）を 1 にセットする。次に、スタック・オーバーフローが発生した場合は x87 FPU ステータス・ワードの条件コードフラグ C1（ビット 9）を 1 にセットし、スタック・アンダーフローが発生した場合は 0 にクリアする。

無効操作例外がマスクされている場合は、x87 FPU は次に、実行中の命令によって、倍精度浮動小数点値、整数値、またはパック形式 10 進整数の不定値をデスティネーション・オペランドに返す。命令で指定されているデスティネーション・レジスタまたはメモリ・ロケーションは、この値によって上書きされる。

無効操作例外がマスクされていない場合は、ソフトウェア例外ハンドラが呼び出される（8.7. 節「ソフトウェア内での x87 FPU 例外の処理」を参照）が、スタック・トップ・ポインタ（TOP）とソース・オペランドはそのまま変わらない。

8.5.1.2. 無効算術オペランド例外（#IA）

x87 FPU は、プログラムにコード化される各種の無効算術演算を検出できる。これらの演算を表 8-10. に示す（このリストには、IEEE 規格 754 に定義された無効操作が含まれる）。

無効算術オペランドを検出すると、x87 FPU は x87 FPU ステータス・ワードの IE フラグ（ビット 0）を 1 にセットする。無効操作例外がマスクされている場合は、表 8-10.

にしたがって、x87 FPU は不定値または QNaN をデスティネーション・オペランドに返すか、または浮動小数点条件コードを設定する。無効操作例外がマスクされていない場合は、ソフトウェア例外ハンドラが呼び出される（8.7. 節「ソフトウェア内での x87 FPU 例外の処理」を参照）が、スタック・トップ・ポインタ（TOP）とソース・オペランドはそのまま変わらない。

表 8-10. 無効算術演算とそれらに対するマスク応答

条件	マスク応答
サポートされていない形式のオペランドに対する算術演算。	QNaN 浮動小数点不定値をデスティネーション・オペランドに返す。
SNaN に対する算術演算。	QNaN をデスティネーション・オペランドに返す。（表 4-8. を参照）
順序比較およびテスト操作 : 一方または両方のオペランドが NaN。	x87 FPU ステータス・ワード、または EFLAGS レジスタの CF、PF、ZF フラグの条件コードフラグ（C0、C2、C3）を 111B（比較不可能）にセットする。
加算 : 両オペランドが反対符号の無限大。 減算 : 両オペランドが同じ符号の無限大。	QNaN 浮動小数点不定値をデスティネーション・オペランドに返す。
乗算 : ∞ × 0; 0 × ∞。	QNaN 浮動小数点不定値をデスティネーション・オペランドに返す。
除算 : ∞ × ∞; 0 × 0。	QNaN 浮動小数点不定値をデスティネーション・オペランドに返す。
剰余命令 FPREM, FPREM1: 法（除数）が 0 または被除数が ∞。	QNaN 浮動小数点不定値を返し、条件コードフラグ C2 を 0 にクリアする。
三角関数 FCOS, FPTAN, FSIN, FSINCOS: ソース・オペランドが ∞。	QNaN 浮動小数点不定値を返し、条件コードフラグ C2 を 0 にクリアする。
FSQRT: オペランドが負（FSQRT (–0) = –0 を除く）。 FYL2X: オペランドが負（FYL2X (–0) = –∞ を除く）。 FYL2XP1: オペランドが –1 より小。	QNaN 浮動小数点不定値をデスティネーション・オペランドに返す。
FBSTP: 変換された値が 18 ケタの 10 進数で表現できない、またはソース値が SNaN、QNaN、± ∞、サポートされていないフォーマット。	パックド BCD 整数不定値をデスティネーション・オペランドに格納する。
FIST/FISTP: 変換された値がデスティネーション・オペランドの表現可能な整数範囲を超えている、またはソース値が SNaN、QNaN、± ∞、サポートされていないフォーマット。	整数不定値をデスティネーション・オペランドに格納する。
FXCH: 一方または両方のレジスタが空としてタグ付けされている。	空のレジスタに QNaN 浮動小数点不定値をロードし、交換を実行する。

通常は、ソース・オペランドのいずれかまたは両方が QNaN である（いずれも SNaN やサポートされていないフォーマットではない）場合は、無効オペランド例外は生成されない。ただし、この規則は、比較命令の大半（FCOM 命令や FCOMI 命令など）や、浮動小数点から整数への変換命令（FIST/FISTP 命令および FBSTP 命令）には適用されない。これらの命令では、QNaN ソース・オペランドがあると、無効オペランド例外が生成される。

8.5.2. デノーマル・オペランド例外（#D）

x87 FPU は、次の条件のもとでデノーマル・オペランド例外を通知する。

- 算術命令がデノーマル・オペランドに対して演算を行おうとした場合（4.8.3.2. 項「ノーマル型有限数とデノーマル型有限数」を参照）。
- デノーマルの単精度か倍精度の浮動小数点値を x87 FPU レジスタにロードしようとした場合。（ロードされるデノーマル値が拡張倍精度浮動小数点値である場合は、デノーマル・オペランド例外は報告されない。）

この例外のフラグ（DE）は x87 FPU ステータス・ワードのビット 1 であり、そのマスクビット（DM）は x87 FPU 制御ワードのビット 1 である。

デノーマル・オペランド例外がマスクされている場合にこの例外が発生すると、x87 FPU は DE フラグをセットしてから命令の実行を再開する。単精度または倍精度の浮動小数点フォーマットのデノーマル・オペランドは、拡張倍精度浮動小数点フォーマットに変換されるときに自動的にノーマライズされる。実際、これ以降の演算では、内部の拡張倍精度浮動小数点フォーマットがもたらす追加精度によって、精度の高い結果が得られることが多い。

デノーマル・オペランド例外がマスクされていない場合にこの例外が発生すると、DE フラグがセットされ、ソフトウェア例外ハンドラが呼び出される（8.7. 節「ソフトウェア内での x87 FPU 例外の処理」を参照）。スタック・トップ・ポインタ（TOP）とソース・オペランドはそのまま変わらない。

デノーマル操作例外についての詳細は、4.9.1.2. 項「デノーマル・オペランド例外（#D）」を参照のこと。

8.5.3. ゼロ除算例外（#Z）

命令が非ゼロの有限値オペランドを 0 で割ろうとすると、x87 FPU は常に浮動小数点ゼロ除算例外を報告する。この例外のフラグ（ZE）は x87 FPU ステータス・ワードのビット 2 であり、そのマスクビット（ZM）は x87 FPU 制御ワードのビット 2 である。ゼロ除算例外を報告できるのは、FDIV、FDIVP、FDIVR、FDIVRP、FIDIV、FIDIVR の各命令と、内部的に除算を実行するその他の命令（FYL2X と FXTRACT）である。

ゼロ除算例外がマスクされている場合にこの例外が発生すると、x87 FPU は ZE フラグをセットし、表 8-10. に示されている値を返す。ゼロ除算例外がマスクされていない場合は、ZE フラグがセットされ、ソフトウェア例外ハンドラが呼出される（8.7. 節「ソフトウェア内での x87 FPU 例外の処理」参照）。スタック・トップ・ポインタ（TOP）とソース・オペランドはそのまま変わらない。

表 8-11. ゼロ除算条件とそれらに対するマスク応答

条件	マスク応答
0 の除数による除算または逆除算。	2 つのオペランドの符号の排他論理和を符号とする ∞ をデスティネーション・オペランドに返す。
FYL2X 命令。	非ゼロ・オペランドの反対の符号を持つ ∞ をデスティネーション・オペランドに返す。
FXTRACT 命令。	ST(1) が –∞ にセットされ、ST(0) がソース・オペランドと同じ符号を持つ 0 にセットされる。

8.5.4. 数値オーバーフロー例外（#O）

x87 FPU は、算術命令で丸められた結果がデスティネーション・オペランドの浮動小数点フォーマットの範囲内に収まらず、許容可能な最大有限値を超えた場合は、常に浮動小数点数値オーバーフロー例外（#O）を報告する（数値オーバーフロー例外についての詳細は、4.9.1.4. 項「数値オーバーフロー例外（#O）」を参照）。

x87 FPU を使用すると、数値オーバーフローは、結果が FPU データレジスタに格納される算術演算で発生する場合がある。また、データレジスタに格納された範囲内の値が単精度または倍精度の浮動小数点フォーマットでメモリに格納されるような（FST 命令や FSTP 命令による）浮動小数点ストア操作でも発生する可能性がある。値を整数フォーマットか BCD 整数フォーマットで格納する際にオーバーフローが発生するときは、数値オーバーフロー例外とはならない。この場合は、無効算術オペランド例外が通知される。

数値オーバーフロー例外に対するフラグ（OE）は、x87 FPU ステータス・ワードのビット 3 であり、そのマスクビット（OM）は x87 FPU 制御ワードのビット 3 である。

数値オーバーフロー例外が発生し、しかもこの例外がマスクされている場合は、x87 FPU は OE フラグをセットし、表 4-11. に示すいずれかの値を返す。返される値は、x87 FPU の現在の丸めモード（8.1.4.3. 項「丸め制御フィールド」を参照）によって異なる。

数値オーバーフローが発生し、しかも数値オーバーフロー例外がマスクされていない場合は、命令の結果がメモリとレジスタスタックのいずれに格納されるかによって、x87 FPU が行う処理は異なる。

- **デスティネーションがメモリ・ロケーションの場合。** OE フラグがセットされ、ソフトウェア例外ハンドラが呼び出される（8.7. 節「ソフトウェア内での x87 FPU 例外の処理」を参照）。スタック・トップ・ポインタ（TOP）とソース・オペランドはそのまま変わらない。スタック内のデータは拡張倍精度フォーマットであるため、例外ハンドラは、オペランドの適切な調整の後にストア命令を再実行するか、IEEE 規格の要件にしたがってスタック上の仮数部をデスティネーションの精度に合わせて丸めるかを選択できる。プログラムを続行する場合は、例外ハンドラは、結局はメモリ内のデスティネーション位置に値をストアする必要がある。

- **デスティネーションがレジスタスタックの場合。**結果の仮数部は x87 FPU 制御ワードの精度ビットと丸め制御ビットの現在の設定にしたがって丸められ、結果の指数部は 2^{24576} で割ることによって調整される（精度フィールドの影響を受けない命令の場合は、仮数部は拡張倍精度に丸められる）。得られた値は、デスティネーション・オペランドに格納される。仮数が切り上げ方向に丸められる場合は、x87 FPU ステータス・ワードの条件コードビット C1（この状況では「切り上げビット」と呼ばれる）がセットされ、結果が 0 方向に丸められる場合はクリアされる。結果が格納された後、OE フラグがセットされ、ソフトウェア例外ハンドラが呼び出される。スケーリング・バイアス値の 24,576 は $3 * 2^{13}$ に等しい。指数が 24,576 でバイアスされると、通常、数値は可能な限り拡張倍精度浮動小数点の指数範囲の中央に近い値に変換される。したがって、必要に応じて以降のスケーリング操作に使用すれば、例外を発生させる危険性を減らすことができる。

  FSCALE 命令を使用しているときに、結果が大き過ぎてバイアス調整型の指数を使用しても表現できない場合には、大きなオーバーフローが発生することがある。結果をバイアスした後に再びオーバーフローが発生した場合は、適切な符号を持つ∞がデスティネーション・オペランドに格納される。

8.5.5. 数値アンダーフロー例外（#U）

算術命令の丸められた結果が極小である場合、すなわち、デスティネーション・オペランドの浮動小数点フォーマットに収まる最小のノーマル型有限値より小さい場合は、x87 FPU は浮動小数点数値アンダーフロー条件（#U）を検出する（数値アンダーフロー例外についての詳細は、4.9.1.5. 項「数値アンダーフロー例外（#U）」を参照）。

数値オーバーフローの場合と同じように、数値アンダーフローは結果が x87 FPU データレジスタに格納される算術演算で発生する可能性がある。また、データレジスタの範囲内の値がより小さな単精度または倍精度の浮動小数点フォーマットでメモリに格納されるような（FST 命令と FSTP 命令による）浮動小数点ストア操作でも発生することがある。数値アンダーフロー例外は、数値を整数フォーマットや BCD 整数フォーマットで格納する際は発生することはない。極小値は、常に有効な丸めモードにしたがって、0 または 1 の整数値に丸められる。

数値アンダーフロー例外のフラグ（UE）は x87 FPU ステータス・ワードのビット 4 であり、そのマスクビット（UM）は x87 FPU 制御ワードのビット 4 である。

数値アンダーフロー例外が発生したとき、この例外がマスクされている場合は、x87 FPU は、4.9.1.5. 項「数値アンダーフロー例外（#U）」で説明した操作を実行する。

例外がマスクされていなかった場合には、命令の結果がメモリと x87 FPU レジスタスタックのいずれに格納されるかによって、x87 FPU が行う処理は異なる。

- **デスティネーションがメモリ・ロケーションの場合。（ストア命令でのみ発生する。）**UE フラグがセットされ、ソフトウェア例外ハンドラが呼び出される（8.7. 節「ソフトウェア内での x87 FPU 例外の処理」を参照）。スタック・トップ・ポインタ（TOP）とソース・オペランドはそのまま変わらない。トップ・オブ・スタック・ポインタ（TOP）、ソース・オペランド、デスティネーション・オペランドは変更されず、結果はメモリにストアされない。
- スタック内のデータは拡張倍精度フォーマットであるため、例外ハンドラは、オペランドの適切な調整の後にストア命令を再交換するか、IEEE 規格の要件にしたがってスタック上の仮数部をデスティネーションの精度に合わせて丸めるかを選択できる。プログラムを続行する場合は、例外ハンドラは、結局はメモリ内のデスティネーション位置に値をストアする必要がある。
- **デスティネーションがレジスタスタックの場合。**結果の仮数部は x87 FPU 制御ワードの精度ビットと丸め制御ビットの現在の設定にしたがって丸められ、結果の指数部は 2^{24576} を掛けることによって調整される（精度フィールドの影響を受けない命令の場合は、仮数部は拡張倍精度に丸められる）。得られた値は、デスティネーション・オペランドに格納される。仮数が切り上げ方向に丸められた場合は、x87 FPU ステータス・ワードの条件コードビット C1（この場合は「切り上げビット」の働きをする）がセットされ、結果が 0 方向に丸められた場合はクリアされる。結果がストアされた後、UE フラグがセットされ、ソフトウェア例外ハンドラが呼び出される。スケーリング・バイアス値 24,576 は、オーバーフロー例外に使用される値と同じであり、働きも同じである。つまり、結果を可能な限り拡張倍精度浮動小数点の指数範囲の中央に近い値に変換する。

  FSCALE 命令を使用しているときに、結果が小さ過ぎてバイアス調整型の指数を使用しても表現できない場合には、大きなアンダーフローが発生することがある。結果をバイアスした後に再びアンダーフローが発生した場合は、適切な符号を持つ 0 がデスティネーション・オペランドに格納される。

8.5.6. 不正確結果（精度）例外（#P）

不正確結果例外（精度例外とも呼ばれる）は、演算の結果がデスティネーション・フォーマットで正確に表現できない場合に発生する（不正確結果例外についての詳細は、4.9.1.6. 項「不正確結果（精度）例外（#P）」を参照）。ただし、超越関数命令（FSIN、FCOS、FSINCOS、FPTAN、FPATAN、F2XM1、FYL2X、FYL2XP1）は、性質上、不正確な結果を生じるので注意する。

不正確結果例外フラグ（PE）は x87 FPU ステータス・ワードのビット 5 であり、そのマスクビット（PM）は x87 FPU 制御ワードのビット 5 である。

不正確結果条件が発生したときに不正確結果例外がマスクされており、しかも数値オーバーフローまたはアンダーフローのいずれの条件も発生していない場合は、x87

FPU は、4.9.1.6. 項「不正確結果（精度）例外（#P)」の説明にしたがってこの例外を処理するが、以下の処理が追加される。x87 FPU ステータス・ワードの C1（切り上げ）ビットは、不正確結果が切り上げられた（C1=1）か、切り上げられなかった（C1=0）かを示す。切り上げられなかった場合（C1 がクリアされる）では、丸められた結果がデスティネーション・フォーマットに収まるよう、不正確結果の最下位ビットが切り捨てられる。

不正確結果が発生したときに不正確結果例外がマスクされておらず、しかも数値オーバーフローまたはアンダーフローのいずれの条件も発生していない場合は、x87 FPU は、前に述べた説明にしたがってこの例外を処理し、さらにソフトウェアの例外の処理を実行する。

数値オーバーフローまたは数値アンダーフローと同時に不正確結果例外が発生した場合は、x87 FPU は、以下の操作の 1 つを実行する。

- 不正確結果が、マスクされた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、OE または UE フラグと PE フラグがセットされ、オーバーフローまたはアンダーフロー例外の場合と同じ方法で結果が格納される（8.5.4. 項「数値オーバーフロー例外（#O)」または 8.5.5. 項「数値アンダーフロー例外（#U)」を参照)。不正確結果例外がマスクされていない場合は、x87 FPU はソフトウェア例外ハンドラを起動する。
- 不正確結果が、マスクされていない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・オペランドがレジスタである場合は、OE または UE フラグと PE フラグがセットされ、オーバーフローまたはアンダーフロー例外の場合と同じ方法で結果が格納され（8.5.4. 項「数値オーバーフロー例外（#O)」または 8.5.5. 項「数値アンダーフロー例外（#U)」を参照)、ソフトウェア例外ハンドラが起動される。

マスクされていない数値オーバーフロー例外またはアンダーフロー例外が発生し、デスティネーション・オペランドがメモリ・ロケーションである場合は（これは浮動小数点のストアの場合に限られる)、不正確結果条件は報告されず、C1 フラグがクリアされる。

8.6. x87 FPU 例外の同期

整数ユニットと x87 FPU は別々の命令実行ユニットであるため、プロセッサは浮動小数点命令、整数命令、システム命令を同時に並列して実行できる。この並列実行の利点を活用するのに、特殊なプログラミング手法は不要である。(浮動小数点命令は、整数命令やシステム命令と一緒に命令ストリームに配置される。) ただし、並列実行では浮動小数点例外ハンドラの介入を必要とする問題を生じることがある。

この問題は、マスクされていない浮動小数点例外の有無を x87 FPU が通知する方法に関わるものである。(x87 FPU はマスクされている結果を必ずデスティネーション・オペランドに返すため、マスクされている浮動小数点例外に対しては特別な例外同期の方法は不要である。)

浮動小数点例外がマスクされておらず、しかもその例外条件が発生した場合は、x87 FPU はこれ以降の浮動小数点命令の実行を停止し、例外イベントを通知する。命令ストリーム内に次の浮動小数点命令または WAIT/FWAIT 命令が現れると、プロセッサは x87 FPU ステータス・ワードの ES フラグを調べ、ペンディング状態の浮動小数点例外の有無を確認する。浮動小数点例外がペンディングされている場合は、x87 FPU は浮動小数点ソフトウェア例外ハンドラを暗黙的にコール（トラップ）する。この後、一部、またはすべての浮動小数点例外に対して、例外ハンドラは回復プロシージャを実行できる。

例外が通知されてから、その例外が実際に処理されるまでのタイムフレーム内で同期上の問題が発生する。並列実行のために、このタイムフレーム内で複数の整数命令またはシステム命令が実行される可能性がある。したがって、フォルトを生じた浮動小数点命令のソース・オペランドまたはデスティネーション・オペランドがメモリ上に上書きされ、例外ハンドラが例外を解析したり、例外から回復できなくなる可能性がある。

この問題を解決するために、特定の浮動小数点例外に関係するステート情報が失われたり破壊されるような状況を生じる可能性のある任意の浮動小数点命令の直後に、例外同期用の命令（浮動小数点命令または WAIT/FWAIT 命令）を配置する。データをメモリに格納するような浮動小数点命令では、まず同期をとる必要がある。例えば、次の 3 行のコードには例外同期上の問題が生じる可能性がある。

```
FILD COUNT ; FPU 命令
INC COUNT  ; 整数命令
FSQRT      ; 次の FPU 命令
```

この例では、INC 命令は、浮動小数点命令 FILD のソース・オペランドを変更する。FILD 命令の実行中に例外が報告された場合は、浮動小数点例外ハンドラが呼び出される前に、INC 命令が COUNT メモリ・ロケーションにストアされた値を上書きする可

能性がある。COUNT 変数が変更されてしまうと、浮動小数点例外ハンドラはエラーから回復できなくなる。

命令の順序を次のように変更し、FSQRT 命令を FILD 命令の後に置けば、浮動小数点例外処理の同期が得られ、浮動小数点例外ハンドラが起動される前に COUNT 変数が上書きされる可能性はなくなる。

```
FILD COUNT ; FPU 命令
FSQRT      ; 次の FPU 命令で FILD 命令で
           ; 生成された例外と同期をとる。
INC COUNT  ; 整数命令
```

FSQRT 命令の結果は x87 FPU データレジスタに格納され、次の浮動小数点命令または WAIT/FWAIT 命令が実行されるまでは上書きされずにそこに保持されるため、FSQRT 命令で同期をとる必要はない。FSQRT 命令によって生じたすべての例外を、例えばプロシージャ呼び出しの前に確実に処理するためには、FSQRT 命令のすぐ後に WAIT 命令を配置すればよい。

一部の浮動小数点命令（非同期型命令）は、ペンディング状態のマスクされていない例外の有無を確認しないので注意しなければならない（8.3.11. 項「x87 FPU 制御命令」を参照）。これらの命令には、FNINIT、FNSTENV、FNSAVE、FNSTSW、FNSTCW、FNCLEX などの命令が含まれる。FNINIT、FNSTENV、FNSAVE、FNCLEX のいずれかの命令が実行されると、ペンディング状態の例外すべてが実質的に失われる（x87 FPU ステータス・レジスタがクリアされるか、すべての例外がマスクされる）。これに対し、FNSTSW 命令と FNSTCW 命令では、ペンディング状態の割り込みの有無は確認されないが、x87 FPU ステータス・レジスタや制御レジスタも変更されない。したがって、その後に「同期型」浮動小数点命令を配置すれば、すべてのペンディング状態の例外を処理できる。

8.7. ソフトウェア内での x87 FPU 例外の処理

インテル® Pentium® プロセッサおよび IA-32 以降の各プロセッサの x87 FPU には、浮動小数点例外に対するソフトウェア例外ハンドラを呼び出すために、ネイティブ・モードと MS-DOS* 互換モードの 2 つの操作モードが用意されている。これらの操作モードは、制御レジスタ CR0 の NE フラグで選択する。（NE フラグの詳細については、『IA-32 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、下巻』の第 2 章「システム・アーキテクチャの概要」を参照のこと。）

8.7.1. ネイティブ・モード

浮動小数点例外の処理用にネイティブ・モードを選択するには、制御レジスタ CR0 の NE フラグを 1 にセットする。このモードでは、浮動小数点命令を実行中で、しかも例

外がマスクされていない（その例外に対するマスクビットがクリアされている）場合にx87 FPUが例外条件を検出すると、x87 FPUはまずその例外に対するフラグと、x87 FPU ステータス・ワード内の ES フラグをセットする。次に、浮動小数点エラー例外（#MF、ベクタ 16）を介してソフトウェア例外ハンドラを呼び出し、その直後にプロセッサの命令ストリーム内上にある次の命令のいずれかを実行する。

- ストリーム上の次の浮動小数点命令。ただし、それが非同期型命令（FNINIT、FNCLEX、FNSTSW、FNSTCW、FNSTENV、FNSAVE）のいずれかである場合を除く。
- ストリーム上の次の WAIT/FWAIT 命令。
- ストリーム上の次の MMX 命令。

命令ストリーム上の次の浮動小数点命令が非同期型命令である場合は、x87 FPU はソフトウェア例外ハンドラを呼び出さずに命令を実行する。

8.7.2. MS-DOS* 互換モード

制御レジスタ CR0 の NE フラグが 0 に設定されている場合は、浮動小数点例外処理にMS-DOS* 互換モードが選択される。このモードでは、浮動小数点例外に対するソフトウェア例外ハンドラが、プロセッサの FERR#、INTR、IGNNE# の各ピンを使用して外部的に呼び出される。浮動小数点エラーの報告と例外ハンドラの呼び出しにこの方法が用意されているのは、MS-DOS や Windows* 95 オペレーティング・システムが動作しているPCシステム上の浮動小数点例外処理機構をサポートするためである。

MS-DOS 互換モードでは、一般的に、次の方法を使用して浮動小数点例外ハンドラが呼び出される。

1. マスクされていない浮動小数点例外を検出すると、x87 FPU は例外に対するフラグをセットし、また x87 FPU ステータス・ワード内の ES フラグをセットする。
2. IGNNE# ピンがディアサートされている場合、x87 FPU は FERR# ピンを直ちにアサートするか、次に待機している浮動小数点命令または MMX 命令の実行直前までアサートするのを待つ。FERR# ピンを直ちにアサートするか遅らせるかは、プロセッサ、命令、例外のタイプによって決まる。
3. 直前の浮動小数点命令が、マスクされていない x87 FPU 例外に対して例外フラグをすでにセットしていた場合は、プロセッサは**次**の WAIT 命令の実行直前にフリーズ（停止）し、浮動小数点命令または MMX 命令を待つ。FERR# ピンが直前の浮動小数点命令の時点でアサートされていたか、現時点でアサートされているかにかかわらず、プロセッサがフリーズすることによって、新しい浮動小数点（または MMX）命令が実行される前に x87 FPU 例外ハンドラを確実に呼び出すことができる。

4. FERR# ピンが、カスケードされたプログラマブル割り込みコントローラ（PIC）の IRQ13 に外部ハードウェアを介して接続される。PIC は、FERR# ピンがアサートされると割り込み 75H を生成するようプログラムされている。
5. PIC がプロセッサ上の INTR ピンをアサートし、割り込み 75H を通知する。
6. PC システム用の BIOS が、割り込み 2（NMI）割り込みハンドラに分岐することで割り込み 75H を処理する。
7. 割り込み 2 ハンドラが、割り込みが NMI 割り込みの結果または浮動小数点例外の結果のいずれであるかを判断する。
8. 浮動小数点例外が検出された場合は、割り込み 2 ハンドラが浮動小数点例外ハンドラに分岐する。

IGNNE# ピンがアサートされている場合は、プロセッサは浮動小数点エラー条件を無視する。このピンが用意されているのは、浮動小数点例外ハンドラが以前に通知された浮動小数点例外を処理している間に、別の浮動小数点例外が生成されるのを防ぐためである。

MS-DOS 互換モードについては、付録 D「x87 FPU 例外ハンドラを作成する際のガイドライン」で詳しく説明している。付録 D の説明からも分かるように、このモードは Intel486™ プロセッサやインテル® Pentium® プロセッサに使用されている機構と比較すると幾分複雑になる。

8.7.3. ソフトウェア内での x87 FPU 例外の処理

4.9.3. 項「浮動小数点例外ハンドラの一般的な動作」は、浮動小数点例外ハンドラによって実行される処置を示している。x87 FPU のステートは、FSTENV/FNSTENV 命令または FSAVE/FNSAVE 命令によって保存される（8.1.9. 項「FSTENV/FNSTENV 命令および FSAVE/FNSAVE 命令による x87 FPU のステートのセーブ」を参照）。

フォルトを生じた浮動小数点命令の後に非浮動小数点命令が 1 つ以上ある場合は、フォルトを発生した命令を実行し直しても無駄な場合がある。浮動小数点例外を同期させる方法については、8.6. 節「x87 FPU 例外の同期」を参照のこと。

ハンドラがフォルトを発生した命令からプログラムの実行を再開する必要がある場合は、IRET 命令を直接使用することはできない。その理由は、フォルトを発生した浮動小数点命令の後に続く浮動小数点命令または WAIT/FWAIT 命令までは例外が発生しないため、スタック上のリターン命令ポインタがフォルトを発生した命令をポイントしていない可能性があるためである。フォルトを発生した命令からプログラムの実行を再開するには、例外ハンドラはセーブされている x87 FPU ステート情報からその命令のポインタを入手し、それをスタック上のリターン命令ポインタ位置にロードし、その後で IRET 命令を実行しなければならない。

浮動小数点例外ハンドラの一般的な例や、MS-DOS* 互換モードを使用している場合に浮動小数点例外ハンドラを作成する方法を示した特殊な例については、D.3.4. 項「x87 FPU 例外ハンドラの例」を参照のこと。

9

インテル® MMX® テクノロジによるプログラミング

第 9 章
インテル® MMX® テクノロジによるプログラミング

9

インテル® MMX® テクノロジは、インテル® Pentium® II プロセッサ・ファミリおよび MMX® テクノロジ Pentium プロセッサで IA-32 アーキテクチャに導入された。MMX テクノロジで導入された拡張機能は、高度なメディアおよび通信アプリケーションの処理を高速化する SIMD（Single Instruction、Multiple Data）実行モデルをサポートする。

本章では、MMX テクノロジについて説明する。

9.1. MMX® テクノロジのプログラミング環境の概要

MMX® テクノロジは、64 ビット・パックド整数データを処理する、簡単で柔軟性の高い SIMD 実行モデルを定義している。このモデルは、IA-32 アーキテクチャに以下の機能を追加するが、すべての IA-32 アプリケーションおよびオペレーティング・システム・コードとの下方互換性を維持している。

- 8 つの新しい 64 ビット・データ・レジスタ（MMX テクノロジ・レジスタ）
- 3 つの新しいパックドデータ型
  - 64 ビット・パックド・バイト整数（符号付き / 符号なし）
  - 64 ビット・パックド・ワード整数（符号付き / 符号なし）
  - 64 ビット・パックド・ダブルワード整数（符号付き / 符号なし）
- 新しいデータ型をサポートし、MMX テクノロジ・ステートを管理する命令
- CPUID 命令の拡張

MMX テクノロジには、IA-32 アーキテクチャのすべての実行モード（プロテクト・モード、実アドレスモード、仮想 8086 モード）からアクセスできる。MMX テクノロジは、IA-32 アーキテクチャに新しい実行モードを追加しない。

本章では、MMX テクノロジ・レジスタ・セット、MMX テクノロジ・データ型、MMX 命令セットなど、MMX テクノロジの基本的なプログラミング環境について説明する。SSE および SSE2 では、MMX テクノロジ・レジスタを操作する新しい命令が、IA-32 アーキテクチャに追加された。

詳細については、以下の箇所を参照のこと。

- 10.4.4. 項「SSE 64 ビット SIMD 整数命令」では、SSE で IA-32 アーキテクチャに追加された MMX 命令について説明する。
- 11.4.2. 項「SSE2 64 ビットおよび 128 ビット SIMD 整数命令」では、SSE2 で IA-32 アーキテクチャに追加された MMX 命令について説明する。
- 『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」では、各 MMX 命令について詳しく説明する。
- 『IA-32 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、下巻』の第 11 章「インテル® MMX® テクノロジ・システム・プログラミング・モデル」では、MMX テクノロジが IA-32 システム・プログラミング・モデルにどのように統合されているかについて説明する。

9.2. MMX® テクノロジのプログラミング環境

図 9-1. は、MMX® テクノロジの実行環境を示している。すべての MMX 命令は、次のように、MMX テクノロジ・レジスタ、汎用レジスタ、およびメモリを操作する。

- **MMX テクノロジ・レジスタ。**8 つの MMX テクノロジ・レジスタ（図 9-1. を参照）を使用して、64 ビット・パックド整数データの操作を実行する。これらのレジスタには、MM0 ～ MM7 の名前が付いている。

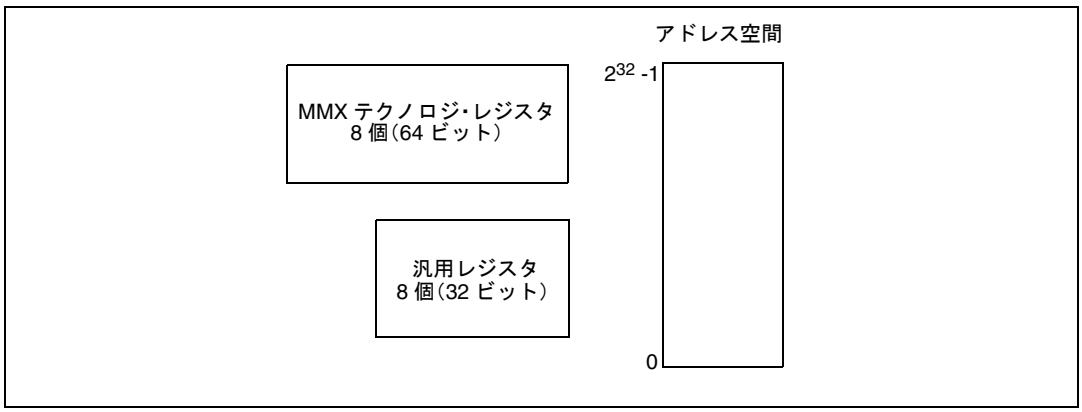


図 9-1. MMX® テクノロジの実行環境

- **汎用レジスタ。**8 つの汎用レジスタ（図 3-4. を参照）と既存の IA-32 アドレス指定モードを組み合わせて、メモリ内のオペランドをアドレス指定する（MMX テクノロジ・レジスタは、メモリのアドレス指定には使用できない）。汎用レジスタは、

いくつかのMMXテクノロジ命令ではオペランドの格納にも使用される。これらのレジスタは、EAX、EBX、ECX、EDX、EBP、ESI、EDI、ESPという名前で参照される。

9.2.1. MMX® テクノロジ・レジスタ

MMX® テクノロジ・レジスタ・セットは、8つの64ビット・レジスタ（図9-2.を参照）で構成される。これらのレジスタを使用して、MMXテクノロジ・パックド整数データ型の計算を実行できる。MMXテクノロジ・レジスタ内の値は、メモリ内の64ビットと同じフォーマットを持つ。

MMXテクノロジ・レジスタには、64ビット・アクセス・モードと32ビット・アクセス・モードの2種類のデータ・アクセス・モードがある。

64ビット・アクセス・モードは以下の目的に使用される。

- 64ビット・メモリ・アクセス
- MMXテクノロジ・レジスタ間の64ビット転送
- すべてのパック命令、論理演算命令、算術命令
- いくつかのアンパック命令

32ビット・アクセス・モードは以下の目的に使用される。

- 32ビット・メモリ・アクセス
- 汎用レジスタとMMXテクノロジ・レジスタの間の32ビット転送
- いくつかのアンパック命令

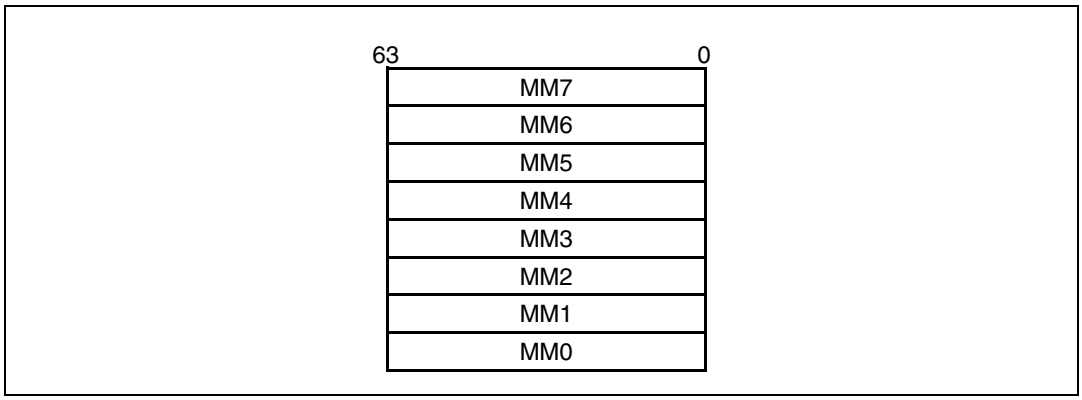


図 9-2. MMX® テクノロジ・レジスタ・セット

IA-32 アーキテクチャでは MMX テクノロジ・レジスタを独立したレジスタとして定義しているが、実際には FPU データ・レジスタ・スタック（R0 ～ R7）のレジスタに別名を付けて使用している。

9.5. 節「x87 FPU アーキテクチャとの互換性」も参照のこと。

9.2.2. MMX® テクノロジ・データ型

MMX® テクノロジでは、以下の 64 ビット・データ型が IA-32 アーキテクチャに追加された（図 9-3. を参照）。

- 64 ビット・パックド・バイト整数 － 8 つのパックドバイト
- 64 ビット・パックド・ワード整数 － 4 つのパックドワード
- 64 ビット・パックド・ダブルワード整数 － 2 つのパックド・ダブルワード

MMX 命令は、64 ビット・パックド・データ型（パックドバイト、パックドワード、またはパックド・ダブルワード）とクワッドワード・データ型を、MMX テクノロジ・レジスタとメモリの間または MMX テクノロジ・レジスタ同士の間で 64 ビット・ブロックで転送する。ただし、MMX 命令は、パックドデータ型の算術演算または論理演算を実行する場合、MMX テクノロジ・レジスタ内の個々のバイト、ワード、またはダブルワードを並列に処理する（9.2.4. 項「SIMD（single-instruction, multiple-data）実行モデル」を参照）

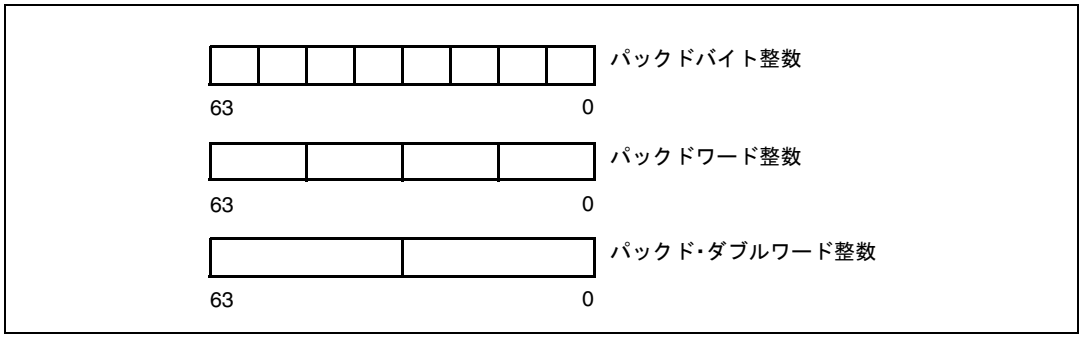


図 9-3. MMX® テクノロジで導入されたデータ型

9.2.3. メモリ内のデータ・フォーマット

メモリに格納するとき、パックドデータ型のバイト、ワード、ダブルワードが連続したアドレスに格納される。最下位のバイト、ワード、ダブルワードが連続するアドレス領域の最下位アドレスに格納され、最上位のバイト、ワード、ダブルワードが、上位アドレスに格納される。バイト、ワード、ダブルワードのメモリへの格納順は常にリトル・エンディアン型であり、下位アドレスにはデータの下位バイトが入り、上位アドレスには上位バイトが入る。

9.2.4. SIMD（single-instruction, multiple-data）実行モデル

MMX® テクノロジでは、単一命令、複数データ（SIMD）技法を使用して、64 ビットの MMX テクノロジ・レジスタにパックされているバイト、ワード、ダブルワードに対する算術演算および論理演算を行う（図 9-4. 参照）。例えば、PADDSW 命令は、第 1 のソース・オペランド内の 4 つの符号付きワード整数を、第 2 のソース・オペランド内の 4 つの符号付きワード整数に加算し、得られた 4 つのワード整数をデスティネーション・オペランドに格納する。このように SIMD 技法では複数のデータ要素に対して同一演算を並列に行うので、ソフトウェアの処理能力を向上させることができる。MMX テクノロジでは、バイト、ワード、ダブルワードのデータ要素が MMX テクノロジ・レジスタに入っている場合にのみ並列演算をサポートしている。

MMX テクノロジがサポートしている SIMD 実行モデルがターゲットとしているのは、最近のメディア、通信、グラフィック関係のアプリケーションである。このようなアプリケーションでは、サイズの小さいデータ型（バイト、ワード、ダブルワード）に対して同一演算を膨大な回数実行するような高度なアルゴリズムを使用している。例えば、ほとんどのオーディオ・データは 16 ビット（ワード）単位で表現されているので、MMX 命令を使用すれば、1 つの命令で 4 つのワードデータの演算を同時に実行できる。また、ビデオやグラフィックのデータは 8 ビット（バイト）単位のパレットで表現されている場合が多い。図 9-4. では MMX 命令を使用して、8 つのバイトデータを同時に演算する。

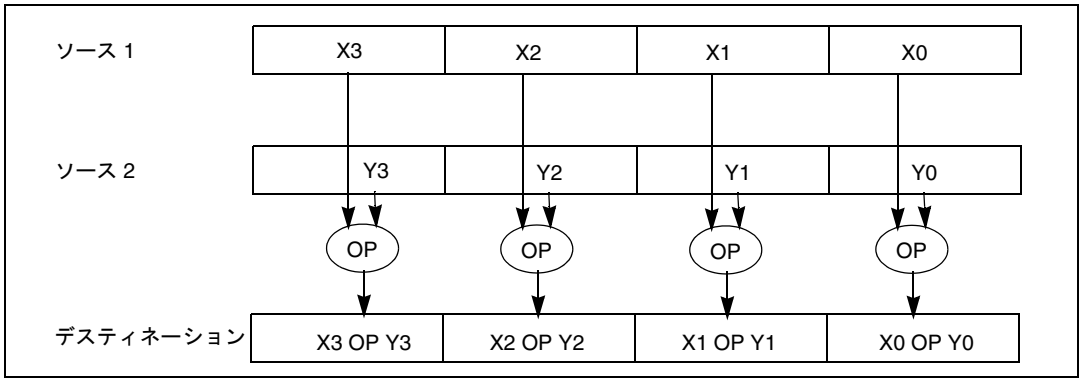


図 9-4. SIMD 実行モデル

9.3. 飽和算術とラップアラウンド・モード

整数算術演算を実行したとき、演算結果が範囲外状態になることがある。演算結果が範囲外の場合、真の結果をデスティネーション・フォーマットで表すことができない。例えば、符号付きワード整数の算術演算の実行時に、正のオーバーフローが発生し、真の符号付き結果が 16 ビットより大きくなることがある。

MMX® テクノロジは、以下の 3 つの方法で範囲外状態を処理する。

- **ラップアラウンド算術**。ラップアラウンド算術では、範囲外の真の結果は切り捨てられる（すなわち、キャリービットまたはオーバーフロー・ビットは無視され、演算結果の下位ビットだけがデスティネーションに返される）。ラップアラウンド算術は、オペランドの範囲を制御して範囲外の結果を防ぐアプリケーションに適している。しかし、オペランドの範囲が制御されない場合は、ラップアラウンド算術によって大きな誤差が生じるときがある。例えば、2 つの大きな符号付き数を加算すると、正のオーバーフローが発生し、負の結果が得られることがある。
- **符号付き飽和算術**。符号付き飽和算術では、範囲外の結果は、操作対象となる整数サイズで表現できる符号付き整数の範囲に合わせて制限される（表 9-1. を参照）。例えば、符号付きワード整数を操作したときに正のオーバーフローが発生した場合は、結果は 7FFFH（16 ビットで表現できる最大の正の整数）に「飽和」される。負のオーバーフローが発生した場合は、結果は 8000H に飽和される。
- **符号なし飽和算術**。符号なし飽和算術では、範囲外の結果は、操作対象となる整数サイズで表現できる符号なし整数の範囲に合わせて制限される。したがって、符号なしバイト整数を操作したときに正のオーバーフローが発生した場合は、FFH が返される。負のオーバーフローが発生した場合は、00H が返される。

表 9-1. 飽和算術演算でのデータ範囲の限界値

データ型	下限値		上限値	
	16 進	10 進	16 進	10 進
符号付きバイト	80H	-128	7FH	127
符号付きワード	8000H	-32,768	7FFFH	32,767
符号なしバイト	00H	0	FFH	255
符号なしワード	0000H	0	FFFFH	65,535

飽和算術は、多くのオーバーフロー状態で自然な解が得られる。例えば、カラー計算で飽和処理を使用すれば、色の反転が起こらないため、色は純粋な黒または純粋な白に保たれる。また、ソース・オペランドの範囲チェックを使用しない場合に、ラップアラウンドによる問題が計算に影響を与えることを防止できる。

MMX 命令では、例外を発生させたり EFLAGS レジスタ内のフラグをセットすることによってオーバーフローやアンダーフローを通知しない。

9.4. MMX® 命令

MMX® 命令セットを構成する 57 個の命令は、次のカテゴリに分類できる。

- データ転送命令
- 算術命令
- 比較命令
- 変換命令
- アンパック命令
- 論理命令
- シフト命令
- MMX テクノロジ・ステート・クリア命令（EMMS）

表 9-2. は、MMX 命令セットの要約を示している。以下の各項では、各グループの命令について簡単に説明する。

注記

本章で説明したMMX命令は、CPUID MMXテクノロジ機能ビット（ビット23）がセットされている場合にIA-32プロセッサ上で使用できる命令である。10.4.4. 項「SSE 64ビットSIMD整数命令」と11.4.2. 項「SSE2 64ビットおよび128ビットSIMD整数命令」では、MMXテクノロジ・レジスタを操作するにもかかわらずMMX命令セットの一部とは見なされない、SSEとSSE2の追加命令について説明する。

表 9-2. MMX® 命令セットのまとめ

カテゴリ		ラップアラウンド	符号付き飽和演算	符号なし飽和演算
算術	加算	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	減算	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	乗算	PMULL, PMULH		
	乗算および加算	PMADD		
比較	一致比較	PCMPEQB, PCMPEQW, PCMPEQD		
	より大きい比較	PCMPGTPB, PCMPGTPW, PCMPGTPD		
変換	パック		PACKSSWB, PACKSSDW	PACKUSWB
アンパック	上位部のアンパック	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	下位部のアンパック	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		

		パックドデータ	クワッドワード全体
論理	AND AND NOT OR XOR		PAND PANDN POR PXOR
シフト	左に論理シフト 右に論理シフト 右に算術シフト	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD	PSLLQ PSRLQ

		ダブルワード転送	クワッドワード転送
データ転送	レジスタ間の転送 メモリからのロード メモリへのストア	MOVD MOVD MOVD	MOVQ MOVQ MOVQ
MMXテクノロジ・ステートのクリア		EMMS	

9.4.1. データ転送命令

MOVD（32 ビット移動）命令では、32 ビットのパックドデータを、メモリから MMX® テクノロジ・レジスタ（または、その反対方向）に移動するか、汎用レジスタから MMX テクノロジ・レジスタ（または、その反対方向）に移動する。

MOVQ（64 ビット移動）命令は、64 ビットのパックドデータを、メモリから MMX テクノロジ・レジスタ（または、その反対方向）に移動するか、MMX テクノロジ・レジスタ間で移動する。

9.4.2. 算術命令

算術命令は、パックドデータ型に対して加算、減算、乗算、乗算プラス加算の各演算を実行する。

PADDB/PADDW/PADDD（add packed integers）命令は、ラップアラウンド・モードを使用して、ソース・オペランドとデスティネーション・オペランドの対応する符号付きまたは符号なしのデータ要素を加算し、PSUBB/PSUBW/PSUBD（subtract packed integers）命令はそのデータ要素を減算する。これらの命令が動作するデータ型は、パックドバイト、パックドワード、パックド・ダブルワードである。

PADDSB/PADDSW（add packed signed integers with signed saturation）命令は、ソース・オペランドとデスティネーション・オペランドの対応する符号付きデータ要素を加算し、PSUBSB/PSUBSW（subtract packed signed integers with signed saturation）命令はそのデータ要素を減算する。計算結果は符号付きの各データ型の範囲内に飽和させる。これらの命令が動作するデータ型は、パックドバイトとパックドワードである。

PADDUSB/PADDUSW（add packed unsigned integers with unsigned saturation）命令は、ソース・オペランドとデスティネーション・オペランドの対応する符号なしデータ要素を加算し、PSUBUSB/PSUBUSW（subtract packed unsigned integers with unsigned saturation）命令はそのデータ要素を減算する。計算結果は符号なしの各データ型の範囲内に飽和させる。これらの命令が動作するデータ型は、パックドバイトとパックドワードである。

PMULHW（multiply packed signed integers and store high result）命令と PMULLW（multiply packed signed integers and store low result）命令は、ソースとデスティネーションの両オペランドに対応する符号付きワードを乗算する。PMULHW 命令では結果の上位 16 ビットを、PMULLW 命令では下位 16 ビットを、それぞれデスティネーション・オペランドに格納する。

PMADDWD（multiply and add packed integers）命令は、ソース・オペランドとデスティネーション・オペランドの対応する符号付きワードの積を計算する。4 つの 32 ビット・

ダブルワードの中間結果を 2 つずつ（上位のペアと下位のペアで）合計し、2 つの 32 ビット・ダブルワードの結果を求める。

9.4.3. 比較命令

PCMPEQB/PCMPEQW/PCMPEQD（compare packed data for equal）命令と PCMPGTB/PCMPGTW/PCMPGTD（compare packed signed integers for greater than）命令は、ソースとデスティネーションの両オペランドに入っている各符号つきデータ要素（バイト、ワード、またはダブルワード）を比較して、対応するもの同士の一致、または大小関係を判定する。

その結果生成された 1 と 0 からなるマスクデータがデスティネーション・オペランドに書き込まれる。論理演算でこのマスクを使用して符号付きデータ要素を選択できる。これを使用すれば、いくつもの分岐命令を使用しなくても、条件付きデータ移動を行える。EFLAGS レジスタ内のフラグは影響を受けない。

9.4.4. 変換命令

PACKSSWB（pack words into bytes with signed saturation）命令は、符号付き飽和処理を使用して、符号付きワードを符号付きバイトに変換する。PACKSSDW（pack doublewords into words with signed saturation）命令は、符号付き飽和処理を使用して、符号付きダブルワードを符号付きワードに変換する。

PACKUSWB（pack words into bytes with unsigned saturation）命令は、符号なし飽和処理を使用して、符号付きワードを符号なしバイトに変換する。

9.4.5. アンパック命令

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ（unpack high-order data elements）命令と、PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ（unpack low-order data elements）命令は、ソース・オペランドとデスティネーション・オペランドの上位または下位のデータ要素から、バイト、ワード、またはダブルワードをアンパックして、デスティネーション・オペランド内にインターリーブする。ソース・オペランドの要素をすべて 0 にすれば、これらの命令を使用して、バイト整数からワード整数への変換、ワード整数からダブルワード整数への変換、またはダブルワード整数からクワッドワード整数への変換を実行できる。

9.4.6. 論理命令

PAND（bitwise logical AND）、PANDN（bitwise logical AND NOT）、POR（bitwise logical OR）、PXOR（bitwise logical exclusive OR）命令は、クワッドワード・ソースとデスティネーションの両オペランドに対してビットごとの論理演算を行う。

9.4.7. シフト命令

論理左シフト、論理右シフト、算術右シフト命令は、各データ要素を指定のビット位置の数だけシフトする。論理左シフトと論理右シフトの命令では、64 ビット領域（クワッドワード）を 1 つのブロックとしてシフトもでき、データ型の変換とアライメント操作に有用である。

PSLLW/PSLLD/PSLLQ（shift packed data left logical）命令、PSRLW/PSRLD/PSRLQ（shift packed data right logical）命令は、データ要素の左または右への論理シフトを実行し、空になる上位ビットまたは下位ビットをゼロで埋める。これらのシフト命令が動作するのはパックドワード、パックド・ダブルワード、クワッドワードである。

PSRAW/PSRAD（shift packed data right arithmetic）命令は、右への算術シフトを実行し、各データ要素の上位側の空いたビット位置に、各データ要素の符号ビットをコピーする。これらのシフト命令が動作するのはパックドワードとパックド・ダブルワードである。

9.4.8. EMMS 命令

EMMS 命令は、x87 FPU タグワード内のタグを 11B（空のレジスタ）に設定することによって、MMX® テクノロジ・ステートをクリアする。MMX テクノロジ・ルーチンの終了時には、浮動小数点命令を実行する他のルーチンを呼び出す前に、この命令を実行しなければならない。この命令の使用法についての詳細は、9.6.3. 項「EMMS 命令の使用法」を参照のこと。

9.5. x87 FPU アーキテクチャとの互換性

MMX® テクノロジ・ステートとは x87 FPU ステートの別名で、MMX テクノロジのサポートする IA-32 アーキテクチャのために追加された新規ステートや新規モードではない。x87 FPU ステートのセーブとリストアを行う浮動小数点命令でも MMX テクノロジ・ステートを操作できる（例えば、コンテキスト・スイッチングの場合）。

MMX テクノロジでも、x87 FPU とオペレーティング・システムとのインターフェイス技法（主にタスク・スイッチングで使用）と同じ技法を使用する。詳細については、『IA-32 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、下

巻』の第 11 章「インテル® MMX® テクノロジ・システム・プログラミング・モデル」を参照のこと。

9.5.1. MMX® 命令と x87 FPU タグワードの関係

MMX® 命令の実行後は、必ず x87 FPU タグワード全体が Valid（00B）に設定されている。EMMS 命令（MMX テクノロジ・ステートのクリア命令）を実行すると、x87 FPU タグワード全体が Empty（11B）に設定される。

『IA-32 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、下巻』の第 11 章「インテル® MMX® テクノロジ・システム・プログラミング・モデル」では、x87 FPU 命令および MMX 命令と x87 FPU タグワードとの関係を詳しく説明している。浮動小数点タグワードの詳細については、8.1.6. 項「x87 FPU タグワード」を参照のこと。

9.6. MMX® テクノロジ・コードによるアプリケーションの作成

以下の各項では、MMX® テクノロジを使用してアプリケーション・コードを作成するときのガイドラインを示す。

9.6.1. MMX® テクノロジのサポートのチェック

アプリケーションは、MMX® テクノロジ命令を使用する前に、プロセッサが MMX テクノロジをサポートしているかどうかを確認する必要がある。以下の手順でこのチェックを実行する。

1. CPUID 命令を実行して、プロセッサが CPUID 命令をサポートしているかどうかチェックする。プロセッサが CPUID 命令をサポートしていない場合は、無効オペコード例外（#UD）が発生する。
2. CPUID 命令を使用して、MMX テクノロジ機能ビットをチェックし、プロセッサが MMX テクノロジをサポートしているかどうか確認する。EAX レジスタ内で引き数を 1 に設定して CPUID 命令を実行し、ビット 23（MMX テクノロジ）が 1 にセットされていることを確認する。
3. コントロール・レジスタ CR0 内の EM ビットが 0 に設定されているかどうかチェックする。これは、x87 FPU のエミュレーションが無効にされていることを示す。

プロセッサがサポートしていない MMX 命令を実行しようとしたり、コントロール・レジスタ CR0 の EM ビットが 1 に設定されているときに MMX 命令を実行しようとすると、無効オペコード例外（#UD）が発生する。

例 9-1. に、MMX テクノロジを検出する CPUID 命令の使用方法を示す。この例は CPUID 命令の詳細な使用手順を示すのではなく、MMX テクノロジ・サポートの検出のための大筋のみを示す。

例 9-1. CPUID 命令による MMX テクノロジ検出ルーチンの一部

```
...                                 ; CPUID 命令の存在を識別
...
...                                 ; インテル・プロセッサを識別
....
mov   EAX, 1                        ; 機能フラグを要求
CPUID                               ; 0Fh, 0A2h CPUID 命令
test  EDX, 00800000h                ; 機能フラグの IA MMX テクノロジ・ビット（EDX の
                                    ; ビット 23）がセットされているかどうかのテスト
jnz   MMX_Technology_Found
```

9.6.2. x87 FPU コードと MMX® テクノロジ・コードの間の移行

1 つのアプリケーション内で、x87 浮動小数点命令と MMX® 命令の両方を使用できる。ただし、MMX テクノロジ・レジスタは x87 FPU レジスタスタックに対して別名参照されるため、x87 FPU 命令と MMX 命令の間の移行を行う際は、コヒーレンシのない結果や予期しない結果が発生しないように、十分に注意する必要がある。

（EMMS 命令以外の）MMX 命令が実行されると、プロセッサは x87 FPU ステートを次のように変更する。

- x87 FPU ステータス・ワードの TOS（トップ・オブ・スタック）値を 0 に設定する。
- x87 FPU タグワード全体を valid の状態に（すべてのタグ・フィールドを 00B に）設定する。
- MMX 命令が MMX テクノロジ・レジスタへの書き込みを行うと、対応する浮動小数点レジスタの指数部（ビット 64 ～ 79）に 1（11B）が書き込まれる。

これらの処置の結果、MMX 命令の実行以前の x87 FPU ステートは、基本的にはすべて失われる。

x87 FPU 命令が実行されるとき、プロセッサは、x87 FPU レジスタスタックおよびコントロール・レジスタの現在の状態が valid であると見なし、x87 FPU ステートをあらかじめ変更することなく、x87 FPU 命令を実行する。

アプリケーション内で x87 FPU 浮動小数点命令と MMX 命令の両方を使用する場合は、以下のガイドラインに従うことをお勧めする。

- x87 FPU コードから MMX テクノロジ・コードに移行する際は、将来の使用に備えて保持しなければならない x87 FPU データレジスタまたはコントロール・レジスタ

の状態をすべて保存する。FSAVE 命令と FXSAVE 命令は、x87 FPU ステート全体を保存する。

- MMX テクノロジ・コードから x87 FPU コードに移行する際は、以下の動作を実行する。
  - 将来の使用に備えて保持しなければならない MMX テクノロジ・レジスタ内のデータをすべて保存する。FSAVE 命令と FXSAVE 命令は、MMX テクノロジ・レジスタの状態も保存する。
  - EMMS 命令を実行して、x87 データレジスタおよびコントロール・レジスタから、MMX テクノロジ・ステートをクリアする。

以下の各項では、EMMS 命令の使用方法と、x87 FPU コードと MMX テクノロジ・コードを混在させる場合のその他のガイドラインについて説明する。

9.6.3. EMMS 命令の使用法

9.6.2. 項「x87 FPU コードと MMX® テクノロジ・コードの間の移行」で説明したように、MMX® 命令が実行されると、x87 FPU タグワードは valid（00B）とマークされる。この状態で x87 FPU 命令を実行すると、x87 FPU レジスタスタックに有効なデータが入っていると見なされるため、予期しない x87 FPU 浮動小数点例外や誤った結果が発生する。EMMS 命令は、クリアされているものとして x87 FPU タグワードをマークすることによって、この問題を回避する。

次のいずれかの場合には、必ず EMMS 命令を実行しなければならない。

- x87 FPU 命令を使用しているアプリケーションから MMX テクノロジ・ライブラリ /DLL を呼び出す場合（MMX テクノロジ・コードの最後で EMMS 命令を実行する）。
- MMX 命令を使用しているアプリケーションから x87　FPU 浮動小数点ライブラリ /DLL を呼び出す場合（x87 FPU コードを呼び出す直前に EMMS 命令を実行する）。
- 非プリエンプティブ（協調型）オペレーティング・システムにおいて、あるタスク / スレッドの MMX テクノロジ・コードと他のタスク / スレッドの切り替えが行われる場合（ただし、x87 FPU コードより前に MMX 命令を実行することが明らかな場合は除く）。

MMX テクノロジ命令と、SSE、SSE2、SSE3 を混在させる場合は、EMMS 命令を使用する必要はない（11.6.7. 項「SSE および SSE2 と x87 FPU 命令および MMX® 命令の相互作用」を参照）。

9.6.4. MMX® 命令と x87 FPU 命令の混在

1 つのアプリケーション内で、x87 FPU 浮動小数点命令と MMX® 命令の両方を使用することができる。ただし、プロセッサによってはパフォーマンスが低下するため、MMX 命令と x87 FPU 命令を頻繁に切り替えることはお勧めできない。

MMX テクノロジ・コードと x87 FPU コードを混在させる場合は、以下のガイドラインに従うこと。

- MMX テクノロジ・コードと x87 FPU コードは、別々のモジュール、プロシージャ、またはルーチン内に置く。
- x87 FPU コード・モジュールと MMX テクノロジ・コード・モジュールの間の移行の前後で、レジスタの内容に依存しない。
- MMX テクノロジ・コードから x87 FPU コードに移行する際は、（将来 MMX テクノロジ・レジスタの状態が必要になる場合は）MMX テクノロジ・レジスタの状態を保存し、EMMS 命令を実行して MMX テクノロジ・ステートを空にする。
- x87 FPU コードから MMX テクノロジ・コードに移行する際は、（将来 x87 FPU ステートが必要になる場合は）x87 FPU ステートを保存する。

9.6.5. MMX® テクノロジ・コードのインターフェイス

MMX® テクノロジ命令を使用して、すべての MMX テクノロジ・レジスタに直接にアクセスできる。つまり、プロセッサの汎用レジスタ（EAX、EBX など）の使用時に適用されるすべての既存のインターフェイス規則は、MMX テクノロジ・レジスタの使用時にも適用される。

MMX テクノロジ・ルーチンへの効率的なインターフェイスは、MMX テクノロジ・レジスタを使用するか、または（スタックを介して）メモリ・ロケーションと MMX テクノロジ・レジスタを組み合わせて、パラメータと戻り値を渡す。MMX テクノロジ・レジスタを使用してパラメータを渡す場合は、EMMS 命令を使用したり、MMX テクノロジ・コードと x87 FPU コードを混在させてはならない。

MMX テクノロジ・データ型を直接サポートしない高水準言語を使用する場合は、パックドデータ型を保持する 64 ビット構造として、MMX テクノロジ・データ型を定義できる。

高水準言語で MMX 命令をコーディングする場合は、次のような他の手法を使用できる。

- スタックを介して構造へのポインタを渡すことによって、MMX テクノロジ・ルーチンにパラメータを渡す。
- 構造へのポインタを返すことによって、関数から値を返す。

9.6.6. マルチタスク・オペレーティング・システム環境での MMX® テクノロジ・コードの使用

アプリケーションは自分がどのようなマルチタスク・オペレーティング・システム上で実行されているかを知る必要がある。タスクスイッチが発生するときに各タスクはそのステートをセーブしておかなければならない。プロセッサ・ステート（コンテキスト）は、汎用レジスタおよび浮動小数点 /MMX® テクノロジ・レジスタで構成される。

オペレーティング・システムには、次の 2 種類がある。

- 非プリエンプティブ（協調的）マルチタスク・オペレーティング・システム
- プリエンプティブ・マルチタスク・オペレーティング・システム

非プリエンプティブ・マルチタスク・オペレーティング・システムでは、コンテキスト・スイッチの際に FPU ステート（または MMX テクノロジ・ステート）をセーブしない。そのため、直接あるいは間接に制御をオペレーティング・システムに返す前にアプリケーションが自分で必要なステートをセーブする必要がある。

プリエンプティブ・マルチタスク・オペレーティング・システムでは、コンテキスト・スイッチの際に FPU ステート（または MMX テクノロジ・ステート）のセーブとリストアを行う。そのため、アプリケーションが自分で FPU ステート（または MMX テクノロジ・ステート）のセーブやリストアを行う必要はない。

コンテキスト・スイッチ時の 2 つのタイプのオペレーティング・システムの動作については、『IA-32 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、下巻』の第 11 章の「コンテキスト・スイッチ」を参照のこと。

9.6.7. MMX® テクノロジ・コードでの例外処理

MMX® 命令で発生するメモリアクセス例外は他の IA-32 命令で発生する例外と同じで、ページフォルト、セグメント不在、境界違反などがある。MMX テクノロジ・コードでは、既存の例外ハンドラを修正せずに使って、これらのタイプの例外を処理できる。

ペンディング状態の浮動小数点例外がなければ、MMX 命令で数値例外が発生することはない。したがって、数値例外を処理するために、既存の例外ハンドラを修正したり、新しい例外ハンドラを追加する必要はない。

ペンディング状態の浮動小数点例外があるときに、MMX 命令を実行すると数値エラー例外（割込み 16 および / または FERR# ピンのアサート）が発生する。例外ハンドラからリターンすると、MMX 命令が実行を再開する。

9.6.8. レジスタのマッピング

MMX® テクノロジ・レジスタとそのタグは、浮動小数点レジスタとそのタグの物理的位置にマッピングされている。レジスタの別名とマッピングの詳細については、『IA-32 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、下巻』の第 11 章「インテル® MMX® テクノロジ・システム・プログラミング・モデル」を参照のこと。

9.6.9. MMX® 命令に対する命令プリフィックスの影響

表9-3.にMMX®命令に対する命令プリフィックスの影響を示す。予測不可能な動作は、ある世代の IA-32 プロセッサでは予約済みの動作として扱われ、他の世代のプロセッサでは無効オペコード例外を発生させることがある。

表 9-3. MMX® テクノロジ命令に対するプリフィックスの影響

プリフィックスのタイプ	MMX 命令に対する影響
アドレス・サイズ・プリフィックス（67H）	メモリ・オペランドを持つ命令に影響する。
	メモリ・オペランドを持たない命令では予約済みであり、予測不可能な動作を発生させる。
オペランド・サイズ（66H）	予約済みであり、予測不可能な動作を発生させる。
セグメント・オーバライド（2EH, 36H, 3EH, 26H, 64H, 65H）	メモリ・オペランドを持つ命令に影響する。
	メモリ・オペランドを持たない命令では予約済みであり、予測不可能な動作を発生させる。
リピート・プリフィックス（F3H）	予約済みであり、予測不可能な動作を発生させる。
リピート NE プリフィックス（F2H）	予約済みであり、予測不可能な動作を発生させる。
ロック・プリフィックス（0F0H）	予約済み。無効オペコード例外（#UD）が発生する。
分岐ヒント・プリフィックス（2EH、3EH）	予約済みであり、予測不可能な動作を発生させる。

命令プリフィックスの詳細については、『IA-32 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、中巻 A』の第 2 章の「命令プリフィックス」を参照のこと。

10

ストリーミング SIMD 拡張命令（SSE）による プログラミング

第 10 章
ストリーミング SIMD 拡張命令（SSE）によるプログラミング

10

ストリーミング SIMD 拡張命令（SSE）は、インテル® Pentium® III プロセッサ・ファミリで IA-32 アーキテクチャに導入された。これらの拡張命令によって、高度な 2D および 3D グラフィックス、モーション・ビデオ、画像処理、音声認識、音声合成、テレフォニ、ビデオ会議などのアプリケーションに対する、IA-32 プロセッサのパフォーマンスが強化される。

本章では、SSE について説明する。第 11 章「ストリーミング SIMD 拡張命令 2（SSE2）によるプログラミング」には、SSE と SSE2 を使用するアプリケーション・プログラムを作成する際に必要な内容が記載されている。第 12 章「ストリーミング SIMD 拡張命令 3（SSE3）によるプログラミング」には、SSE3 の詳細が記載されている。

10.1. SSE の概要

インテル® MMX® テクノロジによって、IA-32 アーキテクチャに SIMD（Single Instruction, Multiple Data）機能が導入された。この機能には、64 ビット MMX テクノロジ・レジスタ、64 ビット・パックド整数データ型、パックド整数に対して SIMD 演算を実行する命令が使用される。SSE は、MMX テクノロジの SIMD 実行モデルを拡張したものであり、128 ビット・レジスタ内のパックドおよびスカラ単精度浮動小数点値を処理するための機能が追加されている。

SSE は、IA-32 アーキテクチャに以下の機能を追加するが、すべての既存の IA-32 プロセッサ、アプリケーション、オペレーティング・システムとの下方互換性を維持している。

- 8 つの 128 ビット・データ・レジスタ（XMM レジスタ）
- 32 ビット MXCSR レジスタ。このレジスタは、XMM レジスタに対して実行される操作の制御ビットとステータス・ビットを格納する。
- 128 ビット・パックド単精度浮動小数点データ型（4 つの IEEE 単精度浮動小数点値を、1 つのダブル・クワッドワードにパックしたもの）
- 単精度浮動小数点値の SIMD 演算を実行する命令と、整数に対して実行される SIMD 演算を拡張する命令
  - XMM レジスタ内のデータを操作する、128 ビット・パックドおよびスカラ単精度浮動小数点命令

- 64 ビット SIMD 整数命令（MMX テクノロジ・レジスタ内のパックド整数オペランドに対する追加の操作をサポート）

- MXCSR レジスタの状態のセーブとリストアを実行する命令
- データの明示的なプリフェッチ、データのキャッシュ制御、およびストア操作のアクセス順序の制御をサポートする命令
- CPUID 命令の拡張

これらの機能によって、IA-32 アーキテクチャの SIMD プログラミング・モデルは、以下の 4 つの点で大きく強化される。

- 4 つのパックド単精度浮動小数点値の SIMD 演算を実行できるため、高い処理能力を必要とするアルゴリズムによって単純なネイティブ・データ要素の大きな配列の反復操作を実行する、高度なメディア・アプリケーションや通信アプリケーションに対して、IA-32 プロセッサのパフォーマンスが大きく向上する。
- XMM レジスタ内で SIMD 単精度浮動小数点演算を実行でき、MMX テクノロジ・レジスタ内で SIMD 整数演算を実行できるため、浮動小数点データと整数データの大きな配列を操作するアプリケーションを実行するための柔軟性とスループットが大きく向上する。
- キャッシュ制御命令によって、キャッシュを汚染することなく、XMM レジスタとの間でデータのストリーミングが行える。また、データが実際に使用される前に、選択したキャッシュ・レベルにそのデータをプリフェッチすることができる。これらのプリフェッチ機能とストリーミング・ストア機能によって、大量のデータに定期的にアクセスする必要があるアプリケーションのパフォーマンスが向上する。
- SFENCE（store fence）命令により、順序設定の緩いメモリタイプの使用時に、ストア操作のアクセス順序をきめ細かく制御できる。

SSE は、IA-32 プロセッサ用に作成されたすべてのソフトウェアとの完全な互換性を持つ。すべての既存のソフトウェアは、SSE を搭載したプロセッサ上で、修正なしで正常に動作し続ける。CPUID 命令の拡張によって、SSE をサポートするかどうかを検出できるようになった。SSE には、IA-32 アーキテクチャのすべての実行モード（プロテクト・モード、実アドレスモード、仮想 8086 モード）からアクセスできる。

本章では、XMM レジスタ、パックド単精度浮動小数点データ型、および SSE など、SSE のプログラミング環境について説明する。詳細については、以下の箇所を参照のこと。

- 11.6. 節「SSE および SSE2 によるアプリケーションの作成」
- 11.5. 節「SSE、SSE2、SSE3 の例外」では、SSE と SSE2 で生成される例外について説明する。

- 『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」では、SSE3 について詳しく説明する。
- 『IA-32 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、下巻』の第 12 章「SSE と SSE2 のシステム・プログラミング」では、SSE と SSE2 をオペレーティング・システム環境に統合する際のガイドラインについて説明する。

10.2. SSE のプログラミング環境

図 10-1. は、SSE の実行環境を示している。すべての SSE は、次のように、XMM レジスタ、MMX® テクノロジ・レジスタ、およびメモリを操作する。

- **XMM レジスタ。** 8 つの XMM レジスタ（図 10-2. と 10.2.1. 項「XMM レジスタ」を参照）を使用して、パックドまたはスカラ単精度浮動小数点データを操作する。スカラ演算とは、XMM レジスタの最下位ダブルワードに格納される、個々の（アンパックされた）単精度浮動小数点値に対して実行される演算である。XMM レジスタは、XMM0 ～ XMM7 の名前で参照される。

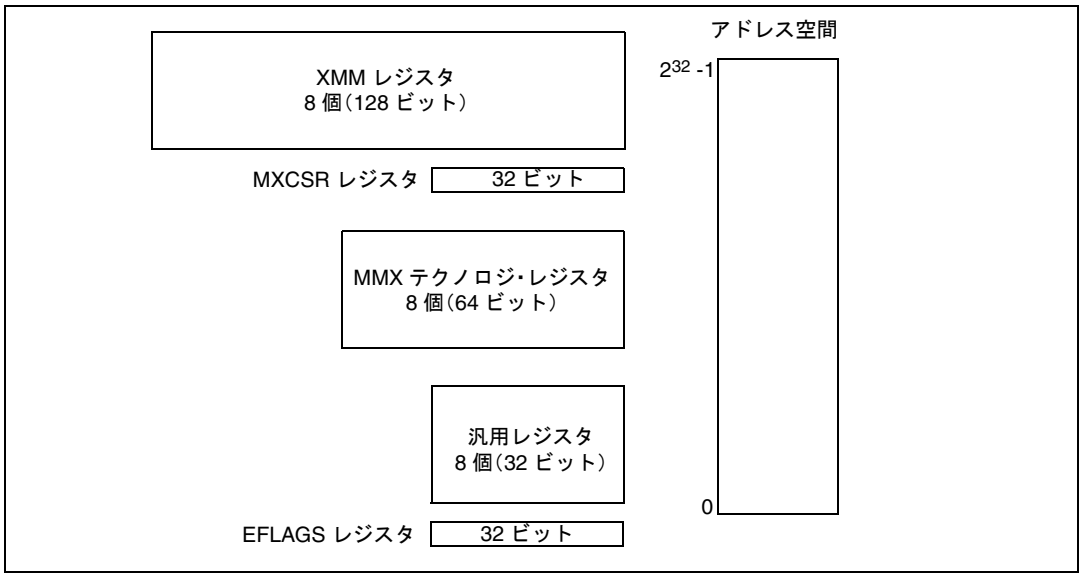


図 10-1. SSE の実行環境

- **MXCSR レジスタ。** この 32 ビット・レジスタ（図 10-3. と 10.2.2. 項「MXCSR 制御およびステータス・レジスタ」を参照）は、SIMD 浮動小数点演算に使用されるステータス・ビットと制御ビットを格納する。

- **MMX テクノロジ・レジスタ。**8 つの MMX テクノロジ・レジスタ（図 9-2. を参照）を使用して、64 ビット・パックド整数データの操作を実行する。MMX テクノロジ・レジスタと XMM レジスタの間で実行される操作では、MMX テクノロジ・レジスタがオペランドの格納にも使用される。MMX テクノロジ・レジスタは、MM0 ～ MM7 の名前で参照される。
- **汎用レジスタ。**8 つの汎用レジスタ（図 3-4. を参照）と既存の IA-32 アドレス指定モードを組み合わせて、メモリ内のオペランドをアドレス指定する（MMX テクノロジ・レジスタと XMM レジスタは、メモリのアドレス指定には使用できない）。一部の SSE では、汎用レジスタがオペランドの格納にも使用される。汎用レジスタは、EAX、EBX、ECX、EDX、EBP、ESI、EDI、ESP の名前で参照される。
- **EFLAGS レジスタ。**この 32 ビット・レジスタ（図 3-7. を参照）は、比較操作の結果を記録する。

10.2.1. XMM レジスタ

SSE では、8 つの 128 ビット XMM データレジスタが IA-32 アーキテクチャに追加された（図 10-2 を参照）。これらのレジスタには、レジスタ名 XMM0 ～ XMM7 で直接アクセスできる。また、これらのレジスタには、x87 FPU/MMX テクノロジ・レジスタおよび汎用レジスタとは無関係にアクセスできる（つまり、XMM レジスタは、IA-32 プロセッサの他のレジスタ用に別名で定義されることはない）。

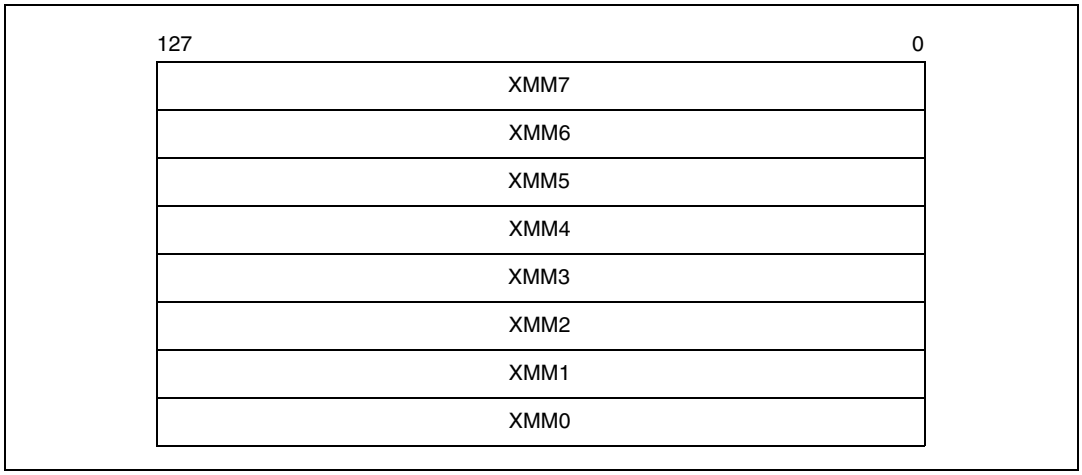


図 10-2. XMM レジスタ

SSE は、XMM レジスタを、パックド単精度浮動小数点オペランドの操作専用に使用する。SSE2 では、XMM レジスタの機能が拡張され、パックドまたはスカラ倍精度浮動小数点オペランドとパックド整数オペランドを操作できるようになった（11.2. 節

「SSE2 のプログラミング環境」と 12.2. 節「SSE3 のプログラミング環境とデータ型」を参照）。

XMM レジスタは、データの計算専用に使用される。XMM レジスタは、メモリのアドレス指定には使用できない。アドレス指定メモリは、汎用レジスタを使用して行われる。

データは、32 ビット、64 ビット、128 ビット単位で XMM レジスタにロードされ、XMM レジスタからメモリに書き込まれる。XMM レジスタの全体の内容をメモリ（128 ビット・ストア）にストアする際は、データは連続する 16 バイトで格納され、レジスタの最下位バイトがメモリの最初のバイトに格納される。

10.2.2. MXCSR 制御およびステータス・レジスタ

新しい 32 ビットの MXCSR レジスタ（図 10-3. を参照）は、SSE、SSE2、SSE3 の操作の制御情報とステータス情報を格納する。このレジスタには、次のものが入る。

- SIMD 浮動小数点例外のフラグビットとマスクビット
- SIMD 浮動小数点演算の丸め制御ビット
- SIMD 浮動小数点演算のアンダーフロー条件を制御するゼロ・フラッシュ・フラグ
- SIMD 浮動小数点命令のデノーマル・ソース・オペランドの処理を制御するデノーマル・ゼロ・フラグ

このレジスタの内容をメモリからロードするときは、LDMXCSR 命令または FXRSTOR 命令を使用する。このレジスタの内容をメモリにストアするときは、STMXCSR 命令または FXSAVE 命令を使用する。

MXCSR レジスタのビット 16 ～ 31 は予約済みであり、プロセッサの電源投入時またはリセット時にクリアされる。FXRSTOR 命令または LDMXCSR 命令を使用して、これらのビットにゼロでない値を書き込もうとすると、一般保護例外（#GP）が発生する。

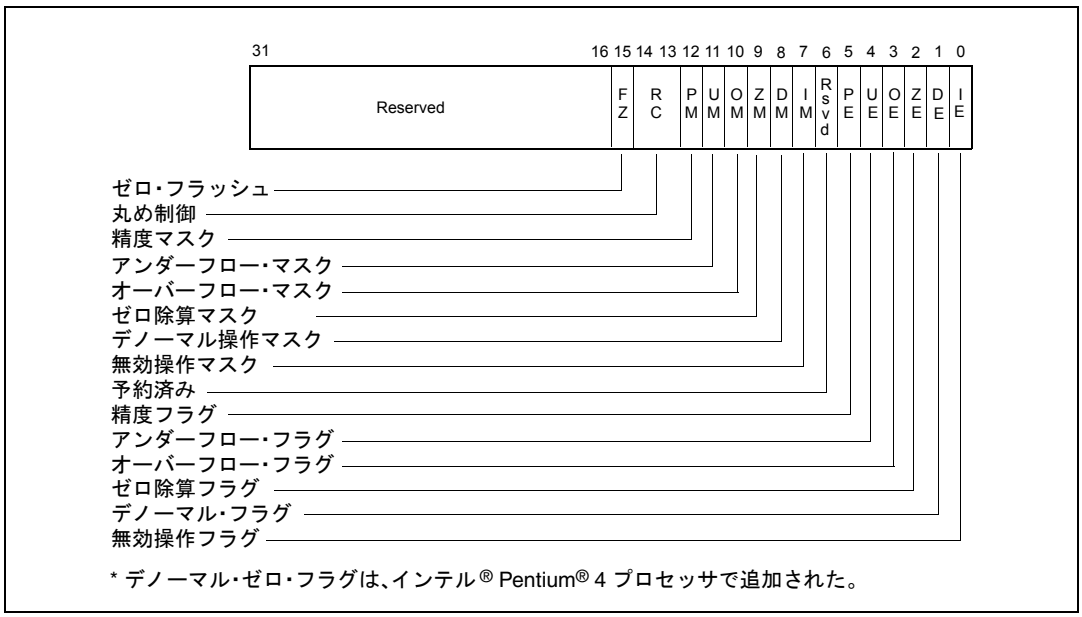


図 10-3. MXCSR 制御 / ステータス・レジスタ

10.2.2.1. SIMD 浮動小数点マスクビットおよびフラグビット

MXCSR レジスタのビット 0 ～ 5 は、SIMD 浮動小数点数値例外が検出されたかどうかを示す。これらのビットは「スティッキー・フラグ」であり、一度セットされると、明示的にクリアされるまではセットされたままになる。これらのフラグをクリアするには、LDMXCSR 命令または FXRSTOR 命令を使用して各フラグにゼロを書き込む必要がある。

ビット 7 ～ 12 は、SIMD 浮動小数点例外の個々のマスクビットを格納する。ある例外タイプに対応するマスクビットがセットされると、その例外はマスクされる。対応するマスクビットがクリアされると、その例外はアンマスクされる。電源投入時またはリセット時には、これらのマスクビットはセットされる。つまり、初期状態では、すべての SIMD 浮動小数点例外がマスクされる。

LDMXCSR 命令または FXRSTOR 命令によってマスクビットがクリアされ、対応する例外フラグビットがセットされても、この変更によって例外が生成されるわけではない。アンマスクされた例外は、その後に実行される SSE、SSE2 または SSE3 がその例外条件を検出したときに、初めて生成される。

SIMD 浮動小数点例外のマスクビットとフラグビットの使用法については、11.5. 節「SSE、SSE2、SSE3 の例外」と 12.4. 節「SSE3 の例外」を参照のこと。

10.2.2.2. SIMD 浮動小数点丸め制御フィールド

MXCSR レジスタのビット 13 とビット 14（丸め制御 [RC] フィールド）は、SIMD 浮動小数点命令の結果を丸める方法を制御する。丸め制御ビットの機能とエンコーディングについては、4.8.4. 項「丸め」を参照のこと。

10.2.2.3. ゼロ・フラッシュ

MXCSR レジスタのビット 15（FZ）は、ゼロ・フラッシュ・モードを有効にする。このモードは、SIMD 浮動小数点アンダーフロー条件に対するマスク応答を制御する。アンダーフロー例外がマスクされ、ゼロ・フラッシュ・モードが有効になっている場合、プロセッサは、浮動小数点アンダーフロー条件を検出したとき、以下の処理を実行する。

- 真の結果の符号を使用してゼロの結果を返す。
- 精度例外フラグとアンダーフロー例外フラグをセットする。

アンダーフロー例外がマスクされていない場合は、ゼロ・フラッシュ・ビットは無視される。

ゼロ・フラッシュ・モードは、IEEE 規格 754 に適合していない。IEEE の規定では、アンダーフローに対するマスク応答は、デノーマライズされた結果を返すことである（4.8.3.2. 項「ノーマル型有限数とデノーマル型有限数」を参照）。ゼロ・フラッシュ・モードは、主にパフォーマンス上の理由で用意されている。アンダーフローが頻繁に発生するアプリケーションで、アンダーフロー結果をゼロに丸めても問題がない場合は、精度が多少低下する代わりに、実行速度のアップを実現できる。

プロセッサの電源投入時またはリセット時には、ゼロ・フラッシュ・ビットはクリアされ、ゼロ・フラッシュ・モードは無効になる。

10.2.2.4. デノーマル・ゼロ

MXCSR レジスタのビット 6（DAZ）は、デノーマル・ゼロ・モードを有効にする。このモードは、SIMD 浮動小数点デノーマル・オペランド状態に対するプロセッサの応答を制御する。デノーマル・ゼロ・フラグがセットされている場合、プロセッサは、すべてのデノーマル・ソース・オペランドを元のオペランドと同じ符号の 0 に変換してから、それらのオペランドの計算を実行する。プロセッサは、デノーマル・オペランド例外マスクビット（DM）の設定には関係なく、デノーマル・オペランド例外フラグ（DE）をセットしない。また、デノーマル・オペランド例外がマスクされていない場合、デノーマル・オペランド例外を生成しない。

デノーマル・ゼロ・モードは、IEEE 規格 754 に適合していない（4.8.3.2. 項「ノーマル型有限数とデノーマル型有限数」を参照）。デノーマル・ゼロ・モードは、デノーマ

ル・オペランドを 0 に丸めても処理されるデータの品質にあまり影響を与えない、ストリーミング・メディア処理などのアプリケーションの実行時のプロセッサのパフォーマンスを改善するために用意されている。

プロセッサの電源投入時とリセット時には、デノーマル・ゼロ・フラグはクリアされ、デノーマル・ゼロ・モードは無効になる。

デノーマル・ゼロ・モードは、SSE2 でインテル® Pentium® 4 プロセッサとインテル® Xeon™ プロセッサに追加された。ただし、このモードは、SSE の SIMD 浮動小数点命令と完全な互換性がある（すなわち、デノーマル・ゼロ・フラグは、SSE の SIMD 浮動小数点命令の動作にも影響を与える）。以前の IA-32 プロセッサとインテル Pentium 4 プロセッサの一部のモデルでは、このフラグ（ビット 6）は予約済みになっている。この機能のサポートの有無を検出する命令については、11.6.3. 項「MXCSR レジスタの DAZ フラグのチェック」を参照のこと。

DAZ フラグをサポートしていないプロセッサ上で、MXCSR レジスタのビット 6 をセットしようとすると、一般保護例外（#GP）が発生する。FXSAVE 命令で返される MXCSR_MASK 値を使用してこのような一般保護例外を防ぐ方法については、11.6.6. 項「MXCSR レジスタへの書き込みのガイドライン」を参照のこと。

10.2.3. SSE、SSE2、SSE3、MMX® テクノロジ、x87 FPU のプログラミング環境の互換性

SSE で IA-32 実行環境に導入されたステート（XMM レジスタと MXCSR レジスタ）は、SSE2 と SSE3 でも共有される。SSE、SSE2、SSE3 は、完全な互換性を持つ。これらの命令は、同じ命令ストリーム内で実行できる。命令セットの切り替え時にステートを保存する必要はない。

XMM レジスタは、x87 FPU レジスタおよび MMX® テクノロジ・レジスタに依存しない。したがって、XMM レジスタに対して実行される SSE、SSE2、SSE3 の操作は、x87 FPU および MMX テクノロジ・レジスタと並行して実行することができる（11.6.7. 項「SSE および SSE2 と x87 FPU 命令および MMX® 命令の相互作用」を参照）。

FXSAVE 命令と FXRSTOR 命令は、SSE、SSE2、SSE3 のステートを、x87 FPU と MMX テクノロジのステートと一緒にセーブおよびリストアする。

10.3. SSE のデータ型

SSE では、128 ビット・パック ド単精度浮動小数点データ型が IA-32 アーキテクチャに追加された（図 10-4. を参照）。このデータ型は、4 つの IEEE 32 ビット単精度浮動小数点値を 1 つのダブル・クワッドワードにパックしたものである（単精度浮動小数点値のレイアウトについては、図 4-3. を参照。単精度浮動小数点フォーマットについての詳細は、4.2.2. 項「浮動小数点データ型」を参照）。

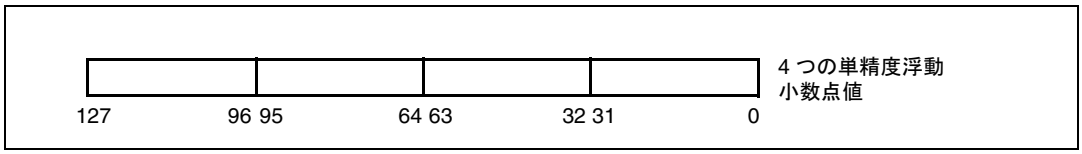


図 10-4. 128 ビット・パック ド単精度浮動小数点データ型

128 ビット・パック ド単精度浮動小数点データ型は、XMM レジスタまたはメモリ内で操作される。変換命令を使用して、2 つのパック ド単精度浮動小数点値を 2 つのパック ド・ダブルワード整数に変換したり、スカラ単精度浮動小数点値をダブルワード整数に変換できる（図 11-8. を参照）。

SSE は、XMM レジスタと MMX® テクノロジ・レジスタの間の変換命令と、XMM レジスタと汎用ビットレジスタの間の変換命令を用意している。図 11-8. を参照のこと。

128 ビット・パック ド・メモリ・オペランドのアドレスは、以下の場合を除き、16 バイトにアライメントが合っていなければならない。

- MOVUPS 命令は、アライメントの合っていないデータへのアクセスをサポートしている。
- スカラ命令が、アライメントの必要条件に従わない 4 バイト・メモリ・オペランドを使用する場合。

図 4-2. は、メモリ内の 128 ビット（ダブル・クワッドワード）データ型のバイト・オーダを示している。

10.4. SSE セット

SSEは、以下の4つの機能グループに分類される。

- パックドおよびスカラ単精度浮動小数点命令
- 64ビットSIMD整数命令
- ステート管理命令
- キャッシュ制御命令、プリフェッチ命令、メモリアクセス順序命令

SSE CPUID機能ビット（EDXレジスタのビット25）は、このIA-32プロセッサがSSEをサポートしているかどうかを示す。

以下の各項では、各グループの命令の概要を説明する。

10.4.1. SSE パックドおよびスカラ浮動小数点命令

パックドおよびスカラ単精度浮動小数点命令は、以下のグループに分類される。

- データ転送命令
- 算術演算命令
- 論理演算命令
- 比較命令
- シャッフル命令
- 変換命令

パックド単精度浮動小数点命令は、パックド単精度浮動小数点オペランドでSIMDを操作する（図10-5.を参照）。各ソース・オペランドには、4つの単精度浮動小数点値が格納される。デスティネーション・オペランドには、各オペランド内の対応する値（X0とY0、X1とY1、X2とY2、X3とY3）に対して並行して実行された操作（OP）の結果が格納される。

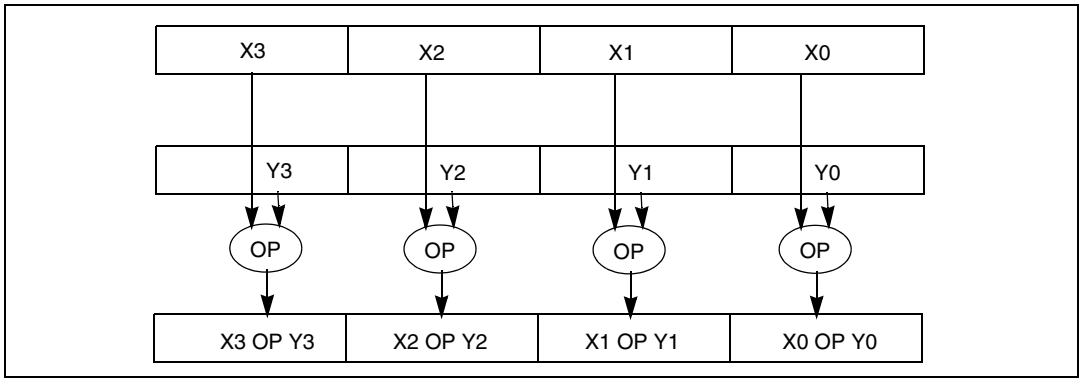


図 10-5. パックド単精度浮動小数点の操作

スカラ単精度浮動小数点命令は、2つのソース・オペランド（X0とY0）の最下位ダブルワードを操作する。図10-6.を参照のこと。第1のソース・オペランドの上位3つのダブルワード（X1、X2、X3）は、デスティネーション・オペランドにそのまま渡される。このスカラ操作は、x87 FPUデータレジスタ内で実行される浮動小数点操作によく似ている。

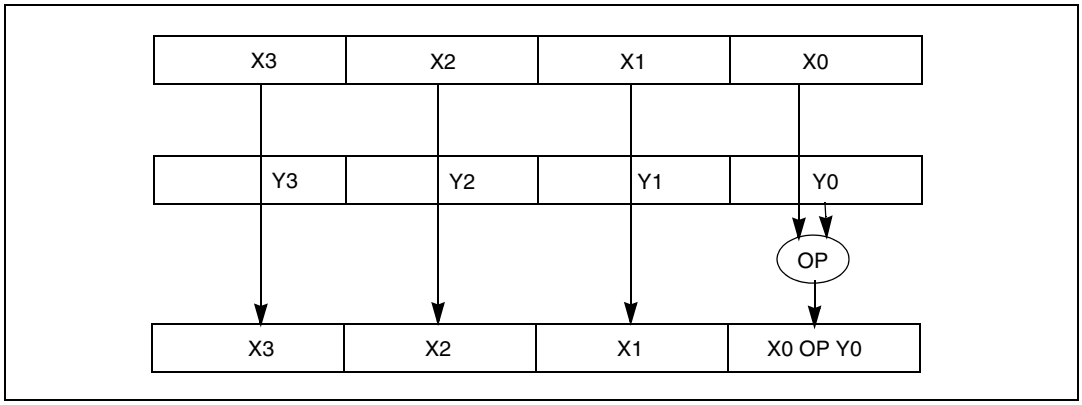


図 10-6. スカラ単精度浮動小数点の操作

10.4.1.1. SSE データ転送命令

SSEデータ転送命令は、XMMレジスタ同士の間およびXMMレジスタとメモリの間で、単精度浮動小数点データを転送する。

MOVAPS（move aligned packed single-precision floating-point values）命令は、4つのパックド単精度浮動小数点値ダブル・クワットワード・オペランドを、メモリからXMMレジスタに（または、その反対方向に）転送するか、XMMレジスタ同士の間で転送する。メモリアドレスは、16バイトにアライメントが合っていなければならない。アライメントが合っていないと、一般保護例外（#GP）が発生する。

MOVUPS（move unaligned packed single-precision, floating-point）命令は、MOVAPS 命令と同じ操作を実行するが、メモリアドレスの 16 バイト・アライメントが要求されない点が異なる。

MOVSS（move scalar single-precision floating-point）命令は、32 ビット単精度浮動小数点オペランドを、メモリから XMM レジスタの最下位ダブルワードに（または、その反対方向に）転送するか、XMM レジスタ同士の間で転送する。

MOVLPS（move low packed single-precision floating-point）命令は、2 つのパックド単精度浮動小数点値を、メモリから XMM レジスタの下位クワッドワードに（または、その反対方向に）転送する。XMM レジスタの上位クワッドワードはそのまま残される。

MOVHPS（move high packed single-precision floating-point）命令は、2 つのパックド単精度浮動小数点値を、メモリから XMM レジスタの上位クワッドワードに（または、その反対方向に）転送する。XMM レジスタの下位クワッドワードはそのまま残される。

MOVLHPS（move packed single-precision floating-point low to high）命令は、2 つのパックド単精度浮動小数点値を、ソース XMM レジスタの下位クワッドワードから、デスティネーション XMM レジスタの上位クワッドワードに転送する。デスティネーション・レジスタの下位クワッドワードはそのまま残される。

MOVHLPS（move packed single-precision floating-point high to low）命令は、2 つのパックド単精度浮動小数点値を、ソース XMM レジスタの上位クワッドワードから、デスティネーション XMM レジスタの下位クワッドワードに転送する。デスティネーション・レジスタの上位クワッドワードはそのまま残される。

MOVMSKPS（move packed single-precision floating-point mask）命令は、XMM レジスタ内の 4 つのパックド単精度浮動小数点値の最上位ビットを、汎用レジスタに転送する。この 4 ビット値は、分岐を実行するための条件として使用される。

10.4.1.2. SSE 算術演算命令

SSE 算術演算命令は、パックドおよびスカラ単精度浮動小数点値に対して、加算、減算、乗算、除算、逆数計算、平方根計算、平方根の逆数計算、最大値 / 最小値計算を実行する。

ADDPS（add packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランド同士を加算する。SUBPS（subtract packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランド同士を減算する。

ADDSS（add scalar single-precision floating-point values）命令は、2 つのオペランドの最下位の単精度浮動小数点値を加算し、その結果をデスティネーション・オペランドの

最下位のダブルワードに格納する。SUBSS（subtract scalar single-precision floating-point values）命令は、2つのオペランドの最下位の単精度浮動小数点値を加算し、その結果をデスティネーション・オペランドの最下位のダブルワードに格納する。

MULPS（multiply packed single-precision floating-point values）命令は、2つのパックド単精度浮動小数点オペランド同士を乗算する。

MULSS（multiply scalar single-precision floating-point values）命令は、2つのオペランドの最下位の単精度浮動小数点値を乗算し、その結果をデスティネーション・オペランドの最下位のダブルワードに格納する。

DIVPS（divide packed single-precision floating-point values）命令は、2つのパックド単精度浮動小数点オペランドの間で除算を行う。

DIVSS（divide scalar single-precision floating-point values）命令は、2つのオペランドの最下位の単精度浮動小数点値の間で除算を行い、その結果をデスティネーション・オペランドの最下位のダブルワードに格納する。

RCPPS（compute reciprocals of packed single-precision floating-point values）命令は、パックド単精度浮動小数点オペランドの値の逆数の近似値を計算する。

RCPSS（compute reciprocal of scalar single-precision floating-point values）命令は、ソース・オペランドの最下位の単精度浮動小数点値の逆数の近似値を計算し、その結果をデスティネーション・オペランドの最下位のダブルワードに格納する。

SQRTPS（compute square roots of packed single-precision floating-point values）命令は、パックド単精度浮動小数点オペランドの値の平方根を計算する。

SQRTSS（compute square root of scalar single-precision floating-point values）命令は、ソース・オペランドの最下位の単精度浮動小数点値の平方根を計算し、その結果をデスティネーション・オペランドの最下位のダブルワードに格納する。

RSQRTPS（compute reciprocals of square roots of packed single-precision floating-point values）命令は、パックド単精度浮動小数点オペランドの値の平方根の逆数の近似値を計算する。

RSQRTSS（reciprocal of square root of scalar single-precision floating-point value）命令は、ソース・オペランドの最下位の単精度浮動小数点値の平方根の逆数の近似値を計算し、その結果をデスティネーション・オペランドの最下位のダブルワードに格納する。

MAXPS（return maximum of packed single-precision floating-point values）命令は、2つのパックド単精度浮動小数点オペランド内の対応する値を比較し、それぞれ大きい方の値をデスティネーション・オペランドに返す。

MAXSS（return maximum of scalar single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランドの最下位の値を比較し、大きい方の値をデスティネーション・オペランドの最下位のダブルワードに返す。

MINPS（return minimum of packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランド内の対応する値を比較し、それぞれ小さい方の値をデスティネーション・オペランドに返す。

MINSS（return minimum of scalar single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランドの最下位の値を比較し、小さい方の値をデスティネーション・オペランドの最下位のダブルワードに返す。

10.4.2. SSE 論理演算命令

SSE 論理演算命令は、パックド単精度浮動小数点値の AND、AND NOT、OR、および XOR 演算を実行する。

ANDPS（bitwise logical AND of packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランドの AND（論理積）を返す。

ANDNPS（bitwise logical AND NOT of packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランドの AND NOT（否定論理積）を返す。

ORPS（bitwise logical OR of packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランドの OR（論理和）を返す。

XORPS（bitwise logical XOR of packed single-precision floating-point values）命令は、2 つのパックド単精度浮動小数点オペランドの XOR（排他的論理和）を返す。

10.4.2.1. SSE 比較命令

SSE 比較命令は、パックドおよびスカラ単精度浮動小数点値同士を比較し、比較の結果をデスティネーション・オペランドまたは EFLAGS レジスタに返す。

CMPPS（compare packed single-precision floating-point values）命令は、即値オペランドをプレディケートとして、2 つのパックド単精度浮動小数点オペランド内の対応する値を比較し、それぞれの結果について、すべて 1 またはすべて 0 の 32 ビット・マスクをデスティネーション・オペランドに返す。即値オペランドの値は、8 つの比較条件（等しい、より小さい、より小さいか等しい、順序化不可能、等しくない、より小さくない、より小さくなく等しくない、順序化）を自由に選択して指定できる。

CMPSS（compare scalar single-precision floating-point values）命令は、即値オペランドをプレディケートとして、2 つのパックド単精度浮動小数点オペランドの最下位の値

を比較し、その結果にしたがって、すべて 1 またはすべて 0 の 32 ビット・マスクをデスティネーション・オペランドの最下位のダブルワードに返す。即値オペランドは、CMPPS 命令と同じ比較条件を選択できる。

COMISS（compare scalar single-precision floating-point values and set EFLAGS）命令と UCOMISS（unordered compare scalar single-precision floating-point values and set EFLAGS）命令は、2 つのパックド単精度浮動小数点オペランドの最下位の値を比較し、その結果（より大きい、より小さい、等しい、または順序化不可能）にしたがって、EFLAGS レジスタの ZF、PF、CF ビットをセットする。2 つの命令の相違点は、次のとおりである。COMISS 命令は、ソース・オペランドが QNaN または SNaN である場合に、浮動小数点無効操作（#I）例外を通知する。UCOMISS 命令は、ソース・オペランドが SNaN である場合にのみ、無効操作例外を通知する。

10.4.2.2. SSE シャッフル命令とアンパック命令

SSE シャッフル命令とアンパック命令は、2 つのパックド単精度浮動小数点オペランドの内容をシャッフルまたはインターリーブし、その結果をデスティネーション・オペランドに格納する。

SHUFPS（shuffle packed single-precision floating-point values）命令は、デスティネーション・オペランドの 4 つのパックド単精度浮動小数点値のうち任意の 2 つを、デスティネーション・オペランドの下位の 2 つのダブルワードに入れる。また、ソース・オペランドの 4 つのパックド単精度浮動小数点値のうち任意の 2 つを、デスティネーション・オペランドの上位の 2 つのダブルワードに入れる（図 10-7. を参照）。ソース・オペランドとデスティネーション・オペランドに同じレジスタを使用すれば、SHUFPS 命令は、任意の順の 4 つの単精度浮動小数点値をシャッフルできる。

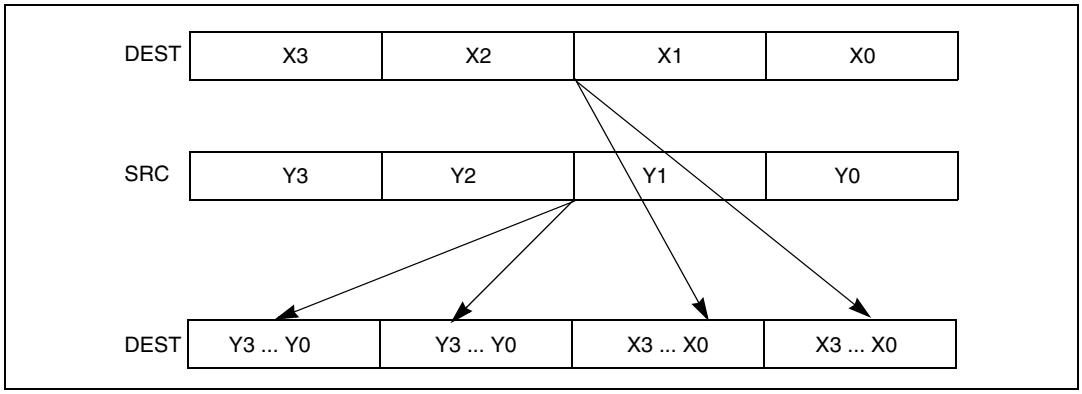


図 10-7. SHUFPS 命令のパックド・シャッフル操作

UNPCKHPS（unpack and interleave high packed single-precision floating-point values）命令は、ソース・オペランドおよびデスティネーション・オペランドの上位の単精度浮動小数点値をアンパックしてインタリーブし、その結果をデスティネーション・オペランドに格納する（図 10-8. を参照）。

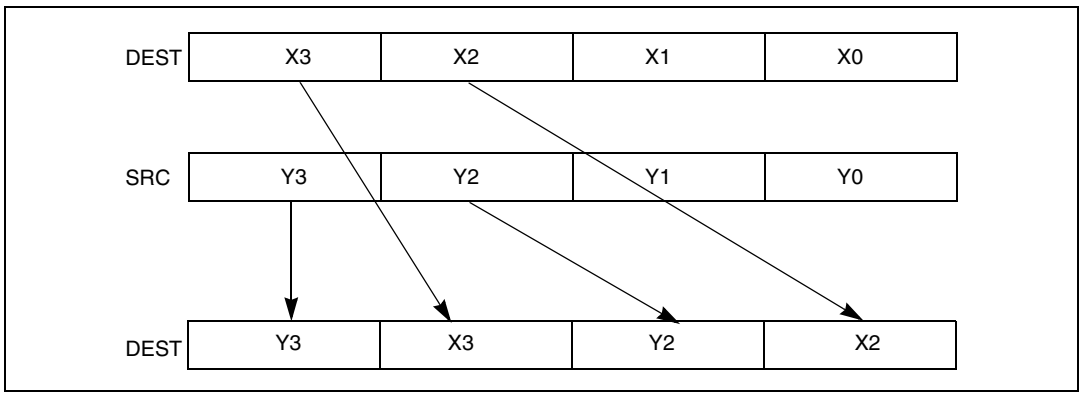


図 10-8. UNPCKHPS 命令のアンパック・ハイ操作とインタリーグ操作

UNPCKLPS（unpack and interleave low packed single-precision floating-point values）命令は、ソース・オペランドおよびデスティネーション・オペランドの下位の単精度浮動小数点値をアンパックしてインタリーブし、その結果をデスティネーション・オペランドに格納する（図 10-9. を参照）。

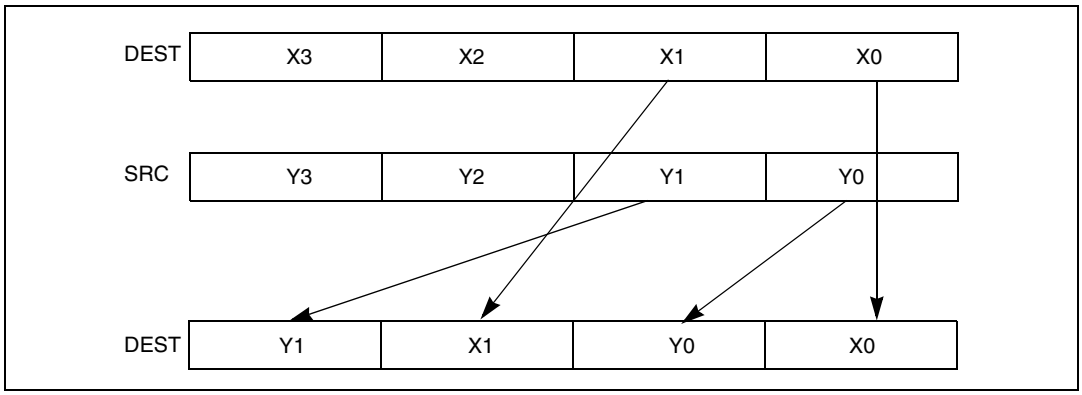


図 10-9. UNPCKLPS 命令のアンパック・ロー操作とインタリーブ操作

10.4.3. SSE 変換命令

SSE の変換命令（図 11-8. を参照）は、単精度浮動小数点フォーマットとダブルワード整数フォーマットの間で、パックド変換およびスカラ変換を実行する。

CVTPI2PS（convert packed doubleword integer to packed single-precision floating-point values）命令は、2 つの符号付きパックド・ダブルワード整数を、2 つのパックド単精度浮動小数点値に変換する。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。

CVTSI2SS（convert doubleword integer to scalar single-precision floating-point values）命令は、1 つの符号付きダブルワード整数を、1 つの単精度浮動小数点値に変換する。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。

CVTPS2PI（convert packed single-precision floating-point values to packed doubleword integers）命令は、2 つのパックド単精度浮動小数点値を、2 つの符号付きパックド・ダブルワード整数に変換する。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。CVTTPS2PI（convert with truncation packed single-precision floating-point values to packed doubleword integer）命令は、CVTPS2PI 命令によく似ているが、ソース・オペランドの値を整数値に丸めるときに切り捨てを使用する点が異なる。（4.8.4.2. 項「SSE および SSE2 変換命令による切り捨て」を参照）。

CVTSS2SI（convert scalar single-precision floating-point valve to doubleword integer）命令は、1 つの単精度浮動小数点値を、1 つの符号付きダブルワード整数に変換する。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。CVTTSS2SI（convert with truncation scalar single-precision floating-point valve to doubleword integer）命令は、CVTSS2SI 命令によく似ているが、ソース・オペランドの値を整数値に丸めるときに切り捨てを使用する点が異なる。（4.8.4.2. 項「SSE および SSE2 変換命令による切り捨て」を参照）。

10.4.4. SSE 64 ビット SIMD 整数命令

SSE では、以下の 64 ビット・パックド整数命令が IA-32 アーキテクチャに追加された。これらの命令は、MMX テクノロジ・レジスタおよび 64 ビット・メモリ・ロケーションのデータを操作する。

注記

IA-32 プロセッサが SSE2 に対応している場合は、これらの命令は拡張され、XMM レジスタおよび 128 ビット・メモリ・ロケーションの 128 ビット・オペランドも操作する。

PAVGB（compute average of packed unsigned byte integers）命令と PAVGW（compute average of packed unsigned word integers）命令は、それぞれ、2 つのパックド符号なしバイトまたはワード整数オペランドの SIMD 平均を計算する。2 つのパックド・ソース・オペランド内のデータ要素の対応する各組について、要素同士が加算され、一時的な和に 1 が加算され、その結果が 1 ビット右にシフトされる。

PEXTRW（extract word）命令は、選択したワードを MMX テクノロジ・レジスタから汎用レジスタにコピーする。

PINSRW（insert word）命令は、汎用レジスタまたはメモリから MMX テクノロジ・レジスタ内の選択した位置に 1 ワードをコピーする。

PMAXUB（maximum of packed unsigned byte integers）命令は、2 つのパックド・オペランド内の対応する符号なしバイト整数を比較し、それぞれ大きい方の値をデスティネーション・オペランドに返す。

PMINUB（minimum of packed unsigned byte integers）命令は、2 つのパックド・オペランド内の対応する符号なしバイト整数を比較し、それぞれ小さい方の値をデスティネーション・オペランドに返す。

PMAXSW（maximum of packed signed word integers）命令は、2 つのパックド・オペランド内の対応する符号付きワード整数を比較し、それぞれ大きい方の値をデスティネーション・オペランドに返す。

PMINSW（minimum of packed signed word integers）命令は、2 つのパックド・オペランド内の対応する符号付きワード整数を比較し、それぞれ小さい方の値をデスティネーション・オペランドに返す。

PMOVMSKB（move byte mask）命令は、MMX テクノロジ・レジスタ内のパックドバイト整数から 8 ビット・マスクを作成し、その結果を汎用レジスタの最下位バイトに格納する。このマスクは、MMX テクノロジ・レジスタの各バイトの最上位ビットで

構成される（128 ビット・オペランドを操作する場合は、16 ビット・マスクが作成される）。

PMULHUW（multiply packed unsigned word integers and store high result）命令は、2 つのソース・オペランド内の対応する各ワードの符号なし SIMD 乗算を実行し、それぞれの結果の上位ワードを MMX テクノロジ・レジスタに返す。

PSADBW（compute sum of absolute differences）命令は、2 つのソース・オペランド内の対応する符号なしバイト整数の SIMD の絶対差を計算して、それらの差を加算し、得られた和をデスティネーション・オペランドの最下位ワードに格納する。

PSHUFW（shuffle packed word integers）命令は、8 ビットの即値オペランドで指定される順序にしたがって、ソース・オペランド内のワードをシャッフルし、その結果をデスティネーション・オペランドに返す。

10.4.5. MXCSR ステート管理命令

MXCSR ステート管理命令の LDMXCSR と STMXCSR はそれぞれ MXCR レジスタの状態のロードと保存を行う。LDMXCSR 命令はメモリから MXCSR レジスタをロードし、STMXCSR 命令はレジスタ内容をメモリに保存する。

10.4.6. キャッシュ制御命令、プリフェッチ命令、メモリアクセス順序命令

SSE では、プログラムによってデータのキャッシュ処理をよりきめ細かく制御できるように、いくつかの新しい命令が追加された。また、ストリーミング SIMD 拡張命令には、PREFETCHh 命令と SFENCE 命令が追加された。PREFETCH*h* 命令は、指定されたキャッシュ・レベルにデータをプリフェッチできる。SFENCE 命令は、ストア時のプログラムの順序設定を強化する。これらの命令について、以下の各項で説明する。

10.4.6.1. キャッシュ制御命令

次の 3 つの命令は、非テンポラルなヒントを使用して、MMX テクノロジ・レジスタおよび XMM レジスタからメモリにデータをストアする。非テンポラルなヒントは、可能な限りデータをキャッシュ階層内に書き込まずにメモリにストアするようにプロセッサに指示する（非テンポラルなストアとヒントについての詳細は、10.4.6.2. 項「テンポラルなデータと非テンポラルなデータのキャッシュ処理」を参照のこと）。

MOVNTQ（store quadword using non-temporal hint）命令は、非テンポラルなヒントを使用して、パックド整数データを MMX テクノロジ・レジスタからメモリにストアする。

MOVNTPS（store packed single-precision floating-point values using non-temporal hint）命令は、非テンポラルなヒントを使用して、パックド浮動小数点データを XMM レジスタからメモリにストアする。

MASKMOVQ（store selected bytes of quadword）命令は、書き込むバイトをバイトマスクで個々に選択した上で、選択したバイト整数を MMX テクノロジ・レジスタからメモリにストアする。この命令も非テンポラルなヒントを使用する。

10.4.6.2. テンポラルなデータと非テンポラルなデータのキャッシュ処理

プログラムが参照するデータは、テンポラルなデータまたは非テンポラルなデータである。テンポラルなデータとは、近い将来に再び使用されるデータである。非テンポラルなデータとは、一度参照されると、近い将来には再び使用されないデータである。例えば、一般的にプログラム・コードはテンポラルであるが、3D グラフィックス・アプリケーションの表示リストなどのマルチメディア・データは、非テンポラルであることが多い。プロセッサのキャッシュを効率的に使用するためには、テンポラルなデータをキャッシュし、非テンポラルなデータはキャッシュしないことが一般的に望ましい。プロセッサのキャッシュを非テンポラルなデータでオーバーロードすることを、「キャッシュの汚染」と呼ぶ。SSE および SSE2 のキャッシュ制御命令により、プログラムは、キャッシュの汚染を最小限に抑えるように、非テンポラルなデータをメモリに書き込むことができる。

これらの SSE および SSE2 非テンポラル・ストア命令は、アクセス先のメモリをライト・コンバイニング（WC）タイプとして扱うことで、キャッシュ汚染を最小限に抑える。プログラムがこれらの命令を使用して非テンポラルなストアを指定し、デスティネーション領域がキャッシュ可能メモリ（WB、WT、または WC メモリタイプ）としてマッピングされている場合は、プロセッサは以下の処理を実行する。

- 書き込み先のメモリ・ロケーションがキャッシュ階層内にある場合は、キャッシュ内のデータを排出する。
- WC セマンティクスを使用して、非テンポラルなデータをメモリに書き込む。

WC セマンティクスを使用すると、ストア・トランザクションは緩い順序設定になる。つまり、データはプログラムの順序でメモリに書き込まれるとは限らず、ストア操作はライト・アロケーションを行わない（すなわち、プロセッサは、ストアを実行する前に、対応するキャッシュ・ラインをキャッシュ階層内にフェッチしない）。また、プロセッサによっては、これらのストアのコラプスとコンバインを行うことがある。

非テンポラルなストアで指定されたメモリアドレスがキャッシュ不可メモリ内にある場合は、書き込み先領域のメモリタイプが、非テンポラルなヒントより優先する。ここで、キャッシュ不可という用語は、書き込み先の領域が UC または WP メモリタイプとしてマッピングされているという意味である。

一般的に、WC セマンティクスでは、他のプロセッサおよび他のシステム・エージェント（グラフィック・カードなど）に対するコヒーレンシを、ソフトウェアによって保証する必要がある。データの生産者 / 消費者モデルを使用する場合は、適切な同期化操作とフェンス操作を実行しなければならない。フェンス操作によって、すべてのシステム・エージェントは、ストアされたデータに対してグローバルにアクセス可能になる。例えば、フェンス操作を行わないと、書き込まれたキャッシュ・ラインがプロセッサ内に滞留し、他のエージェントからアクセスできなくなる場合がある。

プロセッサによっては、すでにキャッシュ階層内にあるデータをその位置で更新することによって、非テンポラルなストアを実行するものがある。この場合も、デスティネーション領域は WC としてマッピングされていなければならない。デスティネーション領域が WC ではなく、WB または WT としてマッピングされていると、プロセッサの見込み的な読み込みによって、データがキャッシュにロードされる可能性がある。この場合、非テンポラルなストアはその位置でデータを更新するため、これ以降のフェンス操作によってデータがプロセッサからフラッシュされなくなる。

メモリタイプのエイリアスがある場合、バス上で認識可能なメモリタイプは、プロセッサによって異なる。1 つの例では、バスに書き込まれるメモリタイプは、プログラムの順序でそのラインに対する最初のストアのメモリタイプを反映する。しかし、プロセッサによっては、他の方法が使用される可能性がある。したがって、この動作は予約済みとみなす必要がある。特定のプロセッサの動作に依存すると、今後のプロセッサとの互換性を損なうおそれがある。

10.4.6.3. PREFETCH*h* 命令

PREFETCH*h* 命令によって、プログラムは、必要なときにプロセッサのロードおよびストアユニットの近くにデータがあるように、プロセッサ内の指示されたキャッシュ・レベルにデータをロードすることができる。この命令は、アドレス指定されたバイトを含む、アライメントの合った 32 バイトのデータ（プロセッサによっては、さらに大量のデータ）を、時間的なローカリティのヒントによって指定されたキャッシュ階層内の位置にフェッチする（表 10-1. を参照）。この表では、第 1 レベルのキャッシュがプロセッサに最も近く、第 2 レベルのキャッシュは第 1 レベルのキャッシュよりプロセッサから遠い。キャッシュ・ヒントは、テンポラルなデータまたは非テンポラルなデータのプリフェッチを指定する（10.4.6.2. 項「テンポラルなデータと非テンポラルなデータのキャッシュ処理」を参照）。テンポラルなデータに対するこれ以降のアクセスは、通常のアクセスと同じように扱われる。非テンポラルなデータに対するこれ以降のアクセスでは、キャッシュ汚染が最小限に抑えられる。指定されたデータが、よりプロセッサに近いキャッシュ階層レベルにすでに存在する場合は、PREFETCH*h* 命令はデータを移動しない。PREFETCH*h* 命令は、プログラムの機能に関わる動作には影響を与えない。

表 10-1. PREFETCH*h* 命令のキャッシュ・ヒント

PREFETCH*h* 命令のニーモニック	動作
PREFETCHT0	テンポラルなデータ - キャッシュ階層のすべてのレベルにデータをフェッチする。 • インテル® Pentium® III プロセッサ - L1 キャッシュまたは L2 キャッシュ • インテル® Pentium® 4 プロセッサとインテル® Xeon™ プロセッサ - L2 キャッシュ
PREFETCHT1	テンポラルなデータ - キャッシュ階層のレベル 2 およびそれ以上にデータをフェッチする。 • インテル Pentium III プロセッサ - L2 キャッシュ • インテル Pentium 4 プロセッサとインテル Xeon プロセッサ - L2 キャッシュ
PREFETCHT2	テンポラルなデータ - キャッシュ階層のレベル 2 およびそれ以上にデータをフェッチする。 • インテル Pentium III プロセッサ - L2 キャッシュ • インテル Pentium 4 プロセッサとインテル Xeon プロセッサ - L2 キャッシュ
PREFETCHNTA	非テンポラルなデータ - プロセッサに近い位置にデータをフェッチし、キャッシュ汚染を最小限に抑える。 • インテル Pentium III プロセッサ - L1 キャッシュ • インテル Pentium 4 プロセッサとインテル Xeon プロセッサ - L2 キャッシュ

PREFETCH*h* 命令についての詳細は、11.6.13. 項「キャッシュ・ヒント命令」を参照のこと。

10.4.6.4. SFENCE 命令

SFENCE（Store Fence）命令は、メモリストア操作のフェンスを作成することによって、書き込みの順序を制御する。この命令は、プログラムの順序でストアフェンスに先行するすべてのストア命令の結果が、フェンスに後続するストア命令より前に、グローバルにアクセス可能になることを保証する。SFENCE 命令は、順序設定の緩いデータを生成するプロシージャとそのデータを参照するプロシージャの間の順序を保証するための効率的な方法である。

10.5. FXSAVE 命令と FXRSTOR 命令

FXSAVE 命令と FXRSTOR 命令は、（SSE の導入より前に）インテル® Pentium® II プロセッサ・ファミリで IA-32 アーキテクチャに導入された。これらの命令の元のバージョンは、それぞれ、x87 FPU レジスタの状態の高速セーブとリストアを実行していた（FXSAVE 命令と FXRSTOR 命令は、x87 FPU データレジスタの状態をセーブすることによって、暗黙的に MMX® テクノロジ・レジスタの状態のセーブとリストアも実行する）。

SSE では、これらの命令の有効範囲が拡張され、x87 FPU および MMX テクノロジ・ステートと共に、XMM レジスタと MXCSR レジスタの状態のセーブとリストアも行うようになった。

FXSAVE 命令と FXRSTOR 命令は、FSAVE/FNSAVE 命令と FRSTOR 命令の代わりに使用することができる。ただし、FXSAVE 命令と FXRSTOR 命令の動作は、FSAVE/FNSAVE 命令と FRSTOR 命令の動作と同じではない。

注記

FXSAVE 命令と FXRSTOR 命令は、SSE グループの一部とは見なされない。FXSAVE 命令と FXRSTOR 命令は、これらの命令が特定の IA-32 プロセッサ上でサポートされるかどうかを示す独自の CPUID 機能ビット（EAX レジスタのビット 24）を持つ。SSE の CPUID 機能ビット（EAX レジスタのビット 25）は、FXSAVE 命令と FXRSTOR 命令のサポートの有無を示さない。

10.6. SSE の例外の処理

SSE で生成される一般例外および SIMD 浮動小数点例外と、例外発生時の処理のガイドラインについては、11.5. 節「SSE、SSE2、SSE3 の例外」を参照のこと。

10.7. SSE によるアプリケーションの作成

SSE を使用してアプリケーションとオペレーティング・システム・コードを作成する方法については、11.6. 節「SSE および SSE2 によるアプリケーションの作成」を参照のこと。

11

ストリーミング SIMD 拡張命令 2（SSE2）によるプログラミング

第 11 章
ストリーミング SIMD 拡張命令 2（SSE2）によるプログラミング

11

ストリーミング SIMD 拡張命令 2（SSE2）はインテル® Pentium® 4 プロセッサとインテル® Xeon™ プロセッサで IA-32 アーキテクチャに導入された。これらの拡張命令によって、高度な 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、電子商取引、インターネット、科学計算、工学計算などのアプリケーション向けに、IA-32 プロセッサのパフォーマンスが強化される。

本章では、SSE2 および SSE2 や SSE を使用したアプリケーション・プログラムを作成する際に必要な内容を記載している。

11.1. SSE2 の概要

SSE2 は、MMX® テクノロジおよび SSE と同じように、SIMD（Single Instruction, Multiple Data）実行モデルを使用する。SSE2 では、従来の SIMD 実行モデルが拡張され、パックド倍精度浮動小数点値と 128 ビット・パックド整数の処理がサポートされた。

SSE2 は、IA-32 アーキテクチャに以下の機能を追加するが、すべての既存の IA-32 プロセッサ、アプリケーション、およびオペレーティング・システムとの下方互換性を維持している。

- 以下の 6 種類のデータ型
  - 128 ビット・パックド倍精度浮動小数点（2 つの IEEE 規格 754 倍精度浮動小数点値を、1 つのダブル・クワッドワードにパックしたもの）
  - 128 ビット・パックド・バイト整数
  - 128 ビット・パックド・ワード整数
  - 128 ビット・パックド・ダブルワード整数
  - 128 ビット・パックド・クワッドワード整数
- 追加されたデータ型をサポートする命令と、既存の SIMD 整数演算を拡張する命令
  - パックドおよびスカラ倍精度浮動小数点命令
  - 追加された 64 ビットおよび 128 ビット SIMD 整数命令
  - MMX テクノロジおよび SSE で導入された SIMD 整数命令の 128 ビット版
  - 追加されたキャッシュ制御命令と命令順序付け命令
- 既存の IA-32 命令の修正により、SSE2 の機能をサポート

- CPUID 命令の拡張と修正
- RDPMC 命令の修正

これらの新しい機能によって、IA-32 アーキテクチャの SIMD プログラミング・モデルは、以下の 3 つの点で大きく強化される。

- パック ド倍精度浮動小数点値のペアに対する SIMD 演算を実行できる。これによって、XMM レジスタ内で実行される計算の精度が向上する。これによって、科学計算 / 工学計算アプリケーションや、レイ・トレーシングなどの高度な 3D ジオメトリ手法を使用するアプリケーションでのプロセッサのパフォーマンスが強化される。また、XMM レジスタの下位クワッドワード内の 1 つの（スカラ）倍精度浮動小数点値を操作する命令によって、さらに柔軟な処理が可能になる。
- XMM レジスタ内の 128 ビット・パック ド整数（バイト、ワード、ダブルワード、およびクワッドワード）を操作できる。これによって、パック ド整数に対する SIMD 演算の実行時の柔軟性とスループットが向上する。この機能は、RSA 認証や RC5 暗号化などのアプリケーションに特に効果的である。MMX テクノロジ、SSE、SSE2 で使用できる、すべての SIMD レジスタ、データ型、命令を使用すれば、パック ド単精度および倍精度浮動小数点データと 64 ビットおよび 128 ビット・パック ド整数データを上手に組み合わせたアルゴリズムを開発できる。
- SSE2 では、SSE で導入された、SIMD データのキャッシュ処理を制御する機能が拡張された。新しいキャッシュ制御命令を使用して、キャッシュを汚染することなく、XMM レジスタとの間でデータのストリーミングが行える。また、データを実際に使用する前に、そのデータをプリフェッチできる。

SSE2 は、IA-32 プロセッサ用に作成されたすべてのソフトウェアとの完全な互換性を持つ。すべての既存のソフトウェアは、SSE2 を搭載したプロセッサ上でも、SSE2 を組み込んだ既存および新規のアプリケーションと共存させても、修正なしで正常に動作し続ける。CPUID 命令の拡張によって、SSE2 をサポートするかどうかを簡単に検出できるようになった。SSE2 は、SSE と同じレジスタを使用する。したがって、コンテキスト・スイッチの際にプログラムのステートのセーブとリストアを実行するために、オペレーティング・システムに新機能を追加する必要はない。オペレーティング・システムが SSE をサポートしていれば十分である。

SSE2 には、IA-32 アーキテクチャのすべての実行モード（プロテクト・モード、実アドレスモード、仮想 8086 モード）からアクセスできる。

本章では、128 ビット XMM 浮動小数点レジスタセット、データ型、SSE2 など、SSE2 のプログラミング環境について説明する。また、本章では、SSE と SSE2 で発生する例外についても説明する。さらに、SSE と SSE2 を使用してアプリケーションを作成する際のガイドラインについても説明する。

SSE2 についての詳細は、以下の個所を参照のこと。

- 『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 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、中巻 B』の第 4 章「命令セット・リファレンス N-Z」では、SSE3 について詳しく説明する。
- 『IA-32 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、下巻』の第 12 章「SSE および SSE2 のシステム・プログラミング」では、SSE と SSE2 をオペレーティング・システム環境に統合する際のガイドラインについて説明する。

11.2. SSE2 のプログラミング環境

図 11-1. は、SSE2 のプログラミング環境を示している。SSE2 では、新しいレジスタや新しい命令実行ステートは定義されていない。SSE2 の操作は、次のように、XMM レジスタ、MMX® テクノロジ・レジスタ、IA-32 汎用レジスタ内で実行される。

- **XMM レジスタ。** 8 つの XMM レジスタ（図 10-2. を参照）を使用して、パックドまたはスカラ倍精度浮動小数点データを操作する。スカラ演算とは、XMM レジスタの最下位クワッドワードに格納される、個々の（アンパックされた）倍精度浮動小数点値に対して実行される演算である。XMM レジスタは、128 ビットパックド整数データの操作にも使用される。これらは、XMM0 ～ XMM7 の名前で参照される。

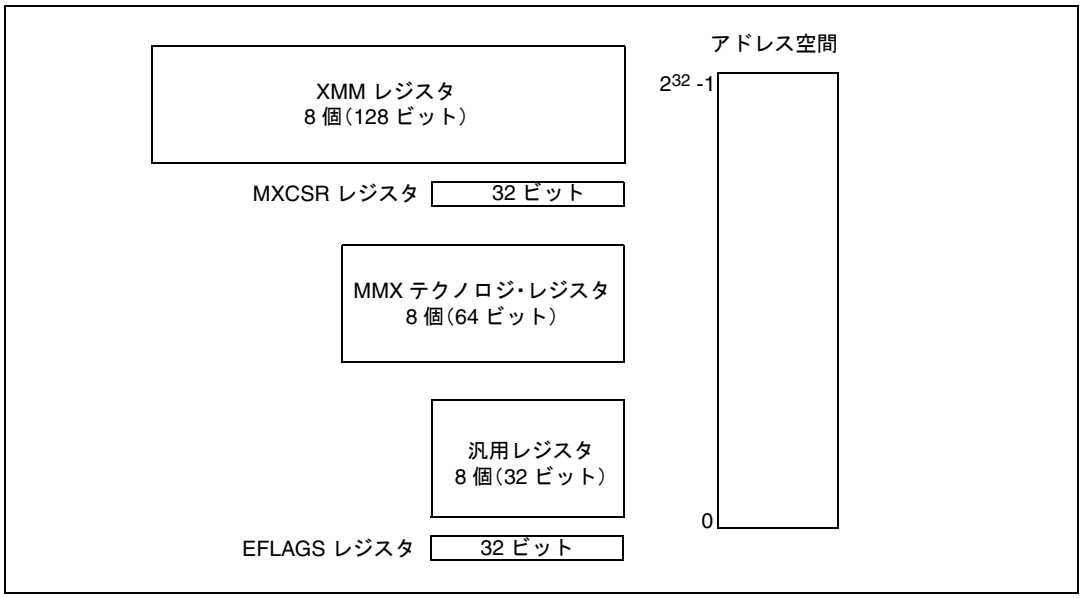


図 11-1. SSE2 の実行環境

- **MXCSR レジスタ。**この 32 ビット・レジスタ（図 10-3. を参照）は、浮動小数点演算に使用されるステータス・ビットと制御ビットを格納する。このレジスタ内のデノーマル・ゼロ・フラグとゼロ・フラッシュ・フラグを使用して、デノーマル・ソース・オペランドとデノーマル（アンダーフロー）結果を処理する際のパフォーマンスを改善できる。これらのフラグの機能についての詳細は、10.2.2.4. 項「デノーマル・ゼロ」と 10.2.2.3. 項「ゼロ・フラッシュ」を参照のこと。
- **MMX テクノロジ・レジスタ。**8 つの MMX テクノロジ・レジスタ（図 9-2. を参照）を使用して、64 ビット・パックド整数データの操作を実行する。MMX テクノロジ・レジスタと XMM レジスタの間で実行される操作では、MMX テクノロジ・レジスタがオペランドの格納にも使用される。MMX テクノロジ・レジスタは、MM0 ～ MM7 の名前で参照される。
- **汎用レジスタ。**8 つの汎用レジスタ（図 3-4. を参照）と既存の IA-32 アドレス指定モードを組み合わせて、メモリ内のオペランドをアドレス指定する。MMX テクノロジ・レジスタと XMM レジスタは、メモリのアドレス指定には使用できない。一部の SSE2 では、汎用レジスタがオペランドの格納にも使用される。汎用レジスタは、EAX、EBX、ECX、EDX、EBP、ESI、EDI、ESP の名前で参照される。
- **EFLAGS レジスタ。**この 32 ビット・レジスタ（図 3-7. を参照）は、比較操作の結果を記録する。

11.2.1. SSE2 と SSE、MMX® テクノロジ、および x87 FPU のプログラミング環境の互換性

SSE2 では、IA-32 実行環境に新しいステートは導入されていない。SSE2 は、SSE を拡張したものである。SSE2 と SSE は、互いに完全な互換性を持ち、同じステート情報を共有する。SSE と SSE2 は、完全な互換性を持つ。これらの命令は、同じ命令ストリーム内で実行できる。命令セットの切り替え時にステートを保存する必要はない。

XMM レジスタは、x87 FPU レジスタおよび MMX® テクノロジ・レジスタに依存しない。したがって、XMM レジスタに対して実行される SSE と SSE2 の操作は、x87 FPU または MMX テクノロジの操作と並行して実行できる（11.6.7. 項「SSE および SSE2 と x87 FPU 命令および MMX® 命令の相互作用」を参照）。

FXSAVE 命令と FXRSTOR 命令は、SSE と SSE2 のステートを、x87 FPU と MMX のステートと一緒にセーブおよびリストアする。

11.2.2. デノーマル・ゼロ・フラグ

デノーマル・ゼロ・フラグ（MXCSR レジスタのビット 6）は、SSE2 で IA-32 アーキテクチャに追加された。このフラグについては、10.2.2.4. 項「デノーマル・ゼロ」を参照のこと。

11.3. SSE2 のデータ型

SSE2 では、1 種類の 128 ビット・パックド浮動小数点データ型と 4 種類の 128 ビット SIMD 整数データ型が IA-32 アーキテクチャに追加された（図 11-2. を参照）。

- **パックド倍精度浮動小数点数。**この 128 ビット・データ型は、2 つの IEEE 64 ビット倍精度浮動小数点値を、1 つのダブル・クワッドワードにパックしたものである（64 ビット倍精度浮動小数点値のレイアウトについては、図 4-3. を参照。倍精度浮動小数点値についての詳細は、4.2.2. 項「浮動小数点データ型」を参照）。
- **128 ビット・パックド整数。**4 種類のパックド整数データ型は、それぞれ 16 個のバイト整数、8 個のワード整数、4 個のダブルワード整数、2 個のクワッドワード整数で構成される（128 ビット・パックド整数についての詳細は、4.6.2. 項「128 ビット・パックド SIMD データ型」を参照）。

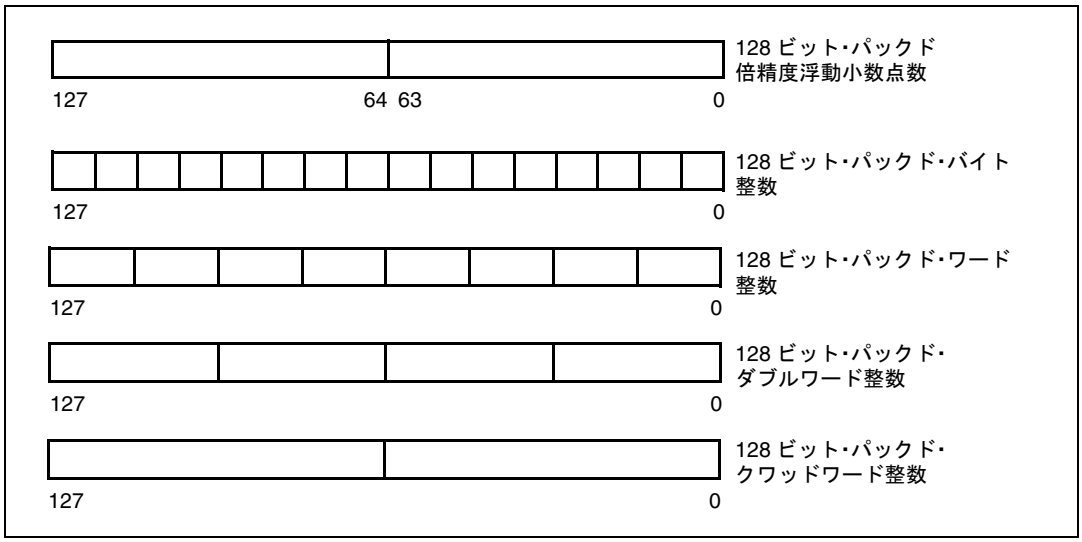


図 11-2. SSE2 のデータ型

これらのデータ型はすべて、XMM レジスタまたはメモリ内で操作される。変換命令を使用して、これらの 128 ビット・データ型と、64 ビットおよび 32 ビット・データ型の間の変換を実行できる。

128 ビット・パックド・メモリ・オペランドのアドレスは、以下の場合を除き、16 バイトにアライメントが合っていなければならない。

- MOVUPS 命令は、アライメントの合っていないデータへのアクセスをサポートしている。
- スカラ命令が、アライメントの必要条件に従わない 8 バイト・メモリ・オペランドを使用する場合。

図 4-2. は、メモリ内の 128 ビット（ダブル・クワッドワード）データ型および 64 ビット（クワッドワード）データ型のバイト・オーダを示している。

11.4. SSE2 命令

SSE2 は、以下の 4 つの機能グループに分類される。

- パックドおよびスカラ倍精度浮動小数点命令
- 64 ビットおよび 128 ビット SIMD 整数命令
- MMX® テクノロジおよび SSE で導入された SIMD 整数命令の 128 ビット拡張
- キャッシュ制御命令および命令順序命令

以下の各項では、各命令の概要を説明する。

11.4.1. パックドおよびスカラ倍精度浮動小数点命令

パックドおよびスカラ倍精度浮動小数点命令は、以下のサブグループに分類される。

- データ転送命令
- 算術演算命令
- 比較命令
- 変換命令
- 論理演算命令
- シャッフル命令

パックド倍精度浮動小数点命令は、パックド単精度浮動小数点命令と同じように SIMD を操作する（図 11-3. を参照）。各ソース・オペランドには、2 つの倍精度浮動小数点値が格納される。デスティネーション・オペランドには、各オペランド内の対応する値（X0 と Y0、X1 と Y1）に対して並行して実行された操作（OP）の結果が格納される。

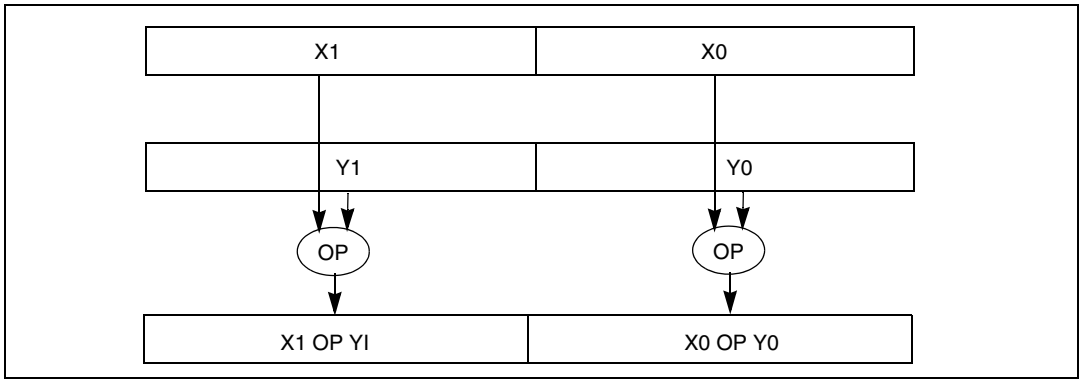


図 11-3. パック ド倍精度浮動小数点の操作

スカラ倍精度浮動小数点命令は、図11-4. に示すように2つのソース・オペランド（X0とY0）の最下位クワッドワードを操作する。第1のソース・オペランドの上位クワッドワード（X1）は、デスティネーション・オペランドにそのまま渡される。このスカラ操作は、x87 FPU データレジスタ内で実行される浮動小数点操作によく似ている。このスカラ操作は、x87 FPU制御ワード内の精度制御フィールドを倍精度（53ビット仮数）に設定して、x87 FPU データレジスタ内で浮動小数点演算を実行するのとよく似ている。XMM レジスタと x87 FPU データレジスタの両方でスカラ倍精度浮動小数点操作を実行する場合に互換性のある結果を得る方法については、11.6.8. 項「SIMD浮動小数点データ型と x87 FPU浮動小数点データ型の互換性」を参照のこと。

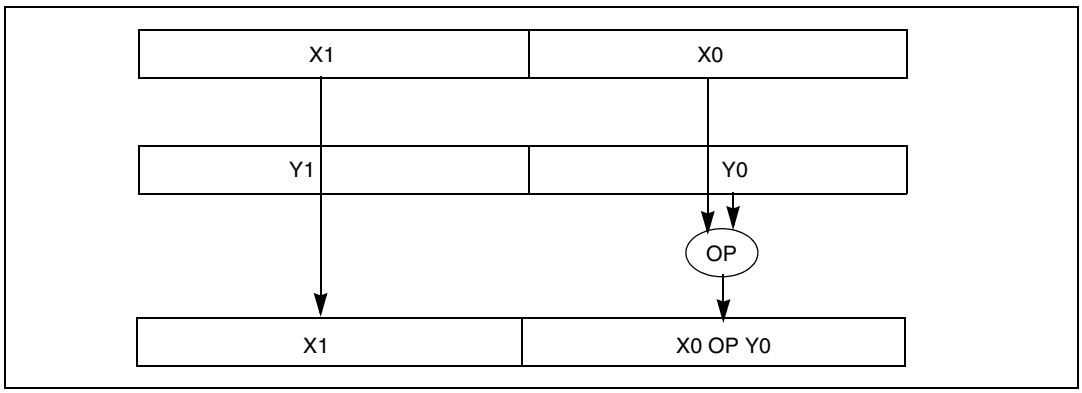


図 11-4. スカラ倍精度浮動小数点の操作

11.4.1.1. データ転送命令

データ転送命令は、XMM レジスタ同士の間およびXMM レジスタとメモリの間で、倍精度浮動小数点データを転送する。

MOVAPD（move aligned packed double-precision floating-point）命令は、128 ビット・パックド倍精度浮動小数点オペランドを、メモリから XMM レジスタに（または、その反対方向に）転送するか、XMM レジスタ同士の間で転送する。メモリアドレスは、16 バイトにアライメントが合っていなければならない。アライメントが合っていないと、一般保護例外（#GP）が発生する。

MOVUPD（move unaligned packed double-precision floating-point）命令は、128 ビット・パックド倍精度浮動小数点オペランドを、メモリから XMM レジスタに（または、その反対方向に）転送するか、XMM レジスタ同士の間で転送する。メモリアドレスのアライメントが合っている必要はない。

MOVSD（move scalar double-precision floating-point）命令は、64 ビット倍精度浮動小数点オペランドを、メモリから XMM レジスタの最下位クワッドワードに（または、その反対方向に）転送するか、XMM レジスタ同士の間で転送する。アライメント・チェックが有効になっている場合を除いて、メモリアドレスのアライメントが合っている必要はない。

MOVLPD（move low packed double-precision floating-point）命令は、64 ビット倍精度浮動小数点オペランドを、メモリから XMM レジスタの下位クワッドワードに（または、その反対方向に）転送する。XMM レジスタの上位クワッドワードはそのまま残される。アライメント・チェックが有効になっている場合を除いて、メモリアドレスのアライメントが合っている必要はない。

MOVHPD（move high packed double-precision floating-point）命令は、64 ビット倍精度浮動小数点オペランドを、メモリから XMM レジスタの上位クワッドワードに（または、その反対方向に）転送する。XMM レジスタの下位クワッドワードはそのまま残される。アライメント・チェックが有効になっている場合を除いて、メモリアドレスのアライメントが合っている必要はない。

MOVMSKPD（move packed double-precision floating-point mask）命令は、XMM レジスタ内の 2 つのパックド倍精度浮動小数点値の符号付きビットを抽出し、それらを汎用レジスタに保存する。この 2 ビット値は、分岐を実行するための条件として使用される。

11.4.1.2. SSE2 算術演算命令

SSE2 算術演算命令は、パックドおよびスカラ倍精度浮動小数点値に対して、加算、減算、乗算、除算、平方根計算、最大値 / 最小値計算を実行する。

ADDPD（add packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランド同士を加算する。SUBPD（subtract packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランド同士を減算する。

ADDSD（add scalar double-precision floating-point values）命令と SUBSD（subtract scalar double-precision floating-point values）命令は、それぞれ、2 つのオペランドの下位の倍精度浮動小数点値を加算または減算し、その結果をデスティネーション・オペランドの下位クワッドワードに格納する。

MULPD（multiply packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランド同士を乗算する。

MULSD（multiply scalar double-precision floating-point values）命令は、2 つのオペランドの下位の倍精度浮動小数点値を乗算し、その結果をデスティネーション・オペランドの下位クワッドワードに格納する。

DIVPD（divide packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの間で除算を行う。

DIVSD（divide scalar double-precision floating-point values）命令は、2 つのオペランドの下位の倍精度浮動小数点値の間で除算を行い、その結果をデスティネーション・オペランドの下位クワッドワードに格納する。

SQRTPD（compute square roots of packed double-precision floating-point values）命令は、パックド倍精度浮動小数点オペランドの値の平方根を計算する。

SQRTSD（compute square root of scalar double-precision floating-point values）命令は、ソース・オペランドの下位の倍精度浮動小数点値の平方根を計算し、その結果をデスティネーション・オペランドの下位クワッドワードに格納する。

MAXPD（return maximum of packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランド内の対応する値を比較し、それぞれ大きい方の値をデスティネーション・オペランドに返す。

MAXSD（return maximum of scalar double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの最下位の倍精度浮動小数点値を比較し、大きい方の値をデスティネーション・オペランドの最下位のクワッドワードに返す。

MINPD（return minimum of packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランド内の対応する値を比較し、それぞれ小さい方の値をデスティネーション・オペランドに返す。

MINSD（return minimum of scalar double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの最下位の値を比較し、小さい方の値をデスティネーション・オペランドの最下位のクワッドワードに返す。

11.4.1.3. SSE2 論理演算命令

SSE2 論理演算命令は、パックド倍精度浮動小数点値の AND、AND NOT、OR、および XOR 演算を実行する。

ANDPD（bitwise logical AND of packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの AND（論理積）を返す。

ANDNPD（bitwise logical AND NOT of packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの AND NOT（否定論理積）を返す。

ORPD（bitwise logical OR of packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの OR（論理和）を返す。

XORPD（bitwise logical XOR of packed double-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点オペランドの XOR（排他的論理和）を返す。

11.4.1.4. SSE2 比較命令

SSE2 比較命令は、パックドおよびスカラ倍精度浮動小数点値同士を比較し、比較の結果をデスティネーション・オペランドまたは EFLAGS レジスタに返す。

CMPPD（compare packed double-precision floating-point values）命令は、即値オペランドをプレディケートとして、2 つのパックド倍精度浮動小数点オペランド内の対応する値を比較し、それぞれの結果について、すべて 1 またはすべて 0 の 64 ビット・マスクをデスティネーション・オペランドに返す。即値オペランドの値は、8 つの比較条件（等しい、より小さい、より小さいか等しい、順序化不可能、等しくない、より小さくない、より小さくなく等しくない、または順序化）を自由に選択して指定できる。

CMPSD（compare scalar double-precision floating-point values）命令は、即値オペランドをプレディケートとして、2 つのパックド倍精度浮動小数点オペランドの最下位の値を比較し、その結果にしたがって、すべて 1 またはすべて 0 の 64 ビット・マスクをデスティネーション・オペランドの最下位のクワッドワードに返す。ソース・オペランドの上位クワッドワードは、デスティネーション・オペランドにそのまま渡される。即値オペランドは、CMPPD 命令と同じ比較条件を選択できる。

COMISD（compare scalar double-precision floating-point values and set EFLAGS）命令と UCOMISD（unordered compare scalar double-precision floating-point values and set EFLAGS）命令は、2 つのパックド倍精度浮動小数点オペランドの最下位の値を比較し、その結果（より大きい、より小さい、等しい、または順序化不可能）にしたがって、EFLAGS レジスタの ZF、PF、CF ビットをセットする。2 つの命令の相違点は、次のとおりである。COMISD 命令は、ソース・オペランドが QNaN または SNaN である場合に、浮動小数点無効操作（#I）例外を通知する。UCOMISD 命令は、ソース・オペランドが SNaN である場合にのみ、無効操作例外を通知する。

11.4.1.5. SSE2 シャッフル命令とアンパック命令

SSE2 シャッフル命令は、2 つのパック ド倍精度浮動小数点オペランドの内容をシャッフルまたはインターリーブし、その結果をデスティネーション・オペランドに格納する。

SHUFPD（shuffle packed double-precision floating-point values）命令は、デスティネーション・オペランドの 2 つのパック ド倍精度浮動小数点値のうちどちらかを、デスティネーション・オペランドの下位のクワッドワードに入れる。また、ソース・オペランドの 2 つのパック ド倍精度浮動小数点値のうちどちらかを、デスティネーション・オペランドの上位のクワッドワードに入れる（図 11-5. を参照）。SHUFPD 命令で、ソース・オペランドとデスティネーション・オペランドに同じレジスタを使用すれば、2 つのパック ド倍精度浮動小数点値を入れ替えられる。

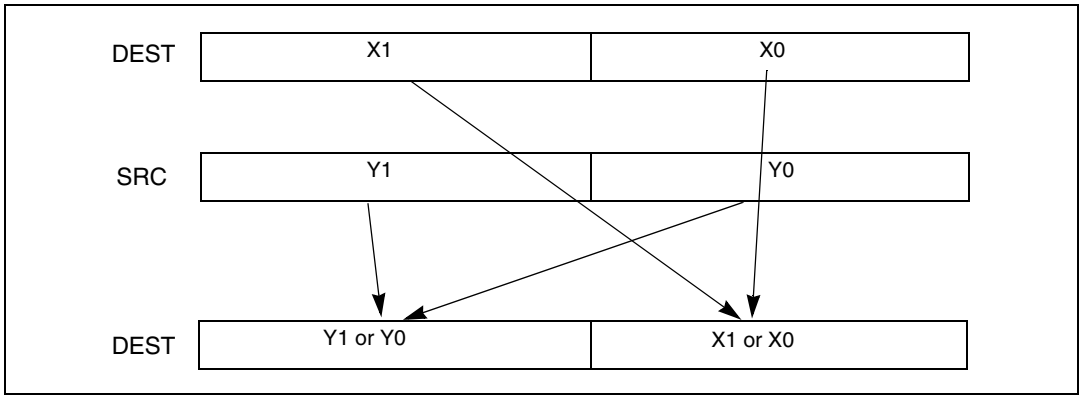


図 11-5. SHUFPD 命令のパック ド・シャッフル操作

UNPCKHPD（unpack and interleave high packed double-precision floating-point values）命令は、ソース・オペランドおよびデスティネーション・オペランドの上位の値をアンパックしてインタリーブし、その結果をデスティネーション・オペランドに格納する（図 11-6. を参照）。

UNPCKLPD（unpack and interleave low packed double-precision floating-point values）命令は、ソース・オペランドおよびデスティネーション・オペランドの下位の値をアンパックしてインタリーブし、その結果をデスティネーション・オペランドに格納する（図 11-7. を参照）。

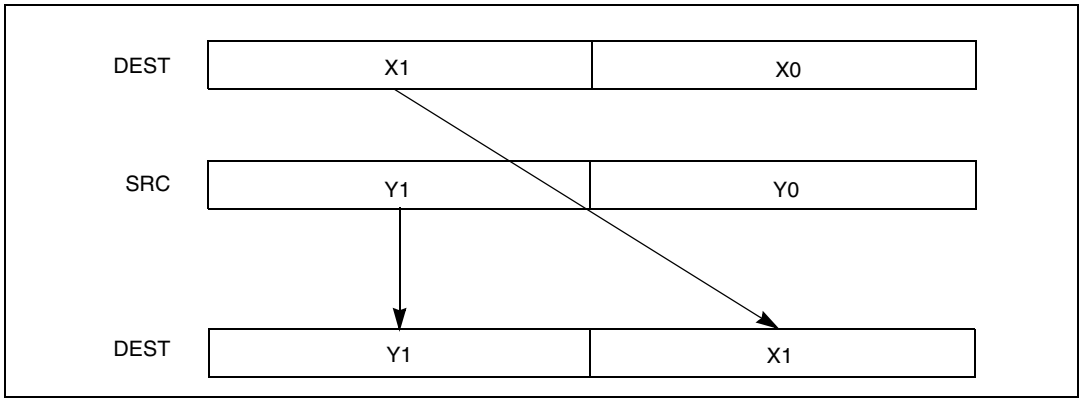


図 11-6. UNPCKHPD 命令のアンパック・ハイ操作とインタリーブ操作

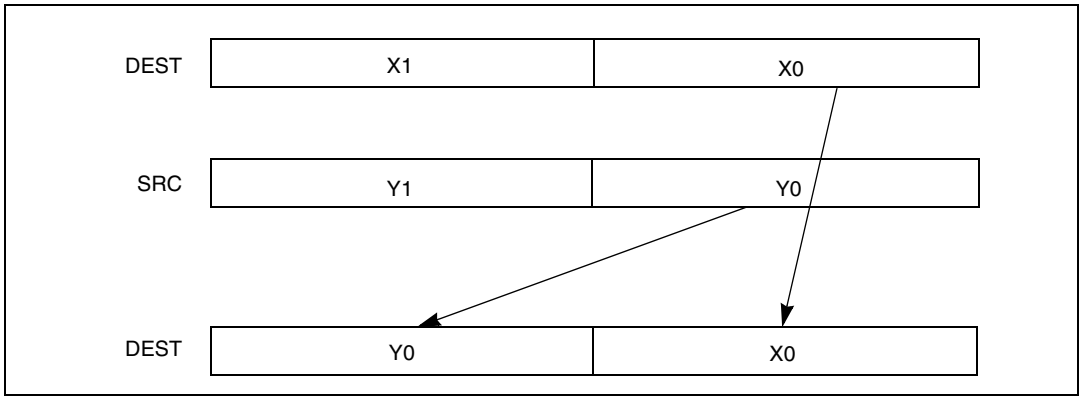


図 11-7. UNPCKLPD 命令のアンパック・ロー操作とインタリーブ操作

11.4.1.6. SSE2 変換命令

SSE2 の変換命令（図 11-8. を参照）は、以下のデータ型の間のパックド変換およびスカラ変換を実行する。

- 倍精度浮動小数点フォーマットと単精度浮動小数点フォーマット
- 倍精度浮動小数点フォーマットとダブルワード整数フォーマット
- 単精度浮動小数点フォーマットとダブルワード整数フォーマット

倍精度浮動小数点値と単精度浮動小数点値の間の変換。以下の命令は、倍精度浮動小数点フォーマットと単精度浮動小数点フォーマットの間でオペランドを変換する。操作対象となるオペランドは、XMM レジスタまたはメモリ内に置かれる。

CVTPS2PD（convert packed single-precision floating-point values to packed double-precision floating-point values）命令は、2 つのパックド単精度浮動小数点値を、2 つの倍精度浮動小数点値に変換する。

CVTPD2PS（convert packed double-precision floating-point values to packed single-precision floating-point values）命令は、2 つのパックド倍精度浮動小数点値を、2 つの単精度浮動小数点値に変換する。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。

CVTSS2SD（convert scalar single-precision floating-point values to scalar double-precision floating-point values）命令は、単精度浮動小数点値を、倍精度浮動小数点値に変換する。

CVTSD2SS（convert scalar double-precision floating-point values to scalar single-precision floating-point values）命令は、下位のパックド倍精度浮動小数点値を、単精度浮動小数点値に変換する。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。

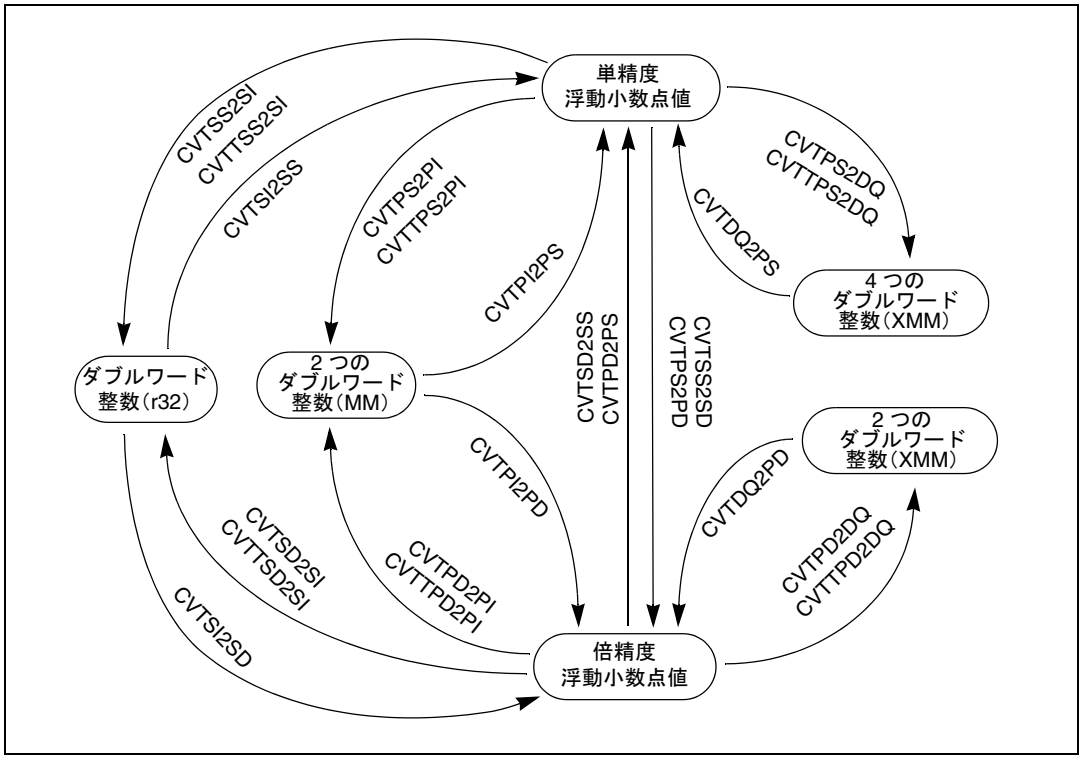


図 11-8. SSE と SSE2 の変換命令

倍精度浮動小数点値とダブルワード整数の間の変換。以下の命令は、倍精度浮動小数点フォーマットとダブルワード整数フォーマットの間でオペランドを変換する。オペランドは、XMM レジスタ、MMX® テクノロジ・レジスタ、またはメモリ内に置かれる。

CVTPD2PI（convert packed double-precision floating-point values to packed doubleword integers）命令は、2 つのパックド倍精度浮動小数点値を 2 つの符号付きパックド・ダブルワード整数に変換し、結果を MMX テクノロジ・レジスタに格納する。整数値への丸めの際に、ソース・オペランドの値は、MXCSR レジスタの丸めモードにしたがって丸められる。CVTTPD2PI（convert with truncation packed double-precision floating-point to packed doubleword integer）命令は、CVTPD2PI 命令によく似ているが、ソース・オペランドの値を整数値に丸めるときに切り捨てを使用する点が異なる。（4.8.4.2. 項「SSE および SSE2 変換命令による切り捨て」を参照）。

CVTPI2PD（convert packed doubleword integer to packed double-precision floating-point values）命令は、2 つの符号付きパックド・ダブルワード整数を、2 つの倍精度浮動小数点値に変換する。

CVTPD2DQ（convert packed double-precision floating-point values to packed doubleword integers）命令は、2 つのパックド倍精度浮動小数点値を 2 つの符号付きパックド・ダブルワード整数に変換し、結果を XMM レジスタの下位クワッドワードに格納する。整数値への丸めの際に、ソース・オペランドの値は、MXCSR レジスタの丸めモードにしたがって丸められる。CVTTPD2DQ（convert with truncate packed double-precision floating-point to packed doubleword integer）命令は、CVTPD2DQ 命令によく似ているが、ソース・オペランドの値を整数値に丸めるときに切り捨てを使用する点が異なる。（4.8.4.2. 項「SSE および SSE2 変換命令による切り捨て」を参照）。

CVTDQ2PD（convert packed doubleword integer to packed double-precision floating-point values）命令は、XMM レジスタの下位ダブルワード内の 2 つの符号付きパックド・ダブルワード整数を、2 つの倍精度浮動小数点値に変換する。

CVTSD2SI（convert scalar double-precision floating-point value to a doubleword integer）命令は、倍精度浮動小数点値をダブルワード整数に変換し、結果を汎用レジスタに格納する。整数値への丸めの際に、ソース・オペランドの値は、MXCSR レジスタで選択された丸めモードにしたがって丸められる。CVTTSD2SI（convert with truncation scalar double-precision floating-point values to doubleword integer）命令は、CVTSD2SI 命令によく似ているが、ソース・オペランドの値を整数値に丸めるときに切り捨てを使用する点が異なる（4.8.4.2. 項「SSE および SSE2 変換命令による切り捨て」を参照）。

CVTSI2SD（convert doubleword integer to scalar double-precision floating-point value）命令は、汎用レジスタ内の符号付きダブルワード整数を倍精度浮動小数点値に変換し、結果を XMM レジスタに格納する。

単精度浮動小数点フォーマットとダブルワード整数フォーマットの間の変換。これらの命令は、XMM レジスタ内のパック単精度浮動小数点値とパック・ダブルワード整数の間の変換を実行する。これらの SSE2 命令は、SSE で追加された変換命令（CVTPI2PS、CVTPS2PI、CVTTPS2PI、CVTSI2SS、CVTSS2SI、CVTTSS2SI）を補うものである。

CVTPS2DQ（convert packed single-precision floating-point values to packed doubleword integers）命令は、4 つのパック単精度浮動小数点値を、4 つの符号付きパック・ダブルワード整数に変換する。ソース・オペランドとデスティネーション・オペランドは XMM レジスタ内に置かれる。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。CVTTPS2DQ（convert with truncation packed single-precision floating-point values to packed doubleword integers）命令は、CVTPS2DQ 命令によく似ているが、ソース・オペランドの値を整数値に丸めるときに切り捨てを使用する点が異なる（4.8.4.2. 項「SSE および SSE2 変換命令による切り捨て」を参照）。

CVTDQ2PS（convert packed doubleword integers to packed single-precision floating-point values）命令は、4 つの符号付きパック・ダブルワード整数を、4 つのパック単精度浮動小数点値に変換する。ソース・オペランドとデスティネーション・オペランドは XMM レジスタ内に置かれる。変換が不正確な場合は、MXCSR レジスタで選択された丸めモードにしたがって丸められた値が返される。

11.4.2. SSE2 64 ビットおよび 128 ビット SIMD 整数命令

SSE2 では、いくつかの 128 ビット・パック整数命令が IA-32 アーキテクチャに追加された。これらの命令には、必要に応じて 64 ビット版も用意されている。128 ビット版の命令は、XMM レジスタ内のデータを操作する。64 ビット版は、MMX® テクノロジ・レジスタ内のデータを操作する。これらの命令には以下のものがある。

MOVDQA（move aligned double quadword）命令は、メモリから XMM レジスタに、XMM レジスタからメモリに、または XMM レジスタ同士の間で、ダブル・クワッドワード・オペランドを転送する。メモリアドレスは、16 バイトにアライメントされていなければならない。アライメントが合っていない場合は、一般保護例外（#GP）が発生する。

MOVDQU（move unaligned double quadword）命令は、MOVDQA 命令と同じ操作を実行するが、メモリアドレスの 16 バイト・アライメントが要求されない点が異なる。

PADDQ（packed quadword add）命令は、2 つのパック・クワッドワード整数オペランド同士または 2 つのシングル・クワッドワード整数オペランド同士を加算し、XMM レジスタもしくは MMX テクノロジ・レジスタに結果をそれぞれ格納する。この命令は、符号なしまたは符号付きの（2 の補数表記の）整数オペランドを操作する。

PSUBQ（packed quadword subtract）命令は、2つのパックド・クワッドワード整数オペランド同士または2つのシングル・クワッドワード整数オペランド同士を減算し、XMM レジスタもしくは MMX テクノロジ・レジスタに結果をそれぞれ格納する。PSUBQ 命令は、PADDQ 命令と同じように、符号なしまたは符号付きの（2 の補数表記の）整数オペランドを操作する。

PMULUDQ（multiply packed unsigned doubleword integers）命令は、符号なしダブルワード整数の乗算を実行し、クワッドワードの結果を返す。この命令には、64 ビット版と 128 ビット版がある。64 ビット版は、それぞれのソース・オペランドの下位ダブルワードに格納された 2 つのダブルワード整数を操作し、クワッドワードの結果を MMX テクノロジ・レジスタに返す。128 ビット版は、2 組のダブルワード整数のパックド乗算を実行する。この場合、各ダブルワードはソース・オペランドの第 1 ダブルワードと第3ダブルワードにパックされ、クワッドワードの結果はXMM レジスタの下位クワッドワードと上位クワッドワードに格納される。

PSHUFLW（shuffle packed low words）命令は、ソース・オペランドの下位クワッドワード内にパックされたワード整数をシャッフルして、シャッフルされた結果をデスティネーション・オペランドの下位クワッドワードに格納する。8 ビット即値オペランドで、シャッフルの順序を指定する。

PSHUFHW（shuffle packed high words）命令は、ソース・オペランドの上位クワッドワード内にパックされたワード整数をシャッフルして、シャッフルされた結果をデスティネーション・オペランドの上位クワッドワードに格納する。8 ビット即値オペランドで、シャッフルの順序を指定する。

PSHUFD（shuffle packed doubleword integers）命令は、ソース・オペランド内にパックされたダブルワード整数をシャッフルして、シャッフルされた結果をデスティネーション・オペランドに格納する。8 ビット即値オペランドで、シャッフルの順序を指定する。

PSLLDQ（shift double quadword left logical）命令は、ソース・オペランドの内容を、即値オペランドで指定されたバイト数だけ左にシフトする。空いた下位バイトはクリア（0 に設定）される。

PSRLDQ（shift double quadword right logical）命令は、ソース・オペランドの内容を、即値オペランドで指定されたバイト数だけ右にシフトする。空いた上位バイトはクリア（0 に設定）される。

PUNPCKHQDQ（Unpack high quadwords）命令は、ソース・オペランドの上位クワッドワードとデスティネーション・オペランドの上位クワッドワードをインターリーブして、結果をデスティネーション・レジスタに書き込む。

PUNPCKLQDQ（Unpack low quadwords）命令は、ソース・オペランドの下位クワッドワードとデスティネーション・オペランドの下位クワッドワードをインターリーブして、結果をデスティネーション・レジスタに書き込む。

MMX テクノロジ・レジスタから XMM レジスタへのデータ転送用に、2 つの新しい SSE 命令が追加された。

MOVQ2DQ（move quadword integer from MMX to XMM registers）命令は、MMX テクノロジ・ソース・レジスタ内のクワッドワード整数を XMM デスティネーション・レジスタに転送する。

MOVDQ2Q（move quadword integer from XMM to MMX registers）命令は、XMM テクノロジ・ソース・レジスタ内の下位クワッドワード整数を MMX テクノロジ・デスティネーション・レジスタに転送する。

11.4.3. 128 ビット SIMD 整数拡張命令

MMX® テクノロジおよび SSE（PSHUFW 命令を除く）で導入されたすべての 64 ビット SIMD 整数命令は、XMM レジスタ内の 128 ビット・パックド整数オペランドを操作できるように、SSE2 で拡張された。128 ビット版の命令に適用される、パックド・オペランドに関する SIMD 規則は、64 ビット版の命令と同じものである。例えば、PADDB 命令の 64 ビット版が 8 個のパックドバイトを操作する場合、その命令の 128 ビット版は、16 個のパックドバイトを操作する。

11.4.4. キャッシュ制御命令およびメモリアクセス順序命令

SSE2 では、プログラムによってキャッシュ処理とロード/ストア操作をきめ細かく制御できる。これらの命令について、以下の各項で説明する。

11.4.4.1. フラッシュのキャッシュ・ライン

CLFLUSH（flush cache line）命令は、指定されたリニアアドレスに対応するキャッシュ・ラインへの書き込みと無効化を行う。無効化は、プロセッサのキャッシュ階層のすべてのレベルに対して適用され、キャッシュのコヒーレンシ・ドメイン全体にブロードキャストされる。

CLFLUSH 命令は SSE2 で導入された命令であるが、SSE2 をサポートしない IA-32 プロセッサでも実行できる。CLFLUSH 命令には独自の機能ビット（EDX レジスタのビット 19）があり、SSE2 とは別にサポートの有無を検出できる。

11.4.4.2. キャッシュ制御命令

次の 4 つの命令は、非テンポラルなヒントを使用して、XMM レジスタおよび汎用レジスタからメモリにデータをストアする。非テンポラルなヒントは、可能な限りデータをキャッシュ階層内に書き込まずにメモリにストアするようにプロセッサに指示する（非テンポラルなストアとヒントについての詳細は、10.4.6.2. 項「テンポラルなデータと非テンポラルなデータのキャッシュ処理」を参照のこと）。

MOVNTDQ（store double quadword using non-temporal hint）命令は、非テンポラルなヒントを使用して、パックド整数データを XMM レジスタからメモリにストアする。

MOVNTPD（store packed double-precision floating-point values using non-temporal hint）命令は、非テンポラルなヒントを使用して、パックド倍精度浮動小数点データを XMM レジスタからメモリにストアする。

MOVNTI（store doubleword using non-temporal hint）命令は、非テンポラルなヒントを使用して、整数データを汎用レジスタからメモリにストアする。

MASKMOVDQU（store selected bytes of double quadword）命令は、書き込むバイトをバイトマスクで個々に選択した上で、選択したバイト整数を XMM レジスタからメモリにストアする。メモリ・ロケーションのアライメントが自然境界に合っている必要はない。この命令も非テンポラルなヒントを使用する。

11.4.4.3. メモリアクセス順序命令

SSE2 では、SSE で導入された SFENCE 命令に関連する命令として、2 つの新しいフェンス命令（LFENCE と MFENCE）が追加された。

LFENCE 命令は、ロード操作のメモリフェンスを設定する。この命令は、2 つのロードの間の順序付けを保証し、見込み的なロードがロードフェンスを超えることを防ぐ（つまり、ロードフェンスより前に指定されたすべてのロードが実行されるまで、見込み的なロードの実行は許可されない）。

MFENCE 命令は、ロード操作とストア操作のメモリフェンスを設定することによって、LFENCE 命令と SFENCE 命令の機能を組み合わせたものである。この命令は、フェンスより前に指定されたすべてのロードとストアが、フェンスより後に実行されるロードまたはストアより前に、グローバルに参照可能になることを保証する。

11.4.4.4. PAUSE

PAUSE 命令は、インテル® Pentium® 4 プロセッサまたはインテル® Xeon™ プロセッサ上で実行される USE 命令は、インテル Pentium 4 プロセッサ上で実行される「時間待ち（spin-wait）ループ」のパフォーマンスを改善するために用意されている。
インテル Pentium 4 プロセッサでは、この命令は時間待ちループの実行中のプロセッ

サの消費電力を軽減するメリットもある。時間待ちループのコード・シーケンスには、常に PAUSE 命令を使用することを推奨する。

11.4.5. 分岐ヒント

SSE2 は、プロセッサに分岐ヒントを与えるための 2 つの命令プリフィックス（2EH と 3EH）を指定している（『IA-32 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、中巻 A』の第 2 章の「命令プリフィックス」を参照）。これらのプリフィックスは、J*cc* 命令と組み合わせて使用しなければならない。また、これらのプリフィックスは、マシン・コード・レベルでのみ使用できる（つまり、分岐ヒントにはニーモニックがない）。

11.5. SSE、SSE2、SSE3 の例外

SSE、SSE2、SSE3 は、次の 2 つの一般的なタイプの例外を生成する。

- 非数値例外
- SIMD 浮動小数点例外[1]

SSE、SSE2、SSE3 は、他の IA-32 アーキテクチャ命令と同じ種類のメモリアクセス例外と非数値例外を生成する。既存の例外ハンドラは、コードの修正なしで、これらの例外を一般的に処理することができる。SSE と SSE2 で生成される非数値例外のリストと、これらの例外の処理のガイドラインについては、『IA-32 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、下巻』の第 12 章「SSE と SSE2 のシステム・プログラミング」の「SSE と SSE2 で生成される例外の非数値例外ハンドラ」を参照のこと。

SSE、SSE2、SSE3 は、パックド整数演算では数値例外を生成しないが、パックド単精度および倍精度浮動小数点演算では数値例外（SIMD 浮動小数点例外）を生成する。これらの SIMD 浮動小数点例外は、2 進浮動小数点演算に関する IEEE 規格 754 に定義されており、x87 FPU 命令で生成される例外と同じものである。これらの例外については、11.5.1. 項「SIMD 浮動小数点例外」を参照のこと。

11.5.1. SIMD 浮動小数点例外

SIMD 浮動小数点例外とは、パックドまたはスカラ浮動小数点オペランドを操作する SSE、SSE2、SSE3 によって発生する例外である。

1. SSE3 の FISTTP 命令では、SIMD 浮動小数点例外は生成されないが、x87 FPU 浮動小数点例外は生成されることがある。

SIMD 浮動小数点例外には、次の 6 つのクラスがある。

- 無効操作（#I）
- ゼロ除算（#Z）
- デノーマル・オペランド（#D）
- 数値オーバーフロー（#O）
- 数値アンダーフロー（#U）
- 不正確結果（精度）（#P）

これらの例外はすべて（デノーマル・オペランド例外を除いて）IEEE 規格 754 に定義されている。これらの例外は、x87 浮動小数点命令で生成される例外と同じものである。それぞれの例外の内容と発生条件については、4.9. 節「浮動小数点例外の概要」を参照のこと。以下の各項では、SSE、SSE2、SSE3 の命令と実行環境で、これらの例外がどのように実装されているかについて説明する。

すべての SIMD 浮動小数点例外は正確であり、命令の実行の完了後直ちに発生する。

MXCSR レジスタには、6 つの例外条件のそれぞれに対応するフラグ（IE、DE、ZE、OE、UE、PE）とマスクビット（IM、DM、ZM、OM、UM、PM）がある（図 10-3. を参照）。マスクビットは、LDMXCSR 命令または FXRSTOR 命令によって設定される。マスクビットとフラグビットは、STMXCSR 命令または FXSAVE 命令によって読み取られる。

コントロール・レジスタ CR4 の OSXMMEXCEPT フラグ（ビット 10）は、オペレーティング・システムが設定するフラグであり、オペレーティング・システムが SIMD 浮動小数点例外のソフトウェア例外ハンドラをサポートするかどうかを指定する。このフラグは、SIMD 浮動小数点例外の処理方法を制御する。マスクされていない SIMD 浮動小数点例外が発生したとき、OSXMMEXCEPT フラグがセットされている場合は、プロセッサは SIMD 浮動小数点例外（#XF）を生成して、ソフトウェア例外ハンドラを起動する。OSXMMEXCEPT ビットがクリアされている場合は、プロセッサは、最初に SIMD 浮動小数点例外条件を検出した SSE または SSE2 で、無効オペコード例外（#UD）を生成する。11.6.2. 項「SSE と SSE2 のサポートのチェック」を参照のこと。

11.5.2. SIMD 浮動小数点例外条件

以下の各項では、SIMD 浮動小数点例外を発生させる条件と、各例外がマスクされているときにその例外条件が検出された場合のプロセッサの応答について説明する。

1 つの命令に対して複数の浮動小数点例外条件が検出された場合の例外の優先規則については、4.9.2. 項「浮動小数点例外の優先順位」を参照のこと。

11.5.2.1. 無効操作例外（#I）

浮動小数点無効操作例外（#I）は、無効な算術オペランドに対して発生する。無効操作例外のフラグ（IE）ビットは、MXCSR レジスタのビット 0 である。マスク（IM）ビットは、MXCSR レジスタのビット 7 である。

無効操作例外がマスクされている場合は、プロセッサは、実行される操作に基づいて、QNaN、QNaN 浮動小数点不定値、整数不定値、またはいずれかのソース・オペランドをデスティネーション・オペランドに返すか、EFLAGS をセットする。デスティネーション・オペランドに値を返す場合は、この値が、命令によって指定されたデスティネーション・レジスタを上書きする。表 11-1. に、命令でプロセッサが検出する無効算術演算と、これらの操作に対するマスク応答を示す。

表 11-1. 無効な算術演算に対する SSE と SSE2 のマスク応答

条件	マスク応答
SNaN オペランドに対する、ADDPS、ADDSS、ADDPD、ADDSD、SUBPS、SUBSS、SUBPD、SUBSD、MULPS、MULSS、MULPD、MULSD、DIVPS、DIVSS、DIVPD、DIVSD、ADDSUBPD、ADDSUBPD、HADDPD、HADDPS、HSUBPD または HSUBPS 命令	QNaN に変換された SNaN を返す。 詳細については、表 4-7. を参照のこと。
SNaN オペランドに対する、SQRTPS、SQRTSS、SQRTPD、または SQRTSD 命令	QNaN に変換された SNaN を返す。
負のオペランド（ゼロを除く）に対する、SQRTPS、SQRTSS、SQRTPD、または SQRTSD 命令	浮動小数点不定値を返す。
QNaN または SNaN オペランドに対する、MAXPS、MAXSS、MAXPD、MAXSD、MINPS、MINSS、MINPD、または MINSD 命令	第 2 のソース・オペランドの値を返す。
QNaN または SNaN オペランドに対する、CMPPS、CMPSS、CMPPD、または CMPSD 命令	すべて 0 のマスクを返す（ただし、「等しくない」、「順序化不可能」、「より小さくない」、または「より小さくなく等しくない」のプレディケートを使用した場合は、すべて 1 のマスクを返す）。
SNaN オペランドに対する、CVTPD2PS、CVTSD2SS、CVTPS2PD、CVTSS2SD 命令	QNaN に変換された SNaN を返す。
QNaN または SNaN オペランドに対する、COMISS または COMISD 命令	EFLAGS の値を「比較不能」に設定する。
逆の符号を持つ無限大同士の加算、または同じ符号を持つ無限大同士の減算	QNaN 浮動小数点不定値を返す。
∞ × 0 の乗算	QNaN 浮動小数点不定値を返す。
(0/0) または (∞ / ∞) の除算	QNaN 浮動小数点不定値を返す。
CVTPS2PI、CVTTPS2PI、CVTSS2SI、CVTTSS2SI、CVTPD2PI、CVTSD2SI、CVTPD2DQ、CVTTPD2PI、CVTTSD2SI、CVTTPD2DQ、CVTPS2DQ、または CVTTPS2DQ 命令による整数への変換時に、ソースレジスタの値が、NaN、∞、または表現可能な範囲を超えている場合	整数不定値を返す。

無効操作例外がマスクされていない場合は、ソフトウェア例外ハンドラが起動され（11.5.4. 項「ソフトウェアによる SIMD 浮動小数点例外の処理」を参照。）各オペランドは変更されない。

通常は、1 つ以上のソース・オペランドが QNaN である（いずれも SNaN やサポートされていないフォーマットではない）場合は、無効操作例外は生成されない。ただし、この規則は、COMISS 命令と COMISD 命令には適用されない。また、CMPPS、CMPSS、CMPPD、CMPSD の各命令にも適用されない（プレディケートが、より小さい、より小さいか等しい、より小さくない、より小さくなく等しくないの場合）。これらの命令では、QNaN ソース・オペランドがあると、無効操作例外が生成される。

無効操作例外は、ゼロ・フラッシュ・モードの影響を受けない。

11.5.2.2. デノーマル・オペランド例外（#D）

算術演算命令がデノーマル・オペランドを操作しようとすると、プロセッサはデノーマル・オペランド例外を通知する。デノーマル・オペランド例外のフラグ（DE）ビットは、MXCSR レジスタのビット 1 である。マスク（DM）ビットは、MXCSR レジスタのビット 8 である。

CVTPI2PD、CVTPD2PI、CVTTPD2PI、CVTDQ2PD、CVTPD2DQ、CVTTPD2DQ、CVTSI2SD、CVTSD2SI、CVTTSD2SI、CVTPI2PS、CVTPS2PI、CVTTPS2PI、CVTSS2SI、CVTTSS2SI、CVTSI2SS、CVTDQ2PS、CVTPS2DQ、CVTTPS2DQ 変換命令は、デノーマル例外を通知しない。また、RCPSS、RCPPS、RSQRTSS、RSQRTPS 命令も、デノーマル例外を通知しない。

MXCSR レジスタのデノーマル・ゼロ・フラグ（ビット 6）は、デノーマル・オペランド例外処理のための追加オプションを提供する。このフラグがセットされている場合、デノーマル・ソース・オペランドは、自動的にソース・オペランドと同じ符号の 0 に変換される（10.2.2.4. 項「デノーマル・ゼロ」を参照）。

デノーマル例外についての詳細は、4.9.1.2. 項「デノーマル・オペランド例外（#D）」を参照のこと。マスクされていない例外の処理については、11.5.4. 項「ソフトウェアによる SIMD 浮動小数点例外の処理」を参照のこと。

11.5.2.3. ゼロ除算例外（#Z）

DIVPD または DIVSD 命令で、ゼロでない有限数オペランドを 0 で割ろうとすると、プロセッサはゼロ除算例外を報告する。ゼロ除算例外のフラグ（ZE）ビットは、MXCSR レジスタのビット 2 である。マスク（ZM）ビットは、MXCSR レジスタのビット 9 である。

ゼロ除算例外についての詳細は、4.9.1.3. 項「ゼロ除算例外（#Z)」を参照のこと。マスクされていない例外の処理については、11.5.4. 項「ソフトウェアによる SIMD 浮動小数点例外の処理」を参照のこと。

ゼロ除算例外は、ゼロ・フラッシュ・モードの影響を受けない。

11.5.2.4. 数値オーバーフロー例外（#O）

算術演算命令の結果を丸めた値が、デスティネーション・オペランドで許される最大の有限値を超えた場合、プロセッサは数値オーバーフロー例外を報告する。この例外は、ADDPS、ADDSS、ADDPD、ADDSD、SUBPS、SUBSS、SUBPD、SUBSD、MULPS、MULSS、MULPD、MULSD、DIVPS、DIVSS、DIVPD、DIVSD、CVTPD2PS、CVTSD2S、ADDSUBPD、ADDSUBPS、HADDPD、HADDPS、HSUBPD、HSUBPS 命令で生成される。数値オーバーフロー例外のフラグ（OE）ビットは、MXCSR レジスタのビット 3 である。マスク（OM）ビットは、MXCSR レジスタのビット 10 である。

数値オーバーフロー例外についての詳細は、4.9.1.4. 項「数値オーバーフロー例外（#O)」を参照のこと。マスクされていない例外の処理については、11.5.4. 項「ソフトウェアによる SIMD 浮動小数点例外の処理」を参照のこと。

数値オーバーフロー例外は、ゼロ・フラッシュ・モードの影響を受けない。

11.5.2.5. 数値アンダーフロー例外（#U）

算術演算命令の結果を丸めた値がデスティネーション・オペランドで許される最小の正規化有限数より小さくなったとき、数値アンダーフロー例外がマスクされていなければ、プロセッサは数値アンダーフロー例外を報告する。数値アンダーフロー例外がマスクされている場合は、アンダーフロー条件と不正確結果条件の両方が検出された場合にのみ、数値アンダーフローが報告される。この例外は、ADDPS、ADDSS、ADDPD、ADDSD、SUBPS、SUBSS、SUBPD、SUBSD、MULPS、MULSS、MULPD、MULSD、DIVPS、DIVSS、DIVPD、DIVSD、CVTPD2PS、CVTSD2SS、ADDSUBPD、ADDSUBPS、HADDPD、HADDPS、HSUBPD、HSUBPS 命令で生成される。数値アンダーフロー例外のフラグ（UE）ビットは、MXCSR レジスタのビット 4 である。マスク（UM）ビットは、MXCSR レジスタのビット 11 である。

MXCSR レジスタのゼロ・フラッシュ・フラグ（ビット 15）は、数値アンダーフロー例外処理のための追加オプションを提供する。このフラグがセットされ、数値アンダーフロー例外がマスクされている場合、極小の結果（アンダーフロー例外を発生させる結果）は、真の結果と同じ符号の 0 として返される（10.2.2.3. 項「ゼロ・フラッシュ」を参照)。

数値アンダーフロー例外についての詳細は、4.9.1.5. 項「数値アンダーフロー例外（#U）」を参照のこと。マスクされていない例外の処理については、11.5.4. 項「ソフトウェアによるSIMD浮動小数点例外の処理」を参照のこと。

11.5.2.6. 不正確結果（精度）例外（#P）

不正確結果例外（精度例外とも呼ばれる）は、演算の結果がデスティネーション・オペランドのフォーマットで正確に表現できない場合に発生する。例えば、1/3 の分数は、2 進形式では正確に表現できない。この例外は頻繁に発生し、若干の（通常は許容範囲内の）精度が失われたことを示す。この例外は、正確な算術演算を実行しなければならないアプリケーションのために用意されている。一般的に、丸められた結果は、ほとんどのアプリケーションで満足のいくものになるため、通常はこの例外はマスクされる。

不正確結果例外のフラグ（PE）ビットは、MXCSR レジスタのビット 2 である。マスク（PM）ビットは、MXCSR レジスタのビット 12 である。

不正確結果例外についての詳細は、4.9.1.6. 項「不正確結果（精度）例外（#P）」を参照のこと。マスクされていない例外の処理については、11.5.4. 項「ソフトウェアによるSIMD浮動小数点例外の処理」を参照のこと。

ゼロ・フラッシュ・モードでは、不正確結果例外が報告される。

11.5.3. SIMD 浮動小数点例外の生成

プロセッサは、パックドまたはスカラ浮動小数点命令を実行するとき、SIMD 浮動小数点例外条件を次の連続した 2 段階で検出し、報告する。

1. 計算前型の例外条件（無効オペランド、ゼロ除算、デノーマル・オペランド）の検出、報告、処理を行う。
2. 計算後型の例外条件（数値オーバーフロー、数値アンダーフロー、不正確結果）の検出、報告、処理を行う。

計算前型の例外と計算後型の例外がいずれもマスクされていない場合は、SSE または SSE2 の実行中に、プロセッサが SIMD 浮動小数点例外（#XF）を 2 回生成する可能性がある。1 回目はプロセッサが計算前型の例外を検出して処理するとき、2 回目は計算後型の例外を検出したときに発生する。

11.5.3.1. マスクされている例外の処理

すべての例外がマスクされている場合、プロセッサは、デスティネーション・オペランドにマスク結果（パックド・オペランドに対する結果）を格納し、プログラムの実行を続けることによって、検出された例外を処理する。マスク結果は、検出された例外条件によって、丸められた正規化数、符号付きの無限大、デノーマル有限数、ゼロ、QNaN 浮動小数点不定値、または QNaN になる。ほとんどの場合は、MXCSR レジスタ内の対応する例外フラグビットもセットされる。ただし、アンダーフロー条件が検出され、不正確結果は発生していない場合は、例外フラグはセットされない。

プロセッサは、パックド浮動小数点オペランドを操作する場合、それぞれのサブオペランドの計算に対してマスク結果を返し計算ごとに内部例外フラグのセットを別々に設定する。次に、プロセッサは、内部例外フラグの設定値の OR（論理和）演算を実行し、OR 演算の結果にしたがって、MXCSR レジスタの例外フラグを設定する。

例えば、表 11-9. は、ADDPS 命令の結果を示している。この例では、すべての SIMD 浮動小数点例外がマスクされている。この例では、サブオペランド X0 と Y0 の加算の前にデノーマル例外条件が検出され、X1 と Y1 の加算では例外条件は検出されず、X2 と Y2 の加算では数値オーバーフロー例外条件が検出され、サブオペランド X3 と Y3 の加算の前にもう 1 つのデノーマル例外が検出されると想定している。デノーマル例外がマスクされているため、プロセッサは、（X0 と Y0）の加算と（X3 と Y3）の加算にはデノーマル・ソース値を使用して、加算の結果をデスティネーション・オペランドにそのまま渡す。デノーマル・オペランドを使用すると、X0 と Y0 の計算の結果は正規化有限数になり、例外は検出されない。しかし、X3 と Y3 の計算の結果は極小かつ不正確になる。これによって、それに対応する内部数値アンダーフロー例外フラグと不正確結果例外フラグがセットされる。

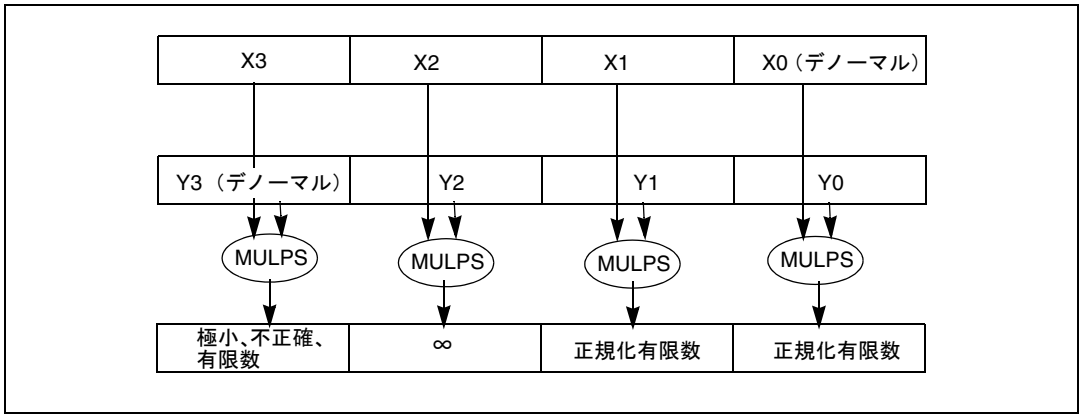


図 11-9. パックド演算のマスク応答の例

X2 と Y2 の加算については、プロセッサは浮動小数点無限大をデスティネーション・オペランドに格納し、それに対応する内部サブオペランド数値オーバーフロー・フラ

グをセットする。X1 と Y1 の加算の結果は、デスティネーション・オペランドにそのまま渡され、内部サブオペランド例外フラグはセットされない。これらの計算の後、個々のサブオペランド例外フラグ（デノーマル・オペランド、数値アンダーフロー、不正確結果、数値オーバーフロー）の OR 演算が行われ、MXCSR レジスタ内の対応するフラグがセットされる。

この計算の結果は、次のようになる。

- X0 と Y0 の加算の結果は正規化有限数になる。
- X1 と Y1 の加算の結果は正規化有限数になる。
- X2 と Y2 の加算の結果は浮動小数点無限大になる。
- X3 と Y3 の加算の結果は極小の不正確な有限数になる。
- デノーマル・オペランド、数値アンダーフロー、数値オーバーフロー、不正確結果のフラグが MXCSR レジスタ内でセットされる。

11.5.3.2. マスクされていない例外の処理

すべての例外がアンマスクされている場合、プロセッサは、次のように処理する。

1. 最初に、計算前型の例外を検出する。プロセッサは、それらの例外の OR 演算を実行して、適切な例外フラグをセットし、ソース・オペランドとデスティネーション・オペランドを変更せずに手順 2 に進む。計算前型の例外を検出しなかった場合は、手順 5 に進む。
2. コントロール・レジスタ CR4 の OSXMMEXCPT フラグ（ビット 10）をチェックする。このフラグがセットされている場合は、手順 3 に進む。このフラグがクリアされている場合は、プロセッサは無効オペコード例外（#UD）を生成し、無効オペコード例外ハンドラに対する暗黙的なコールを実行する。
3. SIMD 浮動小数点例外（#XF）を生成し、SIMD 浮動小数点例外ハンドラに対する暗黙的なコールを実行する。
4. 例外ハンドラが、計算前型の例外を発生させたソース・オペランドを修正できた場合や、プロセッサが命令を続行できるようにその例外条件をマスクできた場合は、プロセッサは手順 5 の説明にしたがって命令の実行を再開する。
5. 例外ハンドラからのリターン後（または、計算前型の例外が検出されなかった場合）、プロセッサは計算後型の例外の有無をチェックする。計算後型の例外が検出された場合は、プロセッサはそれらの例外の OR 演算を実行して、適切な例外フラグをセットし、ソース・オペランドとデスティネーション・オペランドを変更せずに、手順 2、3、4 を繰り返す。
6. 手順 4 の例外ハンドラからのリターン後（または、計算後型の例外が検出されなかった場合）、プロセッサは命令の実行を完了する。

この手順から分かるように、例外がマスクされていない場合、プロセッサは、SIMD 浮動小数点例外（#XF）を 2 回（1 回目は計算前型の例外条件を検出したとき、2 回目は計算後型の例外条件を検出したとき）生成することがある。例えば、図 11-9. の計算で、SIMD 浮動小数点例外がマスクされていないとすると、プロセッサは、デノーマル・オペランド条件に対して第 1 の SIMD 浮動小数点例外を生成し、オーバーフロー、アンダーフロー、不正確な結果条件に対して第 2 の SIMD 浮動小数点例外を生成する。

11.5.3.3. マスクされている例外とマスクされていない例外の組み合わせの処理

マスクされている例外とマスクされていない例外の両方が検出された場合は、プロセッサは、マスクされている例外とマスクされていない例外の両方の例外フラグをセットする。ただし、プロセッサは、マスクされていない計算後型の例外の検出と処理が完了し、（上記の手順 6 のように）例外ハンドラからのリターン後に命令の実行が完了するまで、マスク結果を返さない。

11.5.4. ソフトウェアによる SIMD 浮動小数点例外の処理

4.9.3. 項「浮動小数点例外ハンドラの一般的な動作」は、SIMD 浮動小数点例外ハンドラが実行する処置を示している。SSE、SSE2、SSE3 のステートは、FXSAVE 命令によってセーブされる（11.6.5. 項「SSE と SSE2 のステートのセーブとリストア」を参照）。

11.5.5. SIMD 浮動小数点例外と x87 FPU 浮動小数点例外の相互作用

SIMD 浮動小数点例外は、x87 FPU 浮動小数点例外とは無関係に生成される。SIMD 浮動小数点例外が発生しても、（CR0.NE の値に関係なく）FERR# ピンはアサートされない。また、SIMD 浮動小数点例外は、IGNNE# ピンのアサートとディアサートを無視する。

アプリケーションが、同じタスクまたはプログラム内で、SSE、SSE2、SSE3 を x87 FPU 命令と組み合わせて使用する場合は、以下の点を考慮に入れる必要がある。

- SIMD 浮動小数点例外は、x87 FPU 浮動小数点例外とは無関係に報告される。SIMD 浮動小数点例外と x87 FPU 浮動小数点例外は、別々にアンマスクできる。ただし、x87 FPU 操作と SSE/SSE2/SSE3 操作で同じ例外をアンマスクする場合は、x87 FPU 浮動小数点例外ハンドラと SIMD 浮動小数点例外ハンドラを別々に用意しなければならない。
- MXCSR レジスタで指定された丸めモードは、x87 FPU 命令には影響を与えない。同様に、x87 FPU 制御ワードで指定された丸めモードは、SSE、SSE2、SSE3 には影響を与えない。同じ丸めモードを両方に使用するには、MXCSR レジスタの丸め制御ビットと x87 FPU 制御ワードの丸め制御ビットを、同じ値に明示的に設定する必要がある。

- SSE、SSE2、SSE3 で MXCSR レジスタで設定されるゼロ・フラッシュ・モードに相当する機能は、x87 FPU には存在しない。x87 FPU との互換性を保つには、ゼロ・フラッシュ・ビットを 0 に設定する。
- SSE、SSE2、SSE3 で MXCSR レジスタで設定されるデノーマル・ゼロ・モードに相当する機能は、x87 FPU には存在しない。x87 FPU との互換性を保つには、デノーマル・ゼロ・ビットを 0 に設定する。
- x87 FPU 命令の実行中に発生した x87 FPU 例外を検出できるアプリケーションが、それに対応する SSE、SSE2、SSE3[2] の実行中に例外が発生した場合にそのことを通知されるようにするには、そのアプリケーションが SIMD 浮動小数点例外 (#XF) を処理でき、x87 FPU 制御ワード内で有効になっている例外マスクが MXCSR レジスタ内でも有効になっている必要がある。
  - マスクされた例外が SSE、SSE2、SSE3 ライブラリ・コール中に発生した場合、(例外フラグがセットされたという事実に基づいてフォルトを生成しようとして) その例外をアンマスクしても、例外は検出されない。SIMD 浮動小数点例外フラグがセットされた後で、それに対応する例外フラグをアンマスクしても、フォルトは発生しない。そのマスクされていない例外が次に発生したときに初めて、フォルトが発生する。
  - アプリケーションが、x87 FPU ステータス・ワードをチェックして、x87 FPU ライブラリ・コールの実行中にマスクされている例外のフラグがセットされたかどうかを確認する場合は、MXCSR レジスタもチェックして、SSE、SSE2、SSE3 ライブラリ・コール中にセットされたマスクされている例外フラグが発生したかどうかについても、同じように確認する必要がある。

11.6. SSE および SSE2 によるアプリケーションの作成

以下の各項では、SSE と SSE2 で導入されたデータ型と命令を使用するアプリケーション・プログラムとオペレーティング・システム・コードを作成する際のガイドラインについて説明する。ストリーミング SIMD 拡張命令とストリーミング SIMD 拡張命令 2 は、同じステートを共有し、同様の操作を実行するため、これらのガイドラインは両方の拡張命令に適用される。

『IA-32 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、下巻』の第 12 章「SSE と SSE2 のシステム・プログラミング」では、SSE と SSE2 を使用するコードを作成する際の、コンテキスト・スイッチ用のプロセッサのインターフェイスと、オペレーティング・システムに関するその他の考慮事項について説明する。

2. ここでは、SSE3 とは ADDSUBPD、ADDSUBPS、HADDPD、HADDPS、HSUBPD、HSUBPS のみのことを表している。

11.6.1. SSE と SSE2 の使用時の一般的なガイドライン

以下のガイドラインにしたがって、SSE と SSE2 を使用してプロセッサのパフォーマンスを最大限に活用することができる。

- プロセッサが SSE と SSE2 をサポートしていることを確認する。
- オペレーティング・システムが SSE と SSE2 をサポートしていることを確認する（オペレーティング・システムが SSE をサポートしているのは、SSE2 もサポートしているという意味である。また、その逆も成り立つ）。
- スタック・アライメントとデータ・アライメントの手法を使用して、データのアライメントを保ち、メモリを効率的に使用する。
- SSE と SSE2 の非テンポラルなストア命令を使用する。
- 『IA-32 インテル® アーキテクチャ最適化 リファレンス・マニュアル』の説明にしたがって、最適化とスケジューリングの手法を使用する（本巻の資料番号は、1.4. 節「参考文献」を参照）。

11.6.2. SSE と SSE2 のサポートのチェック

アプリケーションは、SSE と SSE2 を使用する前に、プロセッサがそれらの命令を搭載しており、オペレーティング・システムもそれらの命令をサポートしていることを確認する必要がある。アプリケーションは、以下の手順でこのチェックを実行する。

1. CPUID 命令を実行して、プロセッサが CPUID 命令をサポートしていることを確認する。プロセッサが CPUID 命令をサポートしていない場合は、無効オペコード例外（#UD）が発生する。
2. プロセッサが SSE および SSE2 をサポートしていることを確認する。EAX レジスタ内で引き数を 1 に設定して CPUID 命令を実行し、ビット 25（SSE）とビット 26（SSE2）が 1 にセットされていることを確認する。
3. プロセッサが FXSAVE 命令と FXRSTOR 命令をサポートしていることを確認する。EAX レジスタ内で引き数を 1 に設定して CPUID 命令を実行し、ビット 24（FXSR）が 1 にセットされていることを確認する。
4. オペレーティング・システムが FXSAVE 命令と FXRSTOR 命令をサポートしていることを確認する。MOV 命令を実行して、コントロール・レジスタ CR4 の内容を読み取り、CR4 のビット 9（OSFXSR ビット）が 1 にセットされていることを確認する。
5. オペレーティング・システムが SIMD 浮動小数点例外の処理をサポートしていることを確認する。MOV 命令を実行して、コントロール・レジスタ CR4 の内容を読み取り、CR4 のビット 10（OSXMMEXCPT ビット）が 1 にセットされていることを確認する。

注記

コントロール・レジスタ CR4 の OSFXSR ビットと OSXMMEXCPT ビットは、オペレーティング・システムによって設定されなければならない。プロセッサには、オペレーティング・システムが FXSAVE 命令と FXRSTOR 命令をサポートしているかどうか、また SIMD 浮動小数点例外の処理をサポートしているかどうかを検出する他の方法はない。

6. x87 FPU のエミュレーションが無効にされていることを確認する。MOV 命令を実行して、コントロール・レジスタ CR0 の内容を読み取り、CR0 のビット 2（EM ビット）が 0 にセットされていることを確認する。

プロセッサが、サポートされていない SSE または SSE2 を実行しようとすると、無効オペコード例外（#UD）が発生する。

11.6.3. MXCSR レジスタの DAZ フラグのチェック

MXCSR レジスタのデノーマル・ゼロ・フラグは、初期のものを除いて大部分のインテル® Pentium® 4 プロセッサとインテル® Xeon™ プロセッサで使用可能である。MXCSR レジスタの DAZ フラグの有無をチェックするには、以下の手順を実行する。

1. メモリ内に 512 バイトの FXSAVE 領域を設定する。
2. FXSAVE 領域をすべて 0 にクリアする。
3. クリアされた FXSAVE 領域の第 1 バイトのアドレスをソース・オペランドとして、FXSAVE 命令を実行する。FXSAVE 命令と FXSAVE イメージのレイアウトについては、『IA-32 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、中巻 A』の第 3 章の「FXSAVE - x87 FPU、MMX®、SSE、および SSE2 ステートの保存」を参照のこと。
4. FXSAVEイメージ内のMXCSR_MASKフィールド（バイト28～31）の値をチェックする。
   - MXCSR_MASK フィールドの値が 00000000H になっている場合は、DAZ フラグとデノーマル・ゼロ・モードはサポートされていない。
   - MXCSR_MASK フィールドの値が 0 でなく、ビット 6 がセットされている場合は、DAZ フラグとデノーマル・ゼロ・モードがサポートされている。

DAZ フラグがサポートされていない場合は、ビット 6 は予約ビットになり、このビットに 1 を書き込もうとすると、一般保護例外（#GP）が発生する。MXCSR レジスタに書き込む際に一般保護例外の発生を防ぐための一般的なガイドラインについては、11.6.6. 項「MXCSR レジスタへの書き込みのガイドライン」を参照のこと。

11.6.4. SSE および SSE2 の初期化

SSE および SSE2 のステートは、XMM レジスタと MXCSR レジスタに格納される。プロセッサのハードウェア・リセット時には、このステートは次のように初期化される（表 11-2. を参照）。

- すべての SIMD 浮動小数点例外はマスクされる（MXCSR レジスタのビット 7 ～ 12 は 1 に設定される）。
- すべての SIMD 浮動小数点例外フラグはクリアされる（MXCSR レジスタのビット 0 ～ 5 は 0 に設定される）。
- 丸め制御は、直近値への丸めに設定される（MXCSR レジスタのビット 13 とビット 14 は 00B に設定される）。
- ゼロ・フラッシュ・モードは無効にされる（MXCSR レジスタのビット 15 は 0 に設定される）。
- デノーマル・ゼロ・モードは無効にされる（MXCSR レジスタのビット 6 は 0 に設定される）。デノーマル・ゼロ・モードがサポートされていない場合は、このビットは予約ビットになり、初期設定時には 0 に設定される。
- 各 XMM レジスタはクリアされる（すべて 0 に設定される）。

表 11-2. 電源投入後 / リセットまたは INIT の実行後の SSE と SSE2 のステート

レジスタ	電源投入またはリセット	INIT
XMM0 ～ XMM7	+0.0	変更なし
MXCSR	1F80H	変更なし

INIT# ピンのアサートによってプロセッサがリセットされた場合は、SSE と SSE2 のステートは変更されない。

11.6.5. SSE と SSE2 のステートのセーブとリストア

FXSAVE 命令は、 x 87 FPU、MMX® テクノロジ、SSE、SSE2 のステート（MXCSR レジスタと 8 個の XMM レジスタの内容）を、512 バイトのメモリブロックにセーブする。FXRSTOR 命令は、セーブされた SSE と SSE2 ステートを、メモリからリストアする。512 バイトのステート・ブロックのレイアウトについては、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の FXSAVE 命令を参照のこと。

FXSAVE 命令と FXRSTOR 命令は、SSE と SSE2 ステートのセーブとリストアを行うだけでなく、x87 FPU のステートのセーブとリストアも実行する。これは、x87 FPU データレジスタとして別名定義される MMX テクノロジ・レジスタにも、MMX テクノロジ・ステートが保存、格納されるためである。以下の場合は、コードの効率性を高め

るために、FSAVE/FNSAVE 命令と FRSTOR 命令を、FXSAVE 命令と FXRSTOR 命令で置き換えることが望ましい。

- マルチタスク環境でコンテキスト・スイッチが行われる場合
- 割り込みハンドラおよび例外ハンドラに対するコールとリターンの場合

ただし、x87 FPU 計算と MMX テクノロジ計算の間で（コンテキスト・スイッチや、割り込み / 例外に対するコールなしに）コードが切り替えられる場合は、FSAVE/FNSAVE 命令と FRSTOR 命令を使用する方が、FXSAVE 命令と FXRSTOR 命令を使用するより効率的である。

11.6.6. MXCSR レジスタへの書き込みのガイドライン

MXCSR レジスタにはいくつかの予約ビットがあり、これらのビットに 1 を書き込もうとすると、一般保護例外（#GP）が生成される。ソフトウェアがこれらの予約ビットを識別するために、MXCSR_MASK 値が用意されている。ソフトウェアは、このマスク値を次の手順で確認できる。

1. メモリ内に 512 バイトの FXSAVE 領域を設定する。
2. FXSAVE 領域をすべて 0 にクリアする。
3. クリアされた FXSAVE 領域の第 1 バイトのアドレスをソース・オペランドとして、FXSAVE 命令を実行する。FXSAVE 命令と FXSAVE イメージのレイアウトについては、『IA-32 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、中巻 A』の第 3 章の「FXSAVE - x87 FPU、MMX、SSE、SSE2 ステートの保存」を参照のこと。
4. FXSAVEイメージ内のMXCSR_MASKフィールド（バイト28～31）の値をチェックする。
   - MXCSR_MASK フィールドの値が 00000000H になっている場合は、MXCSR_MASK の値はデフォルト値の 0000FFBFH である（この値は、MXCSR レジスタのビット 6 が予約済みであることを示す。これは、このプロセッサがデノーマル・ゼロ・モードをサポートしていないという意味である）。
   - MXCSR_MASK フィールドの値が 0 でない場合は、その MXCSR_MASK の値が MXCSR_MASK として使用される。

MXCSR_MASK の値の中で 0 に設定されているすべてのビットは、MXCSR レジスタの予約ビットを示す。したがって、MXCSR_MASK の値と MXCSR レジスタに書き込まれる値の間で AND（論理和）演算を実行すれば、得られる値は、すべての予約ビットが必ず 0 に設定される。したがって、この値を MXCSR レジスタに書き込んだとき、一般保護例外が生成される可能性はなくなる。

例えば、FXSAVE イメージ内に 00000000H が返された場合は、MXCSR_MASK の値はデフォルト値の 0000FFBFH である。ソフトウェアが、MXCSR レジスタに書き込まれ

る値と 0000FFBFH の間で AND 演算を実行すれば、演算結果のビット 6（DAZ フラグ）は必ず 0 に設定される。この設定によって、デノーマル・ゼロ・モードをサポートしないプロセッサ上で一般保護例外が発生するのを防止できる。

一般保護例外を防止するには、以下の状況で、MXCSR_MASK の値と MXCSR レジスタに書き込まれる値の間で AND 演算を実行する必要がある。

- オペレーティング・システム・ルーチンが、アプリケーション・プログラムからパラメータを受け取り、（FXRSTOR または LDMXCSR 命令を使用して）その値を MXCSR レジスタに書き込む場合。
- MXCSR レジスタへの書き込みを実行するアプリケーション・プログラムを、異なる IA-32 プロセッサ上で安定して実行する必要がある場合。

MXCSR_MASK 値の中で 1 に設定されているすべてのビットは、MXCSR レジスタによってサポートされる機能を示す。したがって、MXCSR_MASK 値の各ビットは、CPUID 命令で返される機能フラグ情報と同じように、プロセッサの機能を識別する機能フラグとして扱うこともできる。

11.6.7. SSE および SSE2 と x87 FPU 命令および MMX® 命令の相互作用

XMM レジスタと x87 FPU/MMX®テクノロジ・レジスタは、別々の実行環境を表現する。したがって、SSE、SSE2、MMX 命令、x87 FPU 命令を同じコード・モジュール内で使用する場合や、各命令を使用するコード・モジュールを共存させる場合は、以下の点に注意する必要がある。

- XMM レジスタだけを操作する SSE と SSE2（パックドおよびスカラ浮動小数点命令や、128 ビット SIMD 整数命令など）は、64 ビット SIMD 整数命令または x87 FPU 命令と同じ実行ストリーム内で、制限なしに実行可能である。例えば、アプリケーションは、パックドおよびスカラ浮動小数点命令を使用して、XMM レジスタ内で大半の浮動小数点計算を実行すると同時に、x87 FPU 命令を使用して、三角関数などの超越関数計算を実行できる。同様に、アプリケーションは、パックド 64 ビット SIMD 整数演算とパックド 128 ビット SIMD 整数演算を制約なしに同時に実行できる。
- MMX テクノロジ・レジスタを操作する SSE と SSE2（CVTPS2PI、CVTTPS2PI、CVTPI2PS、CVTPD2PI、CVTTPD2PI、CVTPI2PD、MOVDQ2Q、MOVQ2DQ、PADDQ、PSUBQ 命令など）も、64 ビット SIMD 整数命令または x87 FPU 命令と同じ実行ストリーム内で実行可能である。ただし、これらの命令には、MMX テクノロジ命令と x87 FPU 命令の同時使用に関する以下の制限が適用される。
  - x87 FPU 命令から、MMX テクノロジ命令または（MMX テクノロジ・レジスタを操作する）SSE/SSE2 に移行する前に、x87 FPU ステートを保存する必要がある。

— MMX テクノロジ命令または（MMX テクノロジ・レジスタを操作する）SSE/SSE2 から、x87 FPU 命令に移行する前に、EMMS 命令を実行する必要がある。

11.6.8. SIMD 浮動小数点データ型と x87 FPU 浮動小数点データ型の互換性

SSE と SSE2 が操作する単精度および倍精度浮動小数点データ型は、x87 FPU の操作対象と同じデータ型である。ただし、SSE と SSE2 は、これらのデータ型を処理するとき、各データ型をネイティブ・フォーマット（単精度または倍精度）で操作する。これに対して、x87 FPU は、これらのデータ型を計算を行って拡張倍精度浮動小数点フォーマットに拡張し、処理の結果をメモリに書き込む前に、単精度または倍精度フォーマットに丸める。x87 FPU は、高精度のフォーマットを操作した後でその結果を低精度のフォーマットに丸めるため、同じ単精度または倍精度浮動小数点値に対して同じ操作を実行した場合、SSE および SSE2 とは多少異なる結果を返すことがある。この誤差は、仮数の最下位ビットにのみ生じる。

11.6.9. パックドおよびスカラ浮動小数点命令 / データと 128 ビット SIMD 整数命令 / データの組み合わせ

SSE と SSE2 は、パックド / スカラ浮動小数点データ型および 128 ビット SIMD 整数データ型に対する型指定操作を定義している。しかし、IA-32 プロセッサは、アーキテクチャ・レベルではこのデータ型指定を実行せず、マイクロアーキテクチャ・レベルでのみ実行する。したがって、インテル® Pentium® 4 プロセッサまたはインテル® Xeon™ プロセッサは、パックド / スカラ浮動小数点オペランドまたは 128 ビット・パックド整数オペランドをメモリから XMM レジスタにロードするとき、実際にロードされるデータと命令で指定されたデータ型が一致するかどうかをチェックしない。同様に、インテル Pentium 4 プロセッサは、XMM レジスタ内のデータの算術演算を実行するとき、操作対象となるデータと命令で指定されたデータ型が一致するかどうかをチェックしない。

一般的な規則として、SIMD 浮動小数点データ型と SIMD 整数データ型のデータ型は、アーキテクチャ・レベルでは指定されないため、コードが適切なデータ型を指定するように保証することは、プログラマ、アセンブラ、またはコンパイラの責任となる。適切なデータ型を指定しないと、予想外の結果が返されることがある。

例えば、以下のコード例では、2 つのパックド単精度浮動小数点オペランドがメモリから XMM レジスタに転送され（MOVAPS 命令を使用）、次に、これらのオペランドに対して倍精度パックド加算（ADDPD 命令を使用）が実行される。

```
movaps xmm0, [eax]   ; EAX register contains pointer to packed
                     ; single-precision floating-point operand
movaps xmm1, [ebx]
addpd  xmm0, xmm1
```

インテル Pentium 4 プロセッサとインテル Xeon プロセッサは、無効オペランド例外（#UD）を生成せず、予想される結果をレジスタ XMM0 に返す（つまり、各レジスタの上位 64 ビットと下位 64 ビットは倍精度浮動小数点値として扱われ、プロセッサはこれらの値をそのように処理する）。操作対象となるデータ型と ADDPD 命令が受け入れるデータ型が一致しないため、この命令によって SIMD 浮動小数点例外（数値オーバーフロー [#O] や無効操作 [#I] など）が生成されることがあるが、問題の実際の原因（データ型の不一致）は検出されない。

インテル Pentium 4 プロセッサは、実行される命令のデータ型指定と一致しないデータ型を含むオペランドを操作できるため、何種類かの有効な操作が可能になる。例えば、次の命令は、パックド倍精度浮動小数点オペランドをメモリからレジスタ XMM0 にロードし、マスクをレジスタ XMM1 にロードした後、XORPD 命令を使用して、レジスタ XMM0 内の 2 つのパックド値の符号ビットを反転する。

```
movapd xmm0, [eax]   ; EAX register contains pointer to packed
                     ; double-precision floating-point operand
movaps xmm1, [ebx]   ; EBX register contains pointer to packed
                     ; double-precision floating-point mask
xorpd  xmm0, xmm1    ; XOR operation toggles sign bits using
                     ; the mask in xmm1
```

この例では、XORPD 命令の代わりに、XORPS、XORSS、XORSD、または PXOR 命令を使用して、同一の正しい結果を得ることができる。しかし、オペランドのデータ型と命令のデータ型の不一致が原因で、マイクロアーキテクチャ・レベルでの命令の実行時に、レイテンシのペナルティが発生する。

転送命令のデータ型が一致しない場合にも、レイテンシのペナルティが発生する。例えば、パックド単精度オペランドをメモリから XMM レジスタに転送するときは、MOVAPS と MOVAPD のどちらの命令を使用することもできる。しかし、MOVAPD 命令を使用すると、その後に正しいデータ型の命令が XMM レジスタ内のデータを使用しようとしたとき、レイテンシのペナルティが発生する。

ただし、XMM レジスタからメモリにデータを転送する場合は、このようなレイテンシのペナルティは発生しない。

11.6.10. SSE と SSE2 のプロシージャと関数に対するインターフェイス

SSE と SSE2 は、XMM レジスタに直接アクセスすることができる。したがって、汎用レジスタ（EAX、EBX など）の使用について適用される、プロシージャと関数の間のインターフェイスに関する既存の規則はすべて、XMM レジスタの使用についても適用される。

11.6.10.1.XMM レジスタ内でのパラメータの受け渡し

XMM レジスタのステートは、プロシージャ（または境界）の境界を超えて維持される。XMM レジスタ内で、あるプロシージャから他のプロシージャにパラメータを受け渡すことができる。

11.6.10.2.プロシージャ・コールまたは関数呼び出し時の XMM レジスタステートのセーブ

XMM レジスタのステートは、FXSAVE 命令と転送命令の 2 つの方法を使用してセーブできる。FXSAVE 命令は、すべての XMM レジスタのステートを、MXCSR レジスタおよび x87 FPU レジスタのステートと共にセーブする。この命令は、通常は、タスクスイッチなど、実行環境のコンテキストを大きく変更するときに使用される。FXRSTOR 命令は、FXRSTOR 命令で保存された、XMM レジスタ、MXCSR レジスタ、x87 FPU レジスタの内容をリストアする。

XMM レジスタだけをセーブする場合や、選択した XMM レジスタだけをセーブする場合は、転送命令（MOVAPS、MOVUPS、MOVSS、MOVAPD、MOVUPD、MOVSD、MOVDQA、MOVDQU）を使用できる。これらの命令を使用して、XMM レジスタの内容をリストアすることもできる。XMM レジスタをメモリに保存するとき、または XMM レジスタをメモリからロードするときにパフォーマンスが低下しないように、適切なデータ型の転送命令を使用する必要がある。

転送命令を使用して、XMM レジスタの内容をスタック上に保存することもできる。この場合、スタック内の次の空きバイトのメモリアドレスとして、ESP レジスタ内のスタックポインタを使用できる。PUSH 命令とは異なり、転送命令では、スタックポインタは自動的にインクリメントされないことに注意する。

XMM レジスタの内容をスタックに保存する転送命令プロシージャは、ESP レジスタの値を 16 だけデクリメントする責任を負う。同様に、スタックから XMM レジスタの内容をロードする転送命令プロシージャは、ESP レジスタを 16 だけインクリメントする必要がある。XMM レジスタの内容を転送する際にパフォーマンスの低下を避けるには、適切なタイプの転送命令を使用する必要がある。

LDMXCSR 命令と STMXCSR 命令を使用して、プロシージャ・コールおよびリターン時に、MXCSR レジスタの内容のセーブとリストアを行うことができる。

11.6.10.3.プロシージャ・コールと関数呼び出しでの呼び出し元セーブの必要条件

SSE および SSE2 コードから、プロシージャ（または関数）を呼び出す場合は、呼び出し元セーブ規則を使用して、呼び出し元プロシージャのステートを保存することが望ましい。この規則によれば、レジスタの内容をプロシージャ・コールの前後で維持す

る必要がある場合は、コールを実行する前に、呼び出し元プロシージャがそのレジスタをメモリにストアしなければならない。

呼び出し元規則を使用する主な理由は、パフォーマンスの低下を防ぐことにある。XMM レジスタは、パックドまたはスカラ倍精度浮動小数点データ型、パックド単精度浮動小数点データ型、および 128 ビット・パックド整数データ型を格納できる。呼び出し先プロシージャには、呼び出し後に XMM レジスタ内のデータの型を認識する方法がないため、適切なデータ型の転送命令を使用して XMM レジスタの内容をメモリにストアできないし、または XMM レジスタの内容をメモリからもリストアできない。

11.6.9. 項「パックドおよびスカラ浮動小数点命令 / データと 128 ビット SIMD 整数命令 / データの組み合わせ」で説明したように、XMM レジスタとの間で転送されるデータ型に合わない転送命令を実行すると、命令は正常に実行されるが、大きなレイテンシが発生することがある。

11.6.11. 128 ビット SIMD 整数命令の使用時の既存の MMX® テクノロジ・ルーチンのアップデート

SSE2 では、64 ビット MMX® テクノロジ SIMD 整数命令がすべて拡張され、XMM レジスタを使用して 128 ビット SIMD 整数を操作できるようになった。拡張された 128 ビット SIMD 整数命令は、64 ビット SIMD 整数命令と同じように動作する。これによって、MMX テクノロジ・アプリケーションを簡単に移植できる。ただし、以下の点に注意する必要がある。

- データ幅の広い 128 ビット SIMD 整数命令を利用するには、MMX テクノロジ・レジスタの代わりに XMM レジスタを参照するように、MMX テクノロジ・コードを再コンパイルする必要がある。
- 16 バイトにアライメントが合っていないメモリ・オペランドを参照する計算命令は、アライメントが合っていない 128 ビット・データのロード命令（MOVUDQ）と、メモリ・オペランドの代わりにレジスタを使用する同じ計算命令で置き換える必要がある。16 バイトにアライメントが合っていないメモリ・オペランドに対して 128 ビット・パックド整数計算命令を使用すると、生成中の一般保護例外（#GP）が発生する。
- 64 ビット整数オペランド内のワードをシャッフルする PSHUFW 命令を、128 ビット・オペランド全体のワードをシャッフルするように拡張するには、PSHUFHW、PSHUFLW、PSHUFD 命令を組み合わせてエミュレーションを行う必要がある。
- ビット単位の 64 ビット・シフト命令（PSRLQ、PSLLQ）は、次のいずれかの方法で、128 ビットに拡張できる。
  - PSRLQ 命令と PSLLQ 命令を、マスクロジック操作と組み合わせて使用する。

  - （ダブル・クワッドワード・オペランドをバイト単位でシフトする）PSRLDQ　命令とPSLLDQ 命令を使用するように、コード・シーケンスを書き直す。

- 128 ビット SIMD 整数命令は、それに対応する 64 ビット SIMD 整数命令の 2 倍のデータを処理する。このため、ループカウンタをアップデートする必要がある。

11.6.12. 算術演算での分岐

SSE と SSE2 のステートには、条件コードは含まれない。パックドデータ比較命令は、条件を示すマスクを生成し、そのマスクが整数レジスタに転送される。次のコード・シーケンスは、SSE2 の算術演算の結果に基づいて、条件分岐を実行する方法を示している。

```
cmppd    XMM0, XMM1         ; generates a mask in XMM0
movmskpd EAX, XMM0          ; moves a 2 bit mask to eax
test     EAX, 0,2           ; compare with desired result
jne      BRANCH TARGET
```

COMISD 命令と UCOMISD 命令は、スカラ比較操作の結果として、EFLAGS をアップデートする。COMISD/UCOMISD 命令の直後に、条件分岐をスケジューリングできる。

11.6.13. キャッシュ・ヒント命令

SSE と SSE2 のキャッシュ制御命令を使用して、データのプリフェッチ、キャッシュ処理、ロード、およびストアを制御することができる。キャッシュ制御命令を適切に使用すれば、アプリケーションのパフォーマンスが向上する。

プロセッサのスーパースケーラ・マイクロアーキテクチャを効率的に使用するには、ストールが発生しないように、実行中のプログラムに対して安定したデータ・ストリームを供給する必要がある。PREFETCHh 命令は、データを実際に使用する前にプロセッサのキャッシュ階層内にフェッチすることで、アプリケーション・コード内の高いパフォーマンスが要求される部分で、データアクセスのレイテンシを最小限に抑える。

PREFETCHh 命令は、パフォーマンスに影響を与えるが、ユーザから見えるプログラムのセマンティクスを変更することはない。これらの命令の動作はプロセッサによって異なる。したがって、IA-32 プロセッサのモデルに合わせてコードを修正する必要がある。また、PREFETCHh 命令を必要以上に使用すると、メモリ帯域幅が浪費され、パフォーマンスが低下する。プリフェッチのヒントの使用法については、『IA-32 インテル® アーキテクチャ最適化 リファレンス・マニュアル』（本巻の資料番号は、1.4. 節「参考文献」を参照）の第 6 章「キャッシュ利用の最適化」を参照のこと。

非テンポラルなストア命令（MOVNTI、MOVNTPD、MOVNTPS、MOVNTDQ、MOVNTQ、MASKMOVQ、MASKMOVDQU）は、非テンポラルなデータをメモリに書き込むときのキャッシュ汚染を最小限に抑える（10.4.6.2. 項「テンポラルなデータと非テンポラルなデータのキャッシュ処理」、10.4.6.1. 項「キャッシュ制御命令」を参照）。これらの命令は、ストア操作時に、非テンポラルなデータがプロセッサのキャッシュに書き込まれないようにする。これらの命令の動作は、プロセッサによって異なる。したがって、これらの命令を十分に利用するには、IA-32 プロセッサのモデルに合わせてアプリケーションを修正する必要がある。

キャッシュ汚染の軽減以外にも、生成と参照の関係など、特定のデータ共有関係の下では、順序設定の緩いメモリタイプの使い方が重要である。順序設定の緩いメモリを使用すると、データの再構築の効率を向上させることができる。ただし、生成する側のルーチンが渡したいデータを、参照する側のルーチンが確実に取得するように注意する必要がある。次のような一般的なメモリ利用モデルは、順序設定の緩いストアの影響を受けることがある。

- ライブラリ関数が、順序設定の緩いメモリを使用して結果を書き込む場合。
- コンパイラが生成したコードが、順序設定の緩いメモリを使用して結果を書き込む場合。
- 手作業で作成されたコード。

データを参照する側のルーチンが、順序設定の緩いデータであることをどの程度認識しているかは、場合によって異なる。したがって、SFENCE 命令または MFENCE 命令を使用して、順序設定の緩いデータを生成するルーチンとそのデータを参照するルーチンの間の順序付けを保証する必要がある。SFENCE 命令と MFENCE 命令は、プログラムの順序でストアフェンス命令およびメモリフェンス命令に先行するすべてのストア命令が、フェンスに後続するすべてのストア命令より前に、グローバルに参照可能になることを保証する。これによって、ルーチン間の順序付けを効率的に保証できる。

11.6.14. SSE と SSE2 に対する命令プリフィックスの影響

大部分のストリーミング SIMD 拡張命令とストリーミング SIMD 拡張命令 2 には、表 11-3. に示したプリフィックスの使用のガイドラインが適用される（表 11-3. は、ストリーミング SIMD 拡張命令 3 の SIMD 整数命令と SIMD 浮動小数点命令にも適用される）。これらの表に示すように、ストリーミング SIMD 拡張命令に対する命令プリフィックスの影響には、以下の 4 種類がある。

命令プリフィックスについては、『IA-32 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、中巻 A』の第 2 章の「命令プリフィックス」の節を参照のこと。

注記

一部の SSE、SSE2、SSE3 は、長さが 2 バイトまたは 3 バイトの 2 バイト・オペコードを使用する。長さが 3 バイトの 2 バイト・オペコードは、3 つのバイトで構成される（第 1 バイトは 3 つの必須プリフィックス F2H、F3H、66H のいずれか、第 2 バイトは 0FH、第 3 バイトはオペコード・バイト）。表 11-3 に示すように、SSE、SSE2、SSE3 では、オペランド・サイズ・プリフィックスとリピート・プリフィックスは予約されている。

表 11-3. SSE、SSE2、SSE3 に対するプリフィックスの影響

プリフィックスのタイプ	SSE、SSE2、SSE3 命令の影響
アドレス・サイズ・プリフィックス（67H）	メモリ・オペランドを使用する命令の動作に影響を与える。
	メモリ・オペランドを使用しない命令で予約されており、予測不可能な動作が発生する。
オペランド・サイズ（66H）	予約済みと見なされ、予測不可能な動作が発生する。
セグメント・オーバライド（2EH、36H、3EH、26H、64H、65H）	メモリ・オペランドを使用する命令の動作に影響を与える。
	メモリ・オペランドを使用しない命令で予約されており、予測不可能な動作が発生する。
リピート・プリフィックス（F2H および F3H）	予約済みと見なされ、予測不可能な動作が発生する。
ロック・プリフィックス（0F0H）	予約済み、無効オペコード例外（#UD）が発生する。
分岐ヒント・プリフィックス（E2H および E3H）	予約済みと見なされ、予測不可能な動作が発生する。

12

ストリーミング SIMD 拡張命令 3（SSE3）による プログラミング

第 12 章
ストリーミング SIMD 拡張命令 3（SSE3）によるプログラミング

12

ハイパー・スレッディング・テクノロジに対応したインテル® Pentium® 4 プロセッサでは、ストリーミング SIMD 拡張命令 3（SSE3）が導入された。本章では、SSE3 および SSE3 を使用したアプリケーション・プログラムを作成する際に必要な内容について記載している。

12.1. SSE3 の概要

SSE3 は 13 個の命令で構成されている。13 個の命令のうち 10 個は、SSE および SSE2 でも使用される SIMD（Single Instruction Multiple Data）実行モデルをサポートする。SSE3 のうち 1 個は、x87 形式のプログラミングにおける整数への変換を向上させる。残りの 2 個（MONITOR および MWAIT）は、スレッドの同期化を向上させる。詳細については、以下を参照のこと。

- 12.3. 節「SSE3 命令」では、各 SSE3 について紹介する。
- 『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」と『IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパ では、各命令について詳しく説明する。
- 『IA-32 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、下巻』の第 12 章「SSE および SSE2 のシステム・プログラミング」では、IA-32 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、下巻 SSE、SSE2、SSE3 をオペレーティング・システム環境に統合する際のガイドラインについて説明する。

12.2. SSE3 のプログラミング環境とデータ型

SSE3 を使用するためのプログラミング環境は、図 3-1. および図 11-1. で示されたものから変更されていない。SSE3 では、新しいデータ型も追加されていない。パックド整数データ、単精度浮動小数点データ、倍精度浮動小数点データの処理には、XMM レジスタが使用される。x87 形式のプログラミングには、x87 FPU が使用される。SSE3 では、スレッドの同期化に汎用レジスタが使用される。SIMD 浮動小数点演算では、MXCSR レジスタが大きな役割を持つ。ただし、x87FPU を処理する SSE3 は、浮動小数点制御ワード（FCW）の影響を受けない。

12.2.1. SSE3 と MMX® テクノロジ、X87 FPU 環境、SSE、SSE2 の互換性

SSE3 では、IA-32 実行環境に新しいステートは導入されていない。SIMD および x87 のプログラミングの場合、XMM、MXCSR、x87 FPU、MMX の各レジスタのアーキテクチャ・ステートは、FXSAVE 命令でセーブし FXRSTOR 命令でリストアする。MONITOR 命令と MWAIT 命令は、入力に汎用レジスタを使用するが、レジスタの内容を変更することはない。

12.2.2. 水平処理と非対称処理

SSE と SSE2 の大半では、垂直処理と呼ばれるモデルを利用して SIMD データ処理を高速化している。このモデルを利用した場合、入力データ要素と出力データ要素との間のデータフローは垂直である（例については図 10-5. を参照）。SSE3 では、各出力データ要素の結果が入力データ要素の非対称処理または水平データ移動を伴う場合に SIMD 浮動小数点処理を高速化する命令が導入された。図 12-1. は、SSE3 の ADDSUBPD 命令における非対称処理を示している。図 12-2. は、SSE3 の HADDPD 命令における水平データ移動を示している。

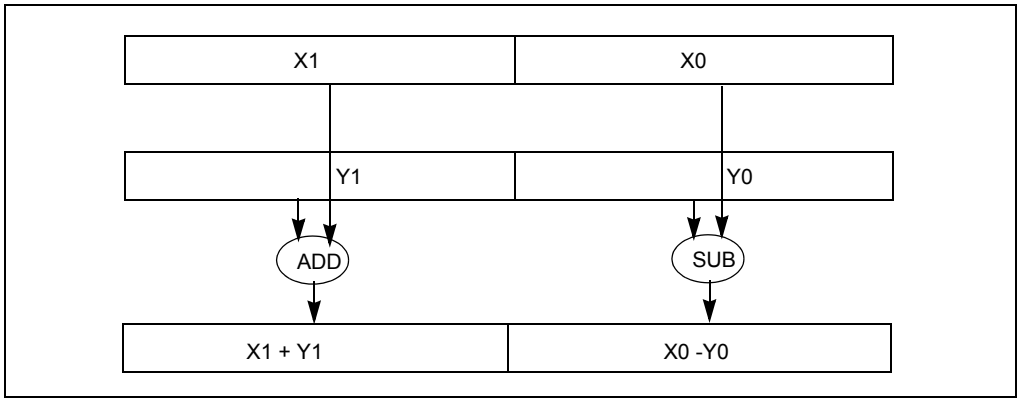


図 12-1. ADDSUBPD における非対称処理

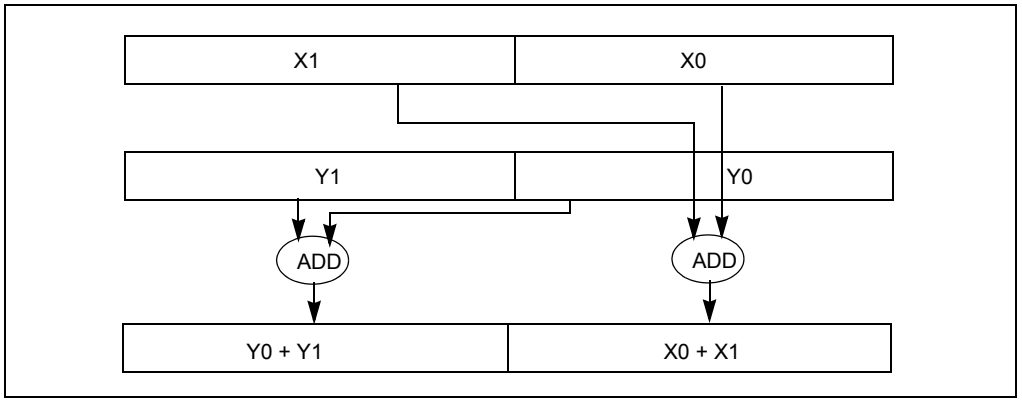


図 12-2. HADDPD における水平データ移動

12.3. SSE3 命令

SSE3 では、SSE テクノロジ、SSE2 テクノロジ、x87-FP 演算機能の性能を高める 13 個の命令が追加されている。SSE3 命令は、以下のように分類される。

- x87 FPU 命令
  - x87-FP 整数変換を向上させる命令× 1
- SIMD 整数命令
  - アライメントの合っていない専用 128 ビット・データ・ロードを実行する命令× 1
- SIMD 浮動小数点命令
  - ロード / 転送 / 複製の性能を高める命令× 3
  - パックド加算 / 減算を実行する命令× 2
  - 水平加算 / 減算を実行する命令× 4
- スレッド同期化命令
  - マルチスレッド・エージェント間での同期化を向上させる命令× 2

以下では、各命令について詳しく説明する。

12.3.1. 整数変換用の x87 FPU 命令

FISTTP 命令（切り捨てを使用して、整数をストアし x87-FP からポップ）は、FISTP と同様の動作をするが、浮動小数点制御ワード（FCW）で指定された丸めモードにかかわらず切り捨てを使用する。この命令では、丸めを使用してスタックのトップ（ST0）を整数に変換し、スタックからポップする。

FISTTP 命令は、短整数（ワード、16 ビット）、整数（ダブルワード、32 ビット）、長整数（64- ビット）の 3 種類の精度で利用できる。FISTTP を使用すれば、アプリケーションは切り捨てが必要な場合でも FCW を変更しなくてすむ。

12.3.2. アライメントの合っていない専用 128 ビット・データ・ロード用の SIMD 整数命令

LDDQU 命令は、キャッシュ・ラインの分割を防止するように設計された、アライメントの合っていない専用 128 ビット・ロードである。16 バイト・ロードのアドレスが 16 バイト境界に合っている場合、LDQQU は要求されたバイトをロードする。アドレスが 16 バイト境界に合わない場合は、アライメントの合った 16 バイト・アドレス（要求されたロードの直前のもの）から始まる 32 バイト・ブロックがロードされる。次に、そのブロックの中から、要求された 16 バイトが抽出される。

この命令には利用モデルの制限がいくつかあるが、アライメントの合っていない 128 ビット・メモリ・アクセスのパフォーマンスを大幅に向上できる。

12.3.3. ロード / 転送 / 複製の性能を高める 3 個の SIMD 浮動小数点命令

MOVSHDUP 命令は、128 ビットをロード / 転送し、2 番目と 4 番目の 32 ビット・データ要素を複製する。

- MOVSHDUP OperandA OperandB
  - OperandA（128 ビット、4 つのデータ要素）：3_a、2_a、1_a、0_a
  - OperandB（128 ビット、4 つのデータ要素）：3_b、2_b、1_b、0_b
  - 結果（OperandA にストア）：3_b、3_b、1_b、1_b

MOVSLDUP 命令は、128 ビットをロード / 転送し、1 番目と 3 番目の 32 ビット・データ要素を複製する。

- MOVSLDUP OperandA OperandB
  - OperandA（128 ビット、4 つのデータ要素）：3_a、2_a、1_a、0_a
  - OperandB（128 ビット、4 つのデータ要素）：3_b、2_b、1_b、0_b
  - 結果（OperandA にストア）：2_b、2_b、0_b、0_b

MOVDDUP 命令は、64 ビットをロード/転送し、ソースから 64 ビットを複製する。

- MOVDDUP OperandA OperandB
  - OperandA（128 ビット、2 つのデータ要素）：1_a、0_a
  - OperandB（64 ビット、1 つのデータ要素）：0_b
  - 結果（OperandA にストア）：0_b、0_b

12.3.4. パックド加算 / 減算を実行する 2 個の SIMD 浮動小数点命令

ADDSUBPS 命令は、2 個の 128 ビット・オペランドを持つ。この命令は、オペランド内の 32 ビット・データ要素の 2 番目と 4 番目のペアに対して単精度の加算を実行し、1 番目と 3 番目のペアに対して単精度の減算を実行する。

- ADDSUBPS OperandA OperandB
  - OperandA（128 ビット、4 つのデータ要素）：3_a、2_a、1_a、0_a
  - OperandB（128 ビット、4 つのデータ要素）：3_b、2_b、1_b、0_b
  - 結果（OperandA にストア）：3_a+3_b、2_a-2_b、1_a+1_b、0_a-0_b

ADDSUBPD 命令は、2 個の 128 ビット・オペランドを持つ。この命令は、クワッドワードの 2 番目のペアに対して倍精度の加算を実行し、1 番目のペアに対して倍精度の減算を実行する。

- ADDSUBPD OperandA OperandB
  - OperandA（128 ビット、2 つのデータ要素）：1_a、0_a
  - OperandB（128 ビット、2 つのデータ要素）：1_b、0_b
  - 結果（OperandA にストア）：1_a+1_b、0_a-0_b

12.3.5. 水平加算 / 減算を実行する 4 個の SIMD 浮動小数点命令

SIMD 命令の大半では、垂直的に処理が実行される。つまり、位置 i にある結果は、それぞれのオペランドの位置 i にある要素の相互作用によって生じたものである。水平加算 / 減算では、水平的に処理が実行される。つまり、同じソース・オペランド内の隣接するデータ要素を使用して、結果を求める。

HADDPS 命令は、隣接したデータ要素に対して単精度の加算を実行する。結果中の最初のデータ要素は、第 1 オペランド中の 1 番目と 2 番目の要素を足して得られたものである。同様に、2 番目のデータ要素は第 1 オペランド中の 3 番目と 4 番目の要素を、3 番目のデータ要素は第 2 オペランド中の 1 番目と 2 番目の要素を、4 番目のデータ要素は第 2 オペランド中の 3 番目と 4 番目の要素をそれぞれ足して得られたものである。

- HADDPS OperandA OperandB
  - OperandA（128 ビット、4 つのデータ要素）：3_a、2_a、1_a、0_a
  - OperandB（128 ビット、4 つのデータ要素）：3_b、2_b、1_b、0_b
  - 結果（OperandA にストア）：3_b+2_b、1_b+0_b、3_a+2_a、1_a+0_a

HSUBPS 命令は、隣接したデータ要素に対して単精度の減算を実行する。結果中の最初のデータ要素は、第1オペランド中の1番目の要素から第1オペランド中の2番目の要素を引いて得られたものである。同様に、2 番目のデータ要素は第 1 オペランド中の3番目の要素から第1オペランド中の4番目の要素を、3番目のデータ要素は第2オペランド中の1番目の要素から第2オペランド中の2番目の要素を、4番目のデータ要素は第2オペランド中の3番目の要素から第2オペランド中の4番目の要素をそれぞれ引いて得られたものである。

- HSUBPS OperandA OperandB
  - OperandA（128 ビット、4 つのデータ要素）：3_a、2_a、1_a、0_a
  - OperandB（128 ビット、4 つのデータ要素）：3_b、2_b、1_b、0_b
  - 結果（OperandA にストア）：2_b-3_b、0_b-1_b、2_a-3_a、0_a-1_a

HADDPD 命令は、隣接したデータ要素に対して倍精度の加算を実行する。結果中の最初のデータ要素は、第 1 オペランド中の 1 番目と 2 番目の要素を足して得られたものである。同様に、2番目のデータ要素は第2オペランド中の1番目と2番目の要素を足して得られたものである。

- HADDPD OperandA OperandB
  - OperandA（128 ビット、2 つのデータ要素）：1_a、0_a
  - OperandB（128 ビット、2 つのデータ要素）：1_b、0_b
  - 結果（OperandA にストア）：1_b+0_b、1_a+0_a

HSUBPD 命令は、隣接したデータ要素に対して倍精度の減算を実行する。結果中の最初のデータ要素は、第1オペランド中の1番目の要素から第1オペランド中の2番目の要素を引いて得られたものである。同様に、2 番目のデータ要素は第 2 オペランド中の1番目の要素から第2オペランド中の2番目の要素を引いて得られたものである。

- HSUBPD OperandA OperandB
  - OperandA（128 ビット、2 つのデータ要素）：1_a、0_a
  - OperandB（128 ビット、2 つのデータ要素）：1_b、0_b
  - 結果（OperandA にストア）：0_b-1_b、0_a-1_a

12.3.6. 2 個のスレッド同期化命令

MONITOR 命令は、ライトバック・ストアの監視に使用されるアドレス範囲をセットアップする。

MWAIT は、MONITOR でセットアップされたアドレス範囲へのライトバック・ストアを待機する間に、論理プロセッサを最適化された状態にすることができる。MONITOR および MWAIT では、入力に汎用レジスタを使用する必要がある。MONITOR と MWAIT によって使用されるレジスタは、適切に初期化する必要があるが、レジスタの内容がそれぞれの命令によっては変更されない。

12.4. SSE3 の例外

SSE3 は、他の IA-32 アーキテクチャ命令と同じ種類のメモリアクセス例外と非数値例外を生成する。既存の例外ハンドラは、コードの修正なしで、これらの例外を一般的に処理できる。

FISTTP は、浮動小数点例外を生成する。以下に示すように、一部の SSE3 は SIMD 浮動小数点例外も生成する。

以下の各項では、SSE3 での追加と変更について説明する。

12.4.1. DNA（Device Not Available）例外

SSE3 では、CR0.TS がセットされているときにプロセッサが SSE3 を実行しようとすると、DNA 例外（#NM）が生成される。CPUID.SSE3 がクリアされている場合に SSE3 を実行すると、CR0.TS のステートにかかわらず、無効オペコード障害が発生する。

12.4.2. 数値エラー・フラグと IGNNE#

SSE3 の大半では、CR0.NE（常時セットされているものとして処理）と IGNNE# ピンが無視される。1 つを除きすべての命令が、ベクタ 19 ソフトウェア例外を使ってエラーを報告する。その 1 つとは FISTTP であり、他の x87-FP 命令と同様の動作をする。

12.4.3. エミュレーション

x87 浮動小数点命令のエミュレーションに使われる CR0.EM ビットは、SSE3 のエミュレーションには使用できない。CR0.EM がセットされているときに SSE3 を実行すると、DNA 例外（Int 7）の代わりに無効オペコード例外（Int 6）が生成される。

12.5. SSE3 によるアプリケーションの作成

以下の各項では、SSE3 で導入された命令を使用するアプリケーション・プログラムとオペレーティング・システム・コードを作成する際のガイドラインについて説明する。

12.5.1. SSE3 の使用時の一般的なガイドライン

以下のガイドラインに従うと、SSE3 を使用する上でのメリットを最大限に活用できる。

- プロセッサが SSE3 をサポートしていることを確認する。
- オペレーティング・システムが SSE、SSE2、SSE3 をサポートしているかを確認する（オペレーティング・システムが SSE をサポートしているのは、SSE2 や、SSE3 の x87 命令および SIMD 命令もサポートしている意味である）。
- オペレーティング・システムが MONITOR と MWAIT をサポートしているかを確認する。
- 『IA-32 インテル® アーキテクチャ最適化 リファレンス・マニュアル』の説明にしたがって、最適化とスケジューリングの手法を使用する（1.4. 節「参考文献」を参照）。

12.5.2. SSE3 のサポートのチェック

アプリケーションは、SSE3 の SIMD サブセットを使用する前に、11.6.2. 節「SSE と SSE2 のサポートのチェック」で説明されている手順と、以下の追加手順を実行する必要がある。

7. プロセッサが SSE3 の SIMD 命令および x87 命令をサポートしているかを確認する。EAX レジスタ内で引き数を 1 に設定して CPUID 命令を実行し、ECX ビット 0 で返されたデータが 1 にセットされていることを確認する

SSE3 のほか SSE と SSE2 のサポートも確認すれば、SSE3 を使用する上でのソフトウェアの柔軟性が向上する。FISTTP を使用する場合、ソフトウェアは上記の手順を実行して、SSE3 がサポートされているかどうかを判断できる。

MONITOR 命令と MWAIT 命令は初回実装時、リング 0 で利用可能であり、条件によっては、0 より大きいリングレベルでも利用できる。アプリケーションは、MONITOR 命令と MWAIT 命令を使用する前に、以下の手順を実行する必要がある。

1. プロセッサが MONITOR と MWAIT をサポートしているかを確認する。EAX レジスタ内で引き数を 01H に設定して CPUID 命令を実行し、返された ECX ビット 3 が 1 にセットされていることを確認する。

2. ECX ビット 3 が 1 である場合、MONITOR と MWAIT はリング 0 で利用できる。0 より大きいリングレベルで MONITOR と MWAIT がサポートされているのを確かめる場合、アプリケーションは例 12-2. と同様のルーチンを使用して、MONITOR と MWAIT がアプリケーション・レベルで利用可能であるかどうかを確認できる。
3. MONITOR で使用される最小ラインサイズと最大ラインサイズを問い合わせる。ラインサイズは、EAX レジスタに 05H を設定して CPUID を実行すれば問い合わせが可能である。
4. MONITOR に提供されるメモリアドレス範囲がメモリタイプの条件に合っているかを確認する。

MONITOR と MWAIT は、効率的なスレッド同期化をサポートしたシステム・ソフトウェアを対象としている。詳細は、『IA-32 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、下巻』の第 12 章「SSE および SSE2 のシステム・プログラミング」を参照のこと。

例 12-1. で示されたようなコード・シーケンスを利用して、SSE3 がサポートされているかを確認すること。

例 12-1. SSE3 のサポートの確認

```
boolean SSE3_SIMD_works = TRUE;
try {
        IssueSSE3_SIMD_Instructions();
        // Use ADDSUBPD
} except (UNWIND) {
        // if we get here, SSE3 not available
        SSE3_SIMD_works = FALSE;
}
```

例 12-2. MONITOR と MWAIT のサポートの確認

```
boolean MONITOR_MWAIT_works = TRUE;
try {
        _asm {
                    xor ecx, ecx
                    xor edx, edx
                    mov eax, MemArea
                    monitor
                    }
        // Use monitor
} except (UNWIND) {
        // if we get here, MONITOR/MWAIT is not available
    MONITOR_MWAIT_works = FALSE;
}
```

12.5.3. SIMD 浮動小数点演算での FTZ と DAZ の有効化

MXCSR レジスタで FTZ フラグと DAZ フラグを有効にすると、厳格な IEEE 規格への準拠が不要な SIMD 浮動小数点演算を高速化できる場合がある。FTZ フラグは SSE をサポートした IA-32 プロセッサで利用可能であり、DAZ フラグは SSE2 をサポートした IA-32 プロセッサの大半で利用できる。

11.6.3. 項～ 11.6.6. 項で説明されている手法を利用すれば、ソフトウェアは DAZ の有無を検出し MXCSR レジスタを変更できる。

12.5.4. SSE および SSE2 と SSE3 を併用したプログラミング

SSE3 の SIMD 命令は、SIMD アプリケーションのプログラミングにおける SSE や SSE2 の利用を補完するためのものである。SSE3 を使用するアプリケーション・ソフトウェアは、SSE と SSE2 を利用できるかどうかも確認すべきである。

SSE3 の FISTTP 命令は、浮動小数点値から整数への頻繁な変換によってパフォーマンスが制限される、x87 形式のプログラミングを向上させるためのものである。このようなパフォーマンス低下は、FCW が頻繁に変更される場合に発生する。FISTTP を使用すれば、FCW にアクセスする必要性をなくすことができる。

13

入出力

第 13 章
入出力

13

インテル® アーキテクチャ・プロセッサ（IA）では、外部メモリとの間でデータ転送するほかに、入出力ポート（I/O ポート）との間でもデータ転送できる。I/O ポートは、プロセッサ上で制御ピン、データピン、アドレスピンをデコードする回路構成としてシステム・ハードウェアに組み込まれており、周辺デバイスとの通信用に構成される。I/O ポートは入力ポート、出力ポート、双方向ポートのどのタイプにも使用できる。I/O ポートには、シリアル・インターフェイス・デバイスの送信レジスタと受信レジスタとの間のデータ転送に使用されるものと、ディスク・コントローラの制御レジスタなど、周辺デバイスの制御に使用されるものがある。

本章では、インテル・プロセッサの I/O アーキテクチャについて、次の項目を説明する。

- I/O ポートのアドレス指定
- I/O 命令
- I/O の保護機構

13.1. I/O ポートのアドレス指定

プロセッサから I/O ポートにアクセスするには、次の 2 つの方法がある。

- 独立した I/O アドレス空間を使用する
- メモリマップド I/O を使用する

I/O アドレス空間を使用して I/O ポートにアクセスするには、一連の I/O 命令と特殊な I/O 保護機構を使用する。メモリマップド I/O を使用して I/O ポートにアクセスするには、プロセッサの汎用の移動命令とストリング操作命令を使用し、保護機構としてセグメンテーションまたはページングを使用する。I/O ポートは、I/O アドレス空間または物理メモリアドレス空間（メモリマップド I/O）、あるいはその両方にマッピングできる。

I/O アドレス空間を使用する利点の 1 つとして、I/O ポートへの書き込みが完了しないと、命令ストリームにおける次の命令が実行されないことが挙げられる。したがって、システム・ハードウェアの制御命令を I/O ポートに書き込むと、システム・ハードウェアが確実にその新しいステートに設定されてから次の命令が実行される。I/O 操作の詳細については、13.6. 節「I/O の順序」を参照のこと。

13.2. ハードウェアからみた I/O ポート

ハードウェアの観点からは、I/O アドレス指定はプロセッサのアドレスラインを介して処理される。P6 ファミリ・プロセッサ、インテル® Pentium® 4 プロセッサ、インテル® Xeon™ プロセッサの場合は、リクエスト・コマンド・ライン信号により、アドレスラインがメモリアドレスとしてドライブされるのか I/O アドレスとしてドライブされるのかが決まる。インテル® Pentium® プロセッサ以前の IA-32 プロセッサでは、M/IO# ピンの状態により、メモリアドレス（1 の場合）か I/O アドレス（0 の場合）かを区別する。独立した I/O アドレス空間が選択された場合は、ハードウェアによってメモリ -I/O 間のバス・トランザクションをデコードしてメモリではなく I/O ポートを選択する。プロセッサと I/O デバイスとの間のデータ転送は、データラインを使用して行われる。

13.3. I/O アドレス空間

プロセッサの I/O アドレス空間は、物理メモリのアドレス空間とは別のアドレス空間である。I/O アドレス空間は、2^{16}（64K）の個別にアドレス可能な 8 ビット I/O ポートで構成されており、各 I/O ポートには 0 ～ FFFFH のアドレスが割り振られている。0F8H ～ 0FFH の I/O ポートアドレスは予約されているので、このアドレスに I/O ポートを割り当ててはならない。I/O アドレス空間の上限 FFFFH を超えるアドレスを使用した場合の動作はインプリメントによって異なるので、詳細については、各プロセッサの『デベロッパーズ・マニュアル』を参照のこと。

連続する 2 つの 8 ビット・ポートを 1 つの 16 ビット・ポートとして、また連続する 4 つの 8 ビット・ポートを 1 つの 32 ビット・ポートとして扱うことができる。したがって、プロセッサは、I/O アドレス空間内のデバイスとの間で 8、16、32 ビット単位で転送できる。メモリ内のワードと同様に、16 ビット・ポートも偶数アドレス（0、2、4...）にアライメントを合わせることにより、16 ビットを 1 バス・サイクルで効率よく転送できる。同様に 32 ビット・ポートの場合も 4 の倍数のアドレス（0、4、8...）にアライメントを合わせておく必要がある。プロセッサは、アライメントが合っていないポートへのデータ転送も可能であるが、その場合は余分なバスサイクルが必要になるので、処理能力が低下する。

アライメントが合っていないポートにアクセスするためのバスサイクル順序は特に定義されていないので、将来発表されるインテル® アーキテクチャ・プロセッサでは変更される可能性がある。また、ハードウェアまたはソフトウェア上の理由で I/O ポートに書き込む順序が決められている場合は、その順序を明示的に指定する必要がある。例えば、アドレス 2H でワードサイズの I/O ポートに 1 ワードをロードし、次にアドレス 4H でさらに 1 ワードをロードする場合は、アドレス 2H にダブルワードを一度に書き込むのではなく、二度に分けてワード単位で書き込まなければならない。

I/O アドレス空間へのバス・サイクルに対しては、プロセッサはパリティ・エラーをマスクしない。したがって、I/O アドレス空間を介して I/O ポートにアクセスする場合には、パリティ・エラーが起こる可能性があることに注意しなければならない。

13.3.1. メモリマップド I/O

メモリ・コンポーネントのように応答する I/O デバイスに対しては、プロセッサの物理メモリアドレス空間を介してアクセスできる（図 13-1. を参照）。メモリマップド I/O を使用する場合は、物理メモリアドレスに割り当てられた I/O ポートにアクセスするのに、メモリを参照するためのプロセッサの命令のうち任意のものを使用できる。例えば、MOV 命令を使用して任意のレジスタとメモリマップド I/O ポートとの間でデータを転送し、AND、OR、TEST の各命令を使用してメモリにマッピングされた周辺デバイスの制御 / ステータス・レジスタのビットを操作できる。

メモリマップド I/O を使用する場合、I/O 操作用にマッピングされたアドレス空間に対するキャッシュを無効に設定しなければならない。インテル® Pentium® 4 プロセッサ、インテル® Xeon™ プロセッサ、および P6 ファミリ・プロセッサの場合は、メモリタイプ範囲レジスタ（MTRR）を使用して、メモリマップド I/O で使用するアドレス空間をキャッシュ不可能（UC）に設定できる。MTRR の詳細については、『IA-32 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、下巻』の第 9 章「メモリ・キャッシュ制御」を参照のこと。

インテル® Pentium® プロセッサと Intel486™ プロセッサの場合は MTRR をサポートしていないので、かわりに KEN# ピンを使用する。KEN# ピンが非アクティブ（ハイ）であれば、システムバスに出力される全アドレスのキャッシングが無効になる。KEN# ピンを使用する場合は、特定のアドレス空間に対してキャッシングを無効にするための外部アドレス・デコード・ロジックが必要になる。

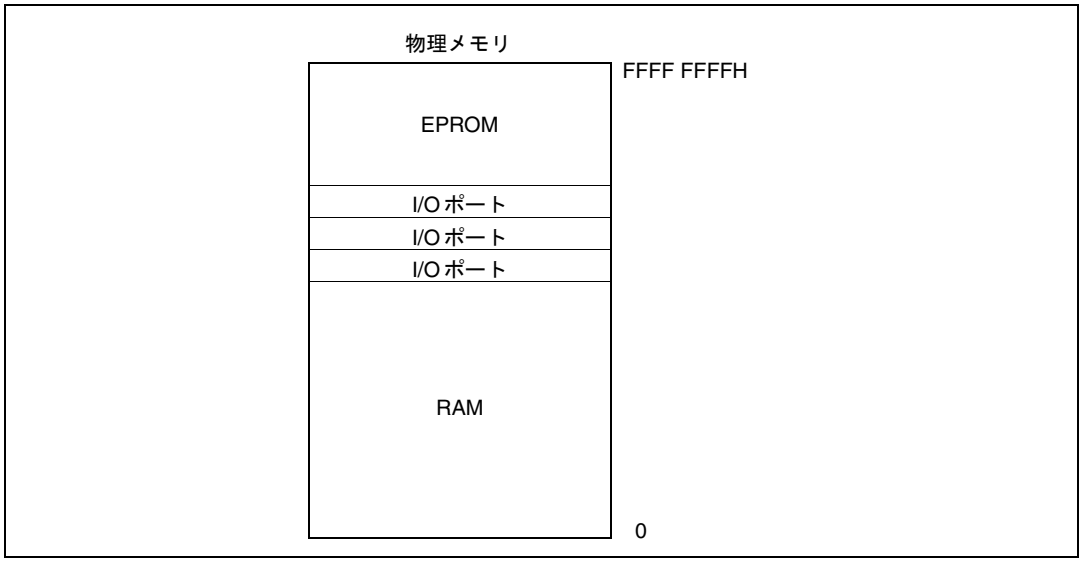


図 13-1. メモリマップド I/O

オンチップ・キャッシュを持つIAプロセッサにはすべて、ページテーブルとページ・ディレクトリ・エントリにPCD（page-level cache disable）フラグがある。このフラグを使用して、ページ単位でキャッシングを無効に設定できる。詳細については、『IA-32 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、下巻』の第3章の「ページ・ディレクトリとページ・テーブル・エントリ」を参照のこと。

13.4. I/O 命令

プロセッサのI/O命令を使用して、I/Oアドレス空間を介してI/Oポートにアクセスする（I/O命令を使用してメモリマップドI/Oポートにアクセスはできない）。I/O命令は次の2つのグループに分類できる。

- 単一項目（バイト、ワード、ダブルワード）をI/Oポートと汎用レジスタとの間で転送する命令
- ストリング項目（複数バイト、ワード、ダブルワードのストリング）をI/Oポートとメモリとの間で転送する命令

レジスタI/O命令のIN（input from I/O port）とOUT（output to I/O port）は、I/Oポートと、EAXレジスタ（32ビットI/Oの場合）、AXレジスタ（16ビットI/Oの場合）、ALレジスタ（8ビットI/Oの場合）との間でデータを転送する。I/Oポートアドレスは即値またはDXレジスタで指定する。

ストリングI/O命令のINS（input string from I/O port）とOUTS（output string to I/O port）は、I/Oポートとメモリ・ロケーションとの間でデータを転送する。I/Oポートアドレ

スはDX レジスタで指定し、メモリアドレスはソースについてはDS:ESI レジスタで、デスティネーションについてはES:EDI レジスタで指定する。

REPなどのリピート・プリフィックスを指定してINS 命令とOUTS 命令を使用すると、ストリング（ブロック）での入力あるいは出力操作を実行できる。INS 命令と OUTS 命令にリピート・プリフィックス REP を付けると、I/O ポートとメモリとの間でデータブロックを転送できる。このとき、指定のI/O ポートとメモリとの間でバイト、ワード、ダブルワードのいずれかが転送されるたびにEFLAGS レジスタの DF フラグの設定にしたがってESI レジスタまたはEDI レジスタの値がインクリメントまたはデクリメントされる。

IN、INS、OUT、OUTSの各命令の詳細については、『IA-32 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、中巻A』の第3章「命令セット・リファレンスA-M」と『IA-32 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、中巻B』の第4章「命令セット・リファレンスN-Z」を参照のこと。

13.5. 保護モードI/O

プロセッサが保護モードで動作しているときは、I/O ポートへのアクセスは次の保護機構によって制御される。

- I/O アドレス空間を介して I/O ポートにアクセスする場合は、次の保護機構で制御される。
  - EFLAGS レジスタの I/O 特権レベル（IOPL）フィールド
  - タスク・ステート・セグメント（TSS）の I/O 許可ビットマップ
- メモリマップド I/O ポートにアクセスする場合は、I/O ポートへのアクセスも通常のセグメンテーションとページングによる保護と MTRR（プロセッサがサポートしている場合のみ）によって制御される。メモリ保護の詳細については、『IA-32 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、下巻』の第4章「保護」と第9章「メモリ・キャッシュ制御」を参照のこと。

以下の各項では、I/O 命令を使用してI/O アドレス空間のI/O ポートにアクセスする場合に適用される保護機構について説明する。

13.5.1. I/O 特権レベル

I/O 保護を使用しているシステムでは、EFLAGS レジスタの IOPL フィールドにより選択した命令の使用を制限して、I/O アドレス空間へのアクセスを制御する。この保護機構により、オペレーティング・システムまたはエクゼクティブが、I/O の実行に必要な特権レベルを設定できる。通常の保護リングモデルでは、I/O アドレス空間にアクセスするには、特権レベルが 0 または 1 でなければならない。すなわち、カーネルとデバイス・ドライバは I/O を実行できるが、特権レベルがこれより低いデバイスドライバやアプリケーション・プログラムでは、I/O アドレス空間にアクセスしようとしても拒否される。したがって、アプリケーション・プログラムはオペレーティング・システムを呼び出して I/O を実行してもらわなければならない。

IN、INS、OUT、OUTS、CLI（clear interrupt-enable flag）、STI（set interrupt-enable flag）の各命令を実行するには、現在実行しているプログラムやタスクの現行特権レベル（CPL）が IOPL 以下でなければならない。これらの命令は、IOPL フィールドの値に影響されるので、「**I/O センシティブな**」命令と呼ぶ。特権レベルが IOPL より低いプログラムやタスクで I/O センシティブな命令を実行しようとすると、一般保護例外（#GP）が発生する。タスクはそれぞれが EFLAGS レジスタのコピーを持っているので、タスクごとに異なる IOPL を持つことができる。

TSS の I/O 許可ビットマップを使用すると、I/O センシティブな命令に対する IOPL の影響を無視して、特権レベルが低いプログラムやタスクでも特定の I/O ポートにアクセスできるように設定できる（13.5.2. 項「I/O 許可ビットマップ」を参照）。

プログラムやタスクが自分の IOPL を変更するには POPF 命令か IRET 命令しかないが、この命令を実行するには特権が必要である。特権レベルが 0 で実行されているプロシージャでなければ、現行 IOPL を変更できない。これより低い特権レベルのプロシージャから IOPL を変更しようとしても、例外は発生しないが IOPL は変わらない。

POPF 命令は、CLI 命令や STI 命令と同様に IF フラグのステートを変更するのにも使用できるが、POPF 命令もまた I/O センシティブである。プロシージャから POPF 命令を使用して IF フラグの値を変更するには、CPL が現行 IOPL 以下でなければならない。これより低い特権レベルのプロシージャから IF フラグを変更しようとしても、例外は発生しないが IF フラグは変わらない。

13.5.2. I/O 許可ビットマップ

I/O 許可ビットマップを使用すると、特権レベルが低いプログラムやタスク、または仮想 8086 モードで実行されているタスクが、I/O ポートに対して制限付きでアクセスすることができる。I/O 許可ビットマップは、現在実行されているタスクやプログラムの TSS 内にある（図 13-2. を参照）。I/O 許可ビットマップの先頭アドレスは、TSS の

I/Oマップ・ベース・アドレス・フィールドに入っている。I/O許可ビットマップのサイズとTSS内の位置は変更可能である。

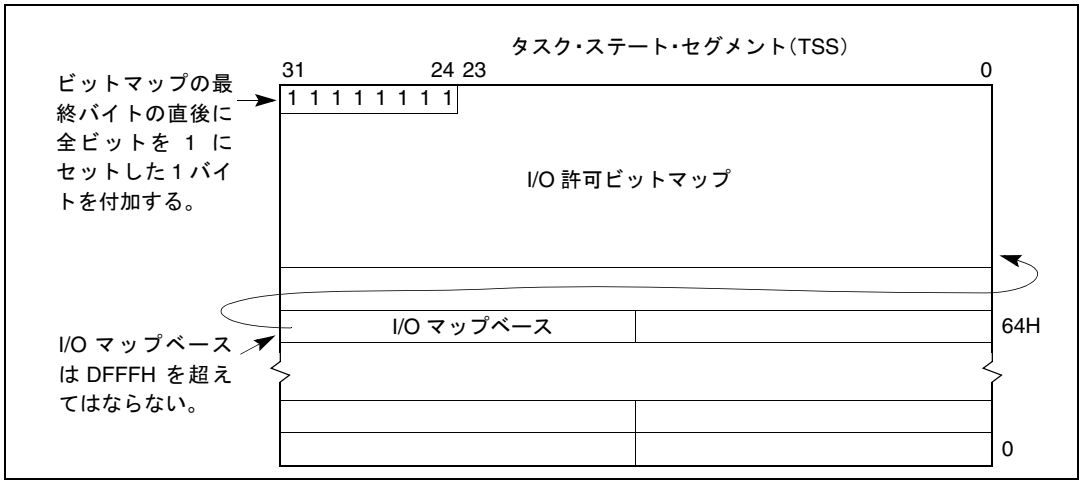


図 13-2. I/O 許可ビットマップ

タスクごとにTSSがあるので、I/O許可ビットマップもタスクごとにあることになる。したがって、個々のI/Oポートへのアクセスはタスクごとに許可できる。

保護モードの場合、CPLが現行IOPL以下であれば、プロセッサはすべてのI/O操作を許可する。CPLがIOPLより大きいか、プロセッサが仮想8086モードで動作している場合は、プロセッサはI/O許可ビットマップを調べて、特定のI/Oポートへのアクセスが許可されているかどうかをチェックする。I/O許可ビットマップ内の各ビットは、各I/Oポートのバイトアドレスに対応している。例えば、I/Oアドレス空間内のI/Oポート・アドレス29Hのポートに対する制御ビットは、I/O許可ビットマップ内の6番目のバイトのビット位置1にある。プロセッサはI/Oアクセスを許可する前に、アクセス対象のI/Oポートに対応するすべての制御ビットを調べる。例えば、ダブルワードでアクセスする場合、プロセッサは4つの隣接しあう8ビット・ポート・アドレスに対応する4つの制御ビットを調べる。調べたビットのうち1つでもセットされていれば、一般保護例外（#GP）が発生する。調べたビットがすべてゼロならば、I/O操作が許可される。

I/Oポートアドレスは必ずしもワードやダブルワードの境界にアライメントされているとは限らないので、プロセッサはI/Oポートにアクセスするたびに、I/O許可ビットマップから2バイトずつ読み込む。このとき、最大アドレスのポートにアクセスする場合に例外が発生しないように、TSSの中のI/O許可ビットマップの最後に余分に1バイトを確保しておく必要がある。このバイトは全ビットが1にセットされていなければならず、セグメント・リミット内に入っていなければならない。

I/O許可ビットマップがすべてのI/Oアドレスに対応している必要はない。I/O許可ビットマップにないI/Oアドレスは、対応するビットがセットされているものとして処理される。例えば、ビットマップのベースアドレスから10バイト目にTSSセグメント・リミットがあれば、I/O許可ビットマップは11バイト分しかなく、最初の80個のI/Oポートだけがマッピングされる。それより高いアドレスのI/Oアドレス空間にアクセスしようとすると例外が発生する。

I/O許可ビットマップのベースアドレスがTSSセグメント・リミット以上であればI/O許可ビットマップは存在しないので、CPLが現行IOPLより大きければ、あらゆるI/O命令で例外が発生する。

13.6. I/O の順序

I/Oデバイスを制御するときには、メモリ操作とI/O操作の実行順序がプログラム順序と正確に一致していなければならない場合がある。例えば、あるI/Oポートにコマンドを書き込んだ後で、別のI/OポートからI/Oデバイスのステータスを読み込む場合を考えてみる。この場合、返されるステータスは、I/Oデバイスがコマンドを受け取った**後**のステータスであって、コマンドを受け取る**前**のステータスではない。

メモリマップドI/Oを使用するときは、プロセッサがプログラム順序を守らないという状況を避けるように注意しなければならない。プロセッサは処理能力を最適化するために、キャッシュ可能なメモリの読み取りを、バッファによる書き込みより先に持ってくることが多い。内部的には、プロセッサの読み取り（キャッシュ・ヒット）とバッファによる書き込みの順序を変更することができる。したがって、メモリマップドI/Oを使用する場合は、I/O読み取り命令がその直前のメモリ書き込み命令より先に実行される可能性がある。インテル® Pentium® 4プロセッサ、インテル® Xeon™ プロセッサ、およびP6ファミリ・プロセッサの場合、メモリマップドI/Oへのアクセスをプログラム順序通りに実行させるには、MTRRを使用してメモリマップドI/Oのアドレス空間に対してキャッシュ不可能に設定する方法をお勧めする。インテル® Pentium® プロセッサとIntel486™ プロセッサの場合は、MTRRの代わりに#KENピンかPCDフラグを使用する（13.3.1.項「メモリマップドI/O」を参照）。

読み取り / 書き込みのアドレスがメモリのキャッシュ不可能領域に入っている場合は、外部的にプロセッサのピン上でメモリのI/O順序の変更は行われず、プログラム順序通りに読み取り / 書き込みが実行される。アドレス空間のメモリマップドI/O領域をキャッシュ不可能に設定することにより、I/Oデバイスへの読み取り / 書き込み順序をプログラム順序と正確に一致できる。MTRRの使用方法の詳細については、『IA-32 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、下巻』の第9章「メモリ・キャッシュ制御」を参照のこと。

I/O実行順序をプログラム順序と一致させるもう1つの方法として、CPUID命令のようなシリアル化命令を命令の間に挿入する。命令のシリアル化の詳細については、『IA-

32 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、下巻』の第7章「マルチ・プロセッサ・マネージメント」を参照のこと。

プロセッサをサポートするためのチップセット（バス・コントローラ、メモリ・コントローラ、I/O コントローラ）がキャッシュ不可能なメモリに対して書き込みを行うことによって、メモリアクセスの実行順序が変わってしまう場合があることにも注意しなければならない。このようにチップセットによってメモリアクセスの実行順序が変更されてメモリマップド I/O 処理に問題が起こる可能性がある状況においては、同期を取ってI/O操作の実行順序を守らせるようなコードを作成しなければならない。この場合もシリアル化命令を使用するとよい。

メモリマップドI/Oではなく I/Oアドレス空間を使用する場合は、次の2つの点が異なる。

- プロセッサは I/O 書き込みをバッファリングしない。したがって、I/O 操作は正確にプログラム順に実行される（メモリマップドI/Oの場合と同様に、I/O領域によってはチップセットが書き込みを行う場合がある）。
- プロセッサは I/O 命令の実行と外部バス・アクティビティとを表 13-1. のように同期させる。

表 13-1. I/O 命令のシリアル化

	プロセッサが実行を遅延させる命令		遅延が解かれる操作	
実行される命令	現在の命令	次の命令	ペンディング状態のストア	現行のストア
IN	○		○	
INS	○		○	
REP INS	○		○	
OUT		○	○	○
OUTS		○	○	○
REP OUTS		○	○	○

14

プロセッサの識別と機能の判定

第 14 章
プロセッサの識別と機能の判定

14

IA-32プロセッサ上で実行できるソフトウェアを作成する場合は、システムのプロセッサのタイプと、アプリケーションで使用できるプロセッサの機能を調べる必要がある。

14.1. CPUID 命令の使用

CPUID 命令を使用して、インテル® Pentium® M プロセッサ・ファミリ、インテル® Pentium® 4 プロセッサ・ファミリ、インテル® Xeon™ プロセッサ・ファミリ、P6 ファミリ、インテル® Pentium プロセッサ、後期の Intel486™ プロセッサのプロセッサ識別情報を取得できる。この命令は、命令を実行するプロセッサのファミリ、モデル、（一部のプロセッサでは）ブランド・ストリングを返す。また、この命令は、プロセッサが搭載している機能を示し、プロセッサのキャッシュと TLB に関する情報を返す。

EFLAGS レジスタの ID フラグ（ビット 21）は、CPUID 命令のサポート状態を示す。ソフトウェア・プロシージャによってこのフラグをセットおよびクリアできるのであれば、そのプロシージャを実行するプロセッサは CPUID 命令をサポートしている。サポートしていないプロセッサ上で CPUID 命令を実行すると、無効オペコード例外（#UD）が生成される。

プロセッサの識別情報を取得するには、EAX レジスタにソース・オペランドの値を入れ、返される情報のタイプを選択する。CPUID 命令を実行すると、選択した情報が、EAX、EBX、ECX、EDX レジスタに返される。CPUID 命令、返される値を示した表、コード例の詳細については、『IA-32 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、中巻 A』と『IA-32 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、中巻 B』の「CPUID—CPUID Identification」の項を参照のこと。

14.1.1. 使用の手引き

CPUID 命令に関する詳しいアプリケーション・ノートについては、『AP-485、インテル® プロセッサの識別と CPUID 命令』(資料番号 241618) を参照のこと。このアプリケーション・ノートには、CPUID 命令の詳細と、IA-32 プロセッサの識別に使用されるソースコードの例が記載されている。また、CPUID 命令を使用して広範囲にわたるソフトウェアの互換性を維持するためのガイドラインも記載されている。以下のガイドラインは、その中で最も重要なものであり、CPUID 命令を使用して使用可能な機能を判定するときは常に適用される。

- EAX の値を 0 にして CPUID 命令を実行する場合は、必ず最初に、EBX、EDX、ECX レジスタに "GenuineIntel," のメッセージが返されるかどうかをテストする。プロセッサがインテル純正のものでない場合は、機能識別フラグの意味が、『IA-32 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、中巻 A』と『IA-32 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、中巻 B』の「CPUID—CPUID Identification」の項の説明と異なる可能性がある。
- 機能識別フラグは個々にテストしなければならない。また、未定義ビットに依存してはならない。

14.1.2. 従来のインテル® アーキテクチャ・プロセッサの識別

CPUID 命令は、初期の Intel486™ プロセッサまでの初期の IA-32 プロセッサでは使用できない。これらのプロセッサの場合は、別の方法でプロセッサのタイプを識別する。

EFLAGS レジスタのビット 12、13（IOPL）、14（NT）、15（予約）の設定値は、インテルの 32 ビット・プロセッサと、インテル® 8086 プロセッサおよびインテル® 286 プロセッサとでは異なる。アプリケーション・プログラムで PUSHF/PUSHFD と POP/POPFD 命令を使用してこれらのビット設定値を調べることにより、プロセッサが 8086 プロセッサ、インテル 286 プロセッサ、あるいはインテルの 32 ビット・プロセッサのいずれであるかを判定できる。

- 8086 プロセッサ ー EFLAGS レジスタのビット 12 ～ 15 は常にセットされている。
- インテル 286 プロセッサ ー 実アドレスモードではビット 12 ～ 15 は常にクリアされている。
- 32 ビット・プロセッサ ー 実アドレスモードの場合は、ビット 15 は常にクリアされており、ビット 12 ～ 14 には最後にロードされた値が残っている。保護モードの場合は、ビット 15 は常にクリアされており、ビット 14 には最後にロードされた値が残っており、IOPL ビットには現行の特権レベル（CPL）の値が入っている。IOPL フィールドを変更できるのは、CPL が 0 のときのみである。

EFLAG レジスタの上記以外のビットを使用して、32 ビット・プロセッサのタイプを判別できる。

- ビット 18（AC）－ インテル® Pentium® 4 プロセッサ、インテル® Xeon™ プロセッサ、P6 ファミリ・プロセッサ、インテル® Pentium® プロセッサ、Intel486 プロセッサでのみ使用できる。Intel386™ プロセッサではこのビットをセットまたはクリアすることはできないので、これによって後期の IA-32 プロセッサと区別できる。
- ビット 21（ID）－ CPUID 命令を実行できるプロセッサかどうかを判定する。このビットをセットまたはクリアできれば、インテル Pentium 4 プロセッサ、インテル Xeon プロセッサ、P6 ファミリ・プロセッサ、インテル Pentium プロセッサ、最新の Intel486 プロセッサのいずれかである。

x87 FPU または NPX の有無を調べるには、アプリケーションで FNINIT 命令を使用して x87 FPU のステータス制御レジスタに書き込みを行い、次に FNSTENV 命令を使用して再び正しい値を読み出せるかどうかを確認する。

x87 FPU または NPX が存在することがわかったら、そのタイプも調べられる。ほとんどの場合、FPU または NPX のタイプはプロセッサのタイプで決まるが、Intel386 プロセッサは、インテル 287 プロセッサと インテル 387 プロセッサの両方の数値演算コプロセッサと互換性がある。

コプロセッサのタイプを調べるには、FINIT、FNINIT、RESET のいずれかの命令を実行して、コプロセッサが無限大をどのように表現するかを調べるとよい。インテル 287 プロセッサ数値演算コプロセッサでは、正負両方の無限大が同じ表現になるが、インテル 387 プロセッサ数値演算コプロセッサでは無限大の表現が正と負で異なる。

A

EFLAGS クロス・リファレンス

付録 A
EFLAGS クロス・リファレンス

A

表 A-2. に、EFLAGS レジスタのフラグが各命令によってどのように影響を受けるかを示す。次の表では、フラグに対する命令の影響を次の記号で示す。

表 A-1. フラグを表すコード

T	命令による、フラグのテスト
M	命令による、フラグの変更（オペランドに応じて、フラグのセットあるいはリセット）
0	命令による、フラグのリセット
1	命令による、フラグのセット
—	フラグに対する命令の影響は未定義
R	命令による、フラグの元の値にリストア
空白	命令によるフラグへの影響なし

表 A-2. EFLAGS クロス・リファレンス

命令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	—	—	—	TM	—	M					
AAD	—	M	M	—	M	—					
AAM	—	M	M	—	M	—					
AAS	—	—	—	TM	—	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					
AND	0	M	M	—	M	0					
ARPL			M								
BOUND											
BSF/BSR	—	—	M	—	—	—					
BSWAP											
BT/BTS/BTR/BTC	—	—	—	—	—	M					
CALL											
CBW											
CLC						0					
CLD									0		

表 A-2. EFLAGS クロス・リファレンス（続き）

命令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
CLI								0			
CLTS											
CMC						M					
CMOV*cc*	T	T	T		T	T					
CMP	M	M	M	M	M	M					
CMPS	M	M	M	M	M	M			T		
CMPXCHG	M	M	M	M	M	M					
CMPXCHG8B			M								
COMSID	0	0	M	0	M	M					
COMISS	0	0	M	0	M	M					
CPUID											
CWD											
DAA	—	M	M	TM	M	TM					
DAS	—	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	—	—	—	—	—	—					
ENTER											
ESC											
FCMOV*cc*			T		T	T					
FCOMI, FCOMIP, FUCOMI, FUCOMIP			M		M	M					
HLT											
IDIV	—	—	—	—	—	—					
IMUL	M	—	—	—	—	M					
IN											
INC	M	M	M	M	M						
INS									T		
INT							0			0	
INTO	T						0			0	
INVD											
INVLPG											
UCOMSID	0	0	M	0	M	M					
UCOMISS	0	0	M	0	M	M					

表 A-2. EFLAGS クロス・リファレンス（続き）

命令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
IRET	R	R	R	R	R	R	R	R	R	T	
J*cc*	T	T	T		T	T					
JCXZ											
JMP											
LAHF											
LAR			M								
LDS/LES/LSS/LFS/LGS											
LEA											
LEAVE											
LGDT/LIDT/LLDT/LMSW											
LOCK											
LODS									T		
LOOP											
LOOPE/LOOPNE			T								
LSL			M								
LTR											
MONITOR											
MWAIT											
MOV											
MOV control, debug, test	—	—	—	—	—	—					
MOVS									T		
MOVSX/MOVZX											
MUL	M	—	—	—	—	M					
NEG	M	M	M	M	M	M					
NOP											
NOT											
OR	0	M	M	—	M	0					
OUT											
OUTS									T		
POP/POPA											
POPF	R	R	R	R	R	R	R	R	R	R	
PUSH/PUSHA/PUSHF											
RCL/RCR 1	M					TM					

表 A-2. EFLAGS クロス・リファレンス（続き）

命令	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
RCL/RCR count	—					TM					
RDMSR											
RDPMC											
RDTSC											
REP/REPE/REPNE											
RET											
ROL/ROR 1	M					M					
ROL/ROR count	—					M					
RSM	M	M	M	M	M	M	M	M	M	M	M
SAHF		R	R	R	R	R					
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M					
SAL/SAR/SHL/SHR count	—	M	M	—	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SET*cc*	T	T	T		T	T					
SGDT/SIDT/SLDT/SMSW											
SHLD/SHRD	—	M	M	—	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
STR											
SUB	M	M	M	M	M	M					
TEST	0	M	M	—	M	0					
UD2											
VERR/VERRW			M								
WAIT											
WBINVD											
WRMSR											
XADD	M	M	M	M	M	M					
XCHG											
XLAT											
XOR	0	M	M	—	M	0					

B

EFLAGS 条件コード

付録 B
EFLAGS 条件コード

B

表 B-1. に、CMOV*cc*、FCMOV*cc*、J*cc*、SET*cc* の各命令でテストされる条件コードをすべて示す。条件コードとは、EFLAGS レジスタの 1 つ以上のステータス・フラグ（CF、OF、SF、ZF、PF）の設定値を調べることである。「ニーモニック」欄には、テスト条件を指定するために命令に追加するサフィックス（*cc*）を示す。「テストされる条件」欄には、「ステータス・フラグの設定」欄に示した条件の説明を示す。「命令サブコード」欄には、テスト条件を指定するためにメイン・オペコードに追加されるサフィックスを示す。

表 B-1. EFLAGS 条件コード

ニーモニック（*cc*）	テストされる条件	命令サブコード	ステータス・フラグの設定
O	Overflow	0000	OF = 1
NO	No overflow	0001	OF = 0
B NAE	Below Neither above nor equal	0010	CF = 1
NB AE	Not below Above or equal	0011	CF = 0
E Z	Equal Zero	0100	ZF = 1
NE NZ	Not equal Not zero	0101	ZF = 0
BE NA	Below or equal Not above	0110	(CF OR ZF) = 1
NBE A	Neither below nor equal Above	0111	(CF OR ZF) = 0
S	Sign	1000	SF = 1
NS	No sign	1001	SF = 0
P PE	Parity Parity even	1010	PF = 1
NP PO	No parity Parity odd	1011	PF = 0
L NGE	Less Neither greater nor equal	1100	(SF XOR OF) = 1
NL GE	Not less Greater or equal	1101	(SF XOR OF) = 0
LE NG	Less or equal Not greater	1110	((SF XOR OF) OR ZF) = 1

表 B-1. EFLAGS 条件コード（続き）

ニーモニック（*cc*）	テストされる条件	命令サブコード	ステータス・フラグの設定
NLE G	Neither less nor equal Greater	1111	((SF XOR OF) OR ZF) = 0

テスト条件の多くには 2 種類の表現がある。例えば、LE（less or equal: より小または等しい）と NG（not greater: より大ではない）は同じテスト条件を表す。コードが理解しやすいように、このような代替ニーモニックが用意されている。

「より上（above）」と「より下（below）」という用語は CF フラグに関連しており、2 つの符号なし整数値の関係を表す。「より大（greater）」と「より小（less）」という用語は SF フラグと OF フラグに関連しており、2 つの符号付き整数値の関係を表す。

C

浮動小数点例外の要約

付録 C
浮動小数点例外の要約

C

本章では、次の命令で生成される、浮動小数点例外の一覧を示す。

- x87 FPU 命令 － 表 C-2. を参照。
- SSE － 表 C-3. を参照。
- SSE2 命令 － 表 C-4. を参照。
- SSE3 命令 － 表 C-5. を参照。

表 C-1. は、x87 FPU 命令、SSE 命令、SSE2 命令、SSE3 命令で生成される可能性のある浮動小数点例外を示している。

表 C-1. x87 FPU 浮動小数点例外と SIMD 浮動小数点例外

浮動小数点例外	説明
#IS	スタック・アンダーフローまたはスタック・オーバーフローによる無効操作例外（x87 FPU 命令でのみ生成される）*
#IA または #I	無効算術オペランドとサポートされていないフォーマットによる無効操作例外 *
#D	デノーマル・オペランド例外
#Z	ゼロ除算例外
#O	数値オーバーフロー例外
#U	数値アンダーフロー例外
#P	不正確結果（精度）例外

* x87 FPU 命令セットは、#IS（スタック・アンダーフローまたはスタック・オーバーフロー）および #IA（無効算術オペランドまたはサポートされていないフォーマットによる無効算術操作）の 2 種類の無効操作例外を生成する。SSE、SSE2、SSE3 は、#I（無効算術オペランドまたはサポートされていないフォーマットによる無効操作例外）を生成する可能性がある。

表 C-1. に示した浮動小数点例外は、#D と #IS を除いて、2 進浮動小数点算術演算に関する IEEE 規格 754-1985 に定義されている。浮動小数点例外についての詳細は、4.9.1. 項「浮動小数点例外条件」を参照のこと。

C.1. x87 FPU 命令

表 C-2. は、x87 FPU 命令をアルファベット順に示し、各命令について、その命令で生成される浮動小数点例外をまとめたものである。

表 C-2. x87 FPU 浮動小数点命令で生成される例外

ニーモニック	命令	#IS	#IA	#D	#Z	#O	#U	#P
F2XM1	指数	○	○	○			○	○
FABS	絶対値	○						
FADD(P)	浮動小数点の加算	○	○	○		○	○	○
FBLD	BCD のロード	○						
FBSTP	BCD のストアとポップ	○	○					○
FCHS	符号の変更	○						
FCLEX	例外のクリア							
FCMOV*cc*	浮動小数点の条件付き移動	○						
FCOM, FCOMP, FCOMPP	浮動小数点の比較	○	○	○				
FCOMI, FCOMIP, FUCOMI, FUCOMIP	浮動小数点の比較と EFLAGS の設定	○	○	○				
FCOS	余弦	○	○	○				○
FDECSTP	スタックポインタのデクリメント							
FDIV(R)(P)	浮動小数点の除算	○	○	○	○	○	○	○
FFREE	レジスタの解放							
FIADD	整数の加算	○	○	○		○	○	○
FICOM(P)	整数の比較	○	○	○				
FIDIV	整数の除算	○	○	○	○		○	○
FIDIVR	項を逆転した整数の除算	○	○	○	○	○	○	○
FILD	整数のロード	○						
FIMUL	整数の乗算	○	○	○		○	○	○
FINCSTP	スタックポインタのインクリメント							
FINIT	プロセッサの初期化							
FIST(P)	整数のストア	○	○					○
FISTTP	整数への切り捨て（SSE3 命令）	○	○					○
FISUB(R)	整数の減算	○	○	○		○	○	○
FLD extended or stack	浮動小数点のロード	○						
FLD single or double	浮動小数点のロード	○	○	○				
FLD1	+1.0 のロード	○						
FLDCW	制御ワードのロード	○	○	○	○	○	○	○

表 C-2. x87 FPU 浮動小数点命令で生成される例外（続き）

ニーモニック	命令	#IS	#IA	#D	#Z	#O	#U	#P
FLDENV	環境のロード	○	○	○	○	○	○	○
FLDL2E	$\log_2 e$ のロード	○						
FLDL2T	$\log_2 10$ のロード	○						
FLDLG2	$\log_{10} 2$ のロード	○						
FLDLN2	$\log_e 2$ のロード	○						
FLDPI	π のロード	○						
FLDZ	+0.0 のロード	○						
FMUL(P)	浮動小数点の乗算	○	○	○		○	○	○
FNOP	ノー・オペレーション							
FPATAN	部分逆正接	○	○	○			○	○
FPREM	部分剰余	○	○	○			○	
FPREM1	IEEE 部分剰余	○	○	○			○	
FPTAN	部分正接	○	○	○			○	○
FRNDINT	整数への丸め	○	○	○				○
FRSTOR	ステートのリストア	○	○	○	○	○	○	○
FSAVE	ステートのセーブ							
FSCALE	スケール	○	○	○		○	○	○
FSIN	正弦	○	○	○			○	○
FSINCOS	正弦と余弦	○	○	○			○	○
FSQRT	平方根	○	○	○				○
FST(P) stack or extended	浮動小数点のストア	○						
FST(P) single or double	浮動小数点のストア	○	○			○	○	○
FSTCW	制御ワードのストア							
FSTENV	環境のストア							
FSTSW (AX)	ステータス・ワードのストア							
FSUB(R)(P)	浮動小数点の減算	○	○	○		○	○	○
FTST	テスト	○	○	○				
FUCOM(P)(P)	浮動小数点の順序付けなしの比較	○	○	○				
FWAIT	CPU ウェイト							
FXAM	検査							
FXCH	レジスタの交換	○						
FXTRACT	抽出	○	○	○	○			
FYL2X	対数	○	○	○	○	○	○	○
FYL2XP1	対数極小値（X + 1）	○	○	○		○	○	○

C.2. SSE

表 C-3. は、以下の特性のいずれか 1 つ以上に当てはまる SSE を示している。

- 浮動小数点オペランドを持つ。
- 浮動小数点形式の結果を生成する。
- 浮動小数点値のステータス情報と制御情報を読み取りまたは書き込みする。

表 C-3. は、各命令について、その命令で生成される浮動小数点例外をまとめたものである。

表 C-3. SSE で生成される例外

ニーモニック	命令	#I	#D	#Z	#O	#U	#P
ADDPS	パックド値の加算	○	○		○	○	○
ADDSS	スカラ値の加算	○	○		○	○	○
ANDNPS	パックド値の INVERT（反転）および AND（論理積）演算						
ANDPS	パックド値の AND（論理積）演算						
CMPPS	パックド値の比較	○	○				
CMPSS	スカラ値の比較	○	○				
COMISS	最下位の単精度浮動小数点値を順序付きで比較し、ステータス・フラグを設定する。	○	○				
CVTPI2PS	MM2/Mem からの 2 つの符号付き 32 ビット整数を 2 つの単精度浮動小数点値に変換する。						○
CVTPS2PI	XMM/Mem からの下位の 2 つの単精度浮動小数点値を MM 内の 2 つの符号付き 32 ビット整数に変換し、MXCSR で指定された方法で丸める。	○					○
CVTSI2SS	整数レジスタ / メモリからの 1 つの符号付き 32 ビット整数を 1 つの単精度浮動小数点値に変換する。						○
CVTSS2SI	XMM/Mem からの 1 つの単精度浮動小数点値を 1 つの符号付き 32 ビット整数に変換し、MXCSR で指定された方法で丸めて、結果を整数レジスタに転送する。	○					○
CVTTPS2PI	切り捨てを使用して、XMM2/Mem からの 2 つの単精度浮動小数点値を MM1 内の 2 つの符号付き 32 ビット整数に変換する。	○					○
CVTTSS2SI	切り捨てを使用して、XMM/Mem の最下位の単精度浮動小数点値を 1 つの符号付き 32 ビット整数に変換し、結果を整数レジスタに転送する。	○					○
DIVPS	パックド値の除算	○	○	○	○	○	○
DIVSS	スカラ値の除算	○	○	○	○	○	○

表 C-3. SSE で生成される例外（続き）

ニーモニック	命令	#I	#D	#Z	#O	#U	#P
LDMXCSR	制御 / ステータス・ワードをロードする。						
MAXPS	パックド値の最大値	○	○				
MAXSS	スカラ値の最大値	○	○				
MINPS	パックド値の最小値	○	○				
MINSS	スカラ値の最小値	○	○				
MOVAPS	4 つのパックド単精度値を転送する。						
MOVHLPS	パックド単精度値を上位から下位に転送する。						
MOVHPS	2 つのパックド単精度値をメモリと XMM レジスタの上位半分との間で転送する。						
MOVLHPS	パックド単精度値を下位から上位に転送する。						
MOVLPS	2 つのパックド単精度値をメモリと XMM レジスタの下位半分との間で転送する。						
MOVMSKPS	符号マスクを r32 に移動						
MOVSS	スカラ単精度値を XMM レジスタとメモリまたは第 2 の XMM レジスタとの間で転送する。						
MOVUPS	アライメントの合っていないパックデータの移動						
MULPS	パックド値の乗算	○	○		○	○	○
MULSS	スカラ値の乗算	○	○		○	○	○
ORPS	パックド値の OR（論理和）演算						
RCPPS	パックド値の平方根の逆数						
RCPSS	スカラ値の逆数						
RSQRTPS	パックド値の平方根の逆数						
RSQRTSS	スカラ値の平方根の逆数						
SHUFPS	シャッフル						
SQRTPS	パックド単精度浮動小数点値の平方根	○	○				○
SQRTSS	スカラ値の平方根	○	○				○
STMXCSR	制御 / ステータス・ワードのストア						
SUBPS	パックド値の減算	○	○		○	○	○
SUBSS	スカラ値の減算	○	○		○	○	○
UCOMISS	最下位の単精度浮動小数点値を順序付けなしで比較し、ステータス・フラグを設定する。	○	○				
UNPCKHPS	単精度浮動小数点値のインタリーブ						
UNPCKLPS	単精度浮動小数点値のインタリーブ						
XORPS	パックド値の XOR（排他的論理和）演算						

C.3. SSE2

表 C-4. は、以下の特性のいずれか 1 つ以上に当てはまる SSE2 を示している。

- 浮動小数点オペランド
- 浮動小数点形式の結果

表 C-4. は、各命令について、その命令で生成される浮動小数点例外をまとめたものである。

表 C-4. SSE2 で生成される例外

命令	説明	#I	#D	#Z	#O	#U	#P
ADDPD	XMM2/Mem と XMM1 のパック ド倍精度浮動小数点値を加算して、結果を XMM1 に格納する。	○	○		○	○	○
ADDSD	XMM2/Mem と XMM1 の下位の倍精度浮動小数点値を加算して、結果を XMM1 に格納する。	○	○		○	○	○
ANDNPD	XMM1 の 128 ビットを反転して、その結果と XMM2/Mem の 128 ビットの間で AND（論理積）演算を実行する。						
ANDPD	XMM2/Mem の 128 ビットと XMM1 の 128 ビットの AND（論理積）演算を実行して、結果を XMM1 レジスタに格納する。						
CMPPD	imm8 をプレディケートとして使用して、XMM2/Mem のパック ド倍精度浮動小数点値と、XMM1 レジスタのパック ド倍精度浮動小数点値を比較する。	○	○				
CMPSD	imm8 をプレディケートとして使用して、XMM2/Mem の最下位の倍精度浮動小数点値と、XMM1 レジスタの最下位の倍精度浮動小数点値を比較する。	○	○				
COMISD	XMM1 レジスタの下位の倍精度浮動小数点値と XMM2/Mem の下位の倍精度浮動小数点値を比較し、その結果にしたがってステータス・フラグをセットする。	○	○				
CVTDQ2PS	XMM/Mem の 4 つの符号付き 32 ビット整数を、4 つの単精度浮動小数点値に変換する。						○
CVTPS2DQ	MXCSR で指定された丸めを使用して、XMM/Mem の 4 つの単精度浮動小数点値を、XMM の 4 つの符号付き 32 ビット整数に変換する。	○					○
CVTTPS2DQ	切り捨てを使用して、XMM/Mem の 4 つの単精度浮動小数点値を、XMM の 4 つの符号付き 32 ビット整数に変換する。	○					○
CVTDQ2PD	MXCSR で指定された丸めを使用して、XMM2/Mem の 2 つの符号付き 32 ビット整数を、xmm1 の 2 つの倍精度浮動小数点値に変換する。						

表 C-4. SSE2 で生成される例外（続き）

命令	説明	#I	#D	#Z	#O	#U	#P
CVTPD2DQ	MXCSR で指定された丸めを使用して、XMM2/Mem の 2 つの倍精度浮動小数点値を、xmm1 の 2 つの符号付き 32 ビット整数に変換する。	○					○
CVTPD2PI	MXCSR で指定された丸めを使用して、XMM/Mem の下位 2 つの倍精度浮動小数点値を、MM の 2 つの符号付き 32 ビット整数に変換する。	○					○
CVTPD2PS	2 つの倍精度浮動小数点値を、2 つの単精度浮動小数点値に変換する。	○	○		○	○	○
CVTPI2PD	MM2/Mem の 2 つの符号付き 32 ビット整数を、2 つの倍精度浮動小数点値に変換する。						
CVTPS2PD	2 つの単精度浮動小数点値を、2 つの倍精度浮動小数点値に変換する。	○	○				
CVTSD2SI	MXCSR で指定された丸めモードを使用して、XMM/Mem の 1 つの倍精度浮動小数点値を 1 つの符号付き 32 ビット整数に変換し、結果を整数レジスタに転送する。	○					○
CVTSD2SS	スカラ倍精度浮動小数点値を、スカラ単精度浮動小数点値に変換する。	○	○		○	○	○
CVTSI2SD	整数レジスタ / メモリの 1 つの符号付き 32 ビット整数を、1 つの倍精度浮動小数点値に変換する。						
CVTSS2SD	スカラ単精度浮動小数点値を、スカラ倍精度浮動小数点値に変換する。	○	○				
CVTTPD2DQ	切り捨てを使用して、XMM2/Mem の 2 つの倍精度浮動小数点値を、XMM1 の 2 つの符号付き 32 ビット整数に変換する。	○					○
CVTTPD2PI	切り捨てを使用して、XMM2/Mem の 2 つの倍精度浮動小数点値を、MM1 の 2 つの符号付き 32 ビット整数に変換する。	○					○
CVTTSD2SI	切り捨てを使用して、XMM/Mem の最下位の倍精度浮動小数点値を 1 つの符号付き 32 ビット整数に変換し、結果を整数レジスタに転送する。	○					○
DIVPD	XMM1 のパック ド倍精度浮動小数点値を、XMM2/Mem で割る。	○	○	○	○	○	○
DIVSD	XMM1 の下位の倍精度浮動小数点値を、XMM2/Mem で割る。	○	○	○	○	○	○
MAXPD	XMM2/Mem と XMM1 の倍精度浮動小数点値の間で最大の値を返す。	○	○				
MAXSD	XMM2/Mem と XMM1 の最下位の倍精度浮動小数点値の間で最大の値を返す。	○	○				
MINPD	XMM2/Mem と XMM1 の倍精度浮動小数点値の間で最小の値を返す。	○	○				

表 C-4. SSE2 で生成される例外（続き）

命令	説明	#I	#D	#Z	#O	#U	#P
MINSD	XMM2/Mem と XMM1 の最下位の倍精度浮動小数点値の間で最小の値を返す。	○	○				
MOVAPD	2 つのパック ド倍精度浮動小数点データを表す 128 ビットを、XMM2/Mem から XMM1 レジスタに転送する。 または、2 つのパック ド倍精度浮動小数点データを表す 128 ビットを、XMM1 レジスタから XMM2/Mem に転送する。						
MOVHPD	1 つの倍精度浮動小数点オペランドを表す 64 ビットを、メモリから XMM レジスタの上位フィールドに転送する。 または、1 つの倍精度浮動小数点オペランドを表す 64 ビットを、XMM レジスタの上位フィールドからメモリに転送する。						
MOVLPD	1 つの倍精度浮動小数点オペランドを表す 64 ビットを、メモリから XMM レジスタの下位フィールドに転送する。 または、1 つの倍精度浮動小数点オペランドを表す 64 ビットを、XMM レジスタの下位フィールドからメモリに転送する。						
MOVMSKPD	1 つのマスクを r32 に転送する。						
MOVSD	1 つのスカラ倍精度浮動小数点オペランドを表す 64 ビットを、XMM2/Mem から XMM1 レジスタに転送する。 または、1 つのスカラ倍精度浮動小数点オペランドを表す 64 ビットを、XMM1 レジスタから XMM2/Mem に転送する。						
MOVUPD	2 つの倍精度浮動小数点データを表す 128 ビットを、XMM2/Mem から XMM1 レジスタに転送する。 または、2 つの倍精度浮動小数点データを表す 128 ビットを、XMM1 レジスタから XMM2/Mem に転送する。						
MULPD	XMM2/Mem と XMM1 のパック ド倍精度浮動小数点値を乗算して、結果を XMM1 に格納する。	○	○		○	○	○
MULSD	XMM2/Mem と XMM1 の最下位の倍精度浮動小数点値を乗算して、結果を XMM1 に格納する。	○	○		○	○	○
ORPD	XMM2/Mem と XMM1 の 128 ビットの OR（論理和）演算を実行して、結果を XMM1 レジスタに格納する。						
SHUFPD	倍精度浮動小数点値のシャッフル						
SQRTPD	パック ド倍精度浮動小数点値の平方根	○	○				○
SQRTSD	スカラ倍精度浮動小数点値の平方根	○	○				○
SUBPD	パック ド倍精度浮動小数点値の減算	○	○		○	○	○
SUBSD	スカラ倍精度浮動小数点値の減算	○	○		○	○	○

表 C-4. SSE2 で生成される例外（続き）

命令	説明	#I	#D	#Z	#O	#U	#P
UCOMISD	XMM1 レジスタの下位の倍精度浮動小数点値と XMM2/Mem の下位の倍精度浮動小数点値を比較し、その結果にしたがってステータス・フラグをセットする。	○	○				
UNPCKHPD	XMM1 の上位半分と XMM2/Mem の上位半分の倍精度浮動小数点値をインターリーブして、XMM1 レジスタに格納する。						
UNPCKLPD	XMM1 の下位半分と XMM2/Mem の下位半分の倍精度浮動小数点値をインターリーブして、XMM1 レジスタに格納する。						
XORPD	XMM2/Mem と XMM1 の 128 ビットの XOR（排他的論理和）演算を実行して、結果を XMM1 レジスタに格納する。						

C.4. SSE3

表 C-5. は、以下の特性のいずれか 1 つ以上に当てはまる SSE3 を示している。

- 浮動小数点オペランドを持つ。
- 浮動小数点形式の結果を生成する。

表 C-5. は、各命令について、その命令で生成される浮動小数点例外をまとめたものである。

表 C-5. SSE2 で生成される例外

命令	説明	#I	#D	#Z	#O	#U	#P
ADDSUBPD	XMM2/Mem と XMM1 のパックド倍精度浮動小数点値を加算および減算して、結果を XMM1 に格納する。	○	○		○	○	○
ADDSUBPS	XMM2/Mem と XMM1 のパックド単精度浮動小数点値を加算および減算して、結果を XMM1 に格納する。	○	○		○	○	○
FISTTP	表 C-2. を参照のこと。	○					○
HADDPD	XMM2/Mem と XMM1 のパックド倍精度浮動小数点値を水平に加算して、結果を XMM1 に格納する。	○	○		○	○	○
HADDPS	XMM2/Mem と XMM1 のパックド単精度浮動小数点値を水平に加算して、結果を XMM1 に格納する。	○	○		○	○	○
HSUBPD	XMM2/Mem と XMM1 のパックド倍精度浮動小数点値を水平に減算して、結果を XMM1 に格納する。	○	○		○	○	○
HSUBPS	XMM2/Mem と XMM1 のパックド単精度浮動小数点値を水平に減算して、結果を XMM1 に格納する。	○	○		○	○	○
LDDQU	アライメントの合っていない 128 ビット整数をロードする。						
MOVDDUP	1 つの倍精度データに相当する 64 ビットを XMM2/Mem から XMM1 に転送して複製する。						
MOVSHDUP	4 つの単精度データに相当する 128 ビットを XMM2/Mem から XMM1 に転送して上位を複製する。						
MOVSLDUP	4 つの単精度データに相当する 128 ビットを XMM2/Mem から XMM1 に転送して下位を複製する。						

D

x87 FPU 例外ハンドラを作成する際のガイドライン

付録 D
x87 FPU 例外ハンドラを作成する際のガイドライン

D

第 8 章「x87 FPU による プログラミング」で説明したように、IA-32 アーキテクチャでは、マスクされていない x87 FPU 例外を処理する例外ハンドラにアクセスするモードとして、ネイティブ・モードと MS-DOS* 互換モードの 2 種類をサポートしている。この付録の主な目的は、ソフトウェア・エンジニアが PC システム上で MS-DOS 互換モード[1] で動作する x87 FPU 例外ハンドラを設計し作成する方法を詳しく説明することであるが、ネイティブ・モードで動作する x87 FPU 例外ハンドラを作成するエンジニアに有益な情報も含まれている。付録 D で説明する項目を次に示す。

- MS-DOS 互換モードの x87 FPU 例外処理メカニズムの由来と、ネイティブ・モードの FPU 例外処理メカニズムとの関係
- MS-DOS 互換モードの x87 FPU 例外処理メカニズムを制御するための IA-32 におけるフラグとプロセッサ・ピン
- MS-DOS 例外処理メカニズムをサポートするうえで必要な外部ハードウェア
- x87 FPU 例外処理メカニズムと、x87 FPU 例外ハンドラのための一般的なプロトコル
- さまざまなレベルの x87 FPU 例外ハンドラを説明するためのコード例
- マルチタスク環境の x87 FPU に関する注意事項
- ネイティブ・モードの x87 FPU 例外処理

ここで説明する内容は、最新の IA-32 プロセッサ（Intel486™ 以降）に関するもので、第 8 章「x87 FPU による プログラミング」の追加情報となるものである。

さらに詳しい情報については、インテルから入手可能なアプリケーション・ノート AP-578『Software and Hardware Considerations for x87 FPU Exception Handlers for Intel Architecture Processors』（資料番号 243291）を参照のこと。

1. Microsoft* Windows* 95 および Windows 3.1（およびそれ以前のバージョン）のオペレーティング・システムでも、MS-DOS* オペレーティング・システムとほぼ同じ x87 FPU 例外処理インターフェイスを使用している。付録 D に示すガイドラインは、この 3 種類のどのオペレーティング・システムにも適用される。

D.1. MS-DOS* 互換モードの x87 FPU 例外処理メカニズムの由来

第一世代の IA-32 プロセッサ（インテル® 8086 と 8088 プロセッサから、インテル® 286 プロセッサ、Intel386™ プロセッサまで）には、オンチップの浮動小数点ユニットがなく、別の数値演算コプロセッサ・チップで浮動小数点機能を提供していた。最初の数値演算コプロセッサはインテル® 8087 で、その後インテル® 287 プロセッサ、インテル® 387 プロセッサの順に発表された。

8087 には、8086 または 8088 プロセッサに浮動小数点例外を通知するための INT 出力ピンがあり、マスクされていない浮動小数点例外が発生すると、この INT ピンがアサートする。8087 の設計者は、INT ピンからの出力を、インテル® 8259A のようなプログラマブル割り込みコントローラ（PIC）を通して 8086 または 8088 プロセッサの INTR ピンに接続するよう推奨している。提供されている割り込みベクタ番号を使用して、浮動小数点例外ハンドラにアクセスできるからである。

ところが、最初の IBM* PC の設計と MS-DOS* オペレーティング・システムにおいては、8087 からの INT 出力を別の方法によって処理していた。すなわち、INT ピンからの出力が 8086 または 8088 の NMI 入力ピンに直接接続されていたため、NMI 割り込みハンドラは、割り込みが浮動小数点例外で発生したのか他の NMI イベントで発生したのかを判定する必要があった。これが現在「MS-DOS 互換モード」と呼ばれるメカニズムの由縁である。このような浮動小数点例外処理メカニズムが使用されたのは、IBM PC が最初に設計された時点では 8087 がまだ発表されていなかったためである。そして、8087 が使用できるようになったときには、PIC の 8 つの入力がすでに他の機能に割り当てられてしまっていた。その機能の 1 つに BIOS ビデオ割り込みがあり、これに 8086 と 8088 の割り込み番号 16 が割り当てられた。

インテル 286 プロセッサでは、浮動小数点例外信号を受信する専用の入力ピン（ERROR#）と専用の割り込み番号（16）を使用して浮動小数点例外を処理する「ネイティブ・モード」が導入された。割り込み番号 16 は、浮動小数点エラー（数値フォルトとも呼ぶ）を通知するのに使用される。これは、Intel 286 の ERROR# ピンが、インテル 287 数値演算コプロセッサの ERROR# ピンと接続されることを前提に設計されたものである。この場合、インテル 278 が浮動小数点例外を通知すると、インテル 286 で割り込み 16 が発生して浮動小数点例外ハンドラが呼び出される。

ところが、IBM PC AT システムの設計では、従来の PC ソフトウェアとの互換性を保つために、インテル 286 と 287 のネイティブ・モードによる浮動小数点例外ハンドラは採用されなかった。そのかわりに、インテル 286 の ERROR# ピンは常時ハイになるように結線され、インテル 287 の ERROR# ピンは第二（カスケード）PIC に接続された。このカスケード PIC からの出力は、例外ハンドラにより最終的に割り込み 2（NMI 割り込み）として処理される。ここで、NMI 割り込みは IBM PC AT の新たなパリティ・チェック機能と共有される。割り込み 16 は、従来通り BIOS ビデオ割り込みハンドラ

に割り当てられた。MS-DOS 互換モードを使用するためには、外部ハードウェアにより、マスクされていない割り込みが発生したときにインテル 286 プロセッサの実行が次の x87 FPU 命令を超えないようにしなければならない。そのためには、インテル 287 から ERROR# 信号がアサートされたら、インテル 286 プロセッサへの BUSY# 信号をアサートするように設計する。

Intel386 プロセッサと Intel 387 数値演算コプロセッサの場合も、浮動小数点例外の信号と処理に関するハードウェア・メカニズムは、インテル 286 プロセッサおよび 287 プロセッサと同様である。また、Intel386 プロセッサを使用した PC においても、従来の MS-DOS ソフトウェアとの互換性を維持するために、IBM PC AT と基本的には同様の MS-DOS 互換モードの浮動小数点例外処理メカニズムを採用している。

D.2. Intel486™ プロセッサ、インテル® Pentium® プロセッサ、P6 プロセッサ・ファミリおよびインテル® Pentium® 4 プロセッサにおける MS-DOS* 互換モード

Intel486™ プロセッサ以降の IA-32 では、MS-DOS* 互換モードによる x87 FPU 例外信号と外部 x87 FPU 例外信号のための専用メカニズムが提供されている。以降の各項では、Intel486 プロセッサ、インテル® Pentium® プロセッサ、P6 ファミリおよびインテル® Pentium® 4 プロセッサにおける MS-DOS 互換モードのインプリメンテーションを説明する。また、MS-DOS 互換モードでの動作をサポートするための推奨される外部ハードウェアについても説明する。

D.2.1. Intel486™ プロセッサとインテル® Pentium® プロセッサにおける MS-DOS* 互換モード

Intel486™ プロセッサでは、数値演算コプロセッサ（現在は浮動小数点ユニット（x87 FPU）と呼ばれる）の機能拡張と高速化のためにさまざまな変更が行われた。そのうち最も重要なのは、x87 FPU の演算速度を高め x87 FPU 例外処理のためのレイテンシを短縮するために、x87 FPU がプロセッサと同一チップに組み込まれたことである。さらに、制御レジスタ CR0 への NE ビットの追加、および FERR#（Floating point ERRor）ピンと IGNNE#（IGNore Numeric Error）ピンの追加により、チップ設計において初めて MS-DOS* 互換モードが組み込まれた。

NE ビットにより、x87 FPU 例外処理をネイティブ・モードで行うか（NE=1）、MS-DOS 互換モードで行うか（NE=0）を選択する。ネイティブ・モードが選択されると、浮動小数点例外信号はすべて Intel486 チップ内部で処理され、その結果、例外 16 が発生する。

MS-DOS 互換モードが選択されると、FERR# ピンと IGNNE# ピンを使用して浮動小数点例外を通知する。FERR# 出力ピンは従来の IA-32 数値演算コプロセッサの ERROR#

ピンに対応するもので、PIC に接続する。新たに追加された IGNNE# 入力ピンは、x87 FPU例外ハンドラが必要に応じてエラー条件をクリアして再び割り込みをトリガしなくても x87 FPU 命令を実行できるようにするためのものである。IGNNE# の機能は、x87 FPU 例外ハンドラの中でエラー条件をクリアする前に、インテル 286 プロセッサとインテル 287 プロセッサのシステムおよび Intel386 プロセッサとインテル 387 プロセッサのシステムにおいて MS-DOS 互換モードで BUSY# 信号をオフにしたときと同じ機能を持たせるためのものである。

Intel486 プロセッサの中には、x87 FPU が不要なマーケット向けに発表された SX バージョンがあることに注意しなければならない。この Intel486 SX プロセッサには浮動小数点ユニットがない。また、x87 FPU を追加してシステムをアップグレードしたいエンド・ユーザ向けに、インテル 487 SX プロセッサも発表されている。インテル 487 SX プロセッサは、オンボードの x87 FPU を持つ標準の Intel486 プロセッサに相当する。

したがって、インテル 487 SX プロセッサで MS-DOS 互換モードをサポートするのに必要な外部回路は、標準の Intel486 DX プロセッサの場合と同じである。

インテル Pentium プロセッサ、P6 ファミリ、インテル Pentium 4 プロセッサでも Intel486 プロセッサと同じメカニズム（NE ビット、および FERR# ピンと IGNNE# ピン）を使用して、MS-DOS 互換モードでの x87 FPU 例外処理を行っている。ただし、P6 ファミリおよびインテル Pentium 4 プロセッサの場合は、D.2.2. 項「P6 ファミリおよびインテル® Pentium® 4 プロセッサにおける MS-DOS* 互換モード」で説明するように、処理動作が少し異なっていて、より単純になっている。

インテル® Pentium® プロセッサ、P6 ファミリ、インテル® Pentium® 4 プロセッサ・ファミリの場合、インテル Pentium プロセッサでの特殊な DP（デュアル・プロセッシング）モードや複数のインテル Pentium プロセッサ、P6 ファミリ、またはインテル Pentium 4 プロセッサが組み込まれているシステム向けの汎用インテル・マルチプロセッサ仕様においては、x87 FPU 例外処理はネイティブ・モードでのみサポートしていることに注意しなければならない。複数のプロセッサが組み込まれているシステムで MS-DOS 互換の x87 FPU モードを使用することはお勧めできない。

D.2.1.1. FERR# 信号発生時の基本規則

Intel486™ プロセッサまたはインテル® Pentium® プロセッサで MS-DOS* 互換モードが選択されていて（NE ビットが 0）、IGNNE# 入力ピンがディアサートされていれば、FERR# 信号は次のように発生する。

1. x87 FPU 命令によりマスクされていない x87 FPU 例外が発生すると、多くの場合、プロセッサは「据え置き」方式でエラーの発生を通知する。すなわち、プロセッサは即座にエラー発生に応答せずに、次の WAIT 命令または x87 FPU 命令の前で初めてフリーズする（ただし、「非同期型」命令の場合は、エラー条件に関係なく x87 FPU の実行が続けられる）。

2. プロセッサはフリーズすると同時に FERR# 出力をアサートする。
3. フリーズ状態になったプロセッサは外部割り込みを待機する。この外部割り込みは、FERR# アサートに応答して外部ハードウェアによって生じる。
4. MS-DOS 互換システムでは、FERR# がカスケード PIC の IRQ13 入力に接続されている。PIC は割り込み 75H を発生させ、これが割り込み 2 に分岐する（インテル 286 プロセッサとインテル 287 プロセッサのシステム、または Intel386 プロセッサとインテル 387 プロセッサのシステムに関する前述の説明を参照のこと）。

据え置き方式でエラー通知が行われるのは、基本算術命令（FADD、FSUB、FMUL、FDIV、FSQRT、FCOM、FUCOM など）によって生じた例外、あらゆるタイプの x87 FPU 命令による精度例外、メモリへの格納を除くあらゆるタイプの x87 FPU 命令による数値アンダーフロー例外とオーバーフロー例外の場合である。

x87 FPU 命令での x87 FPU 例外のなかには、「即時」方式でエラーを通知するものもある。この場合は、例外が発生すると同時に FERR# がアサートされる。即時方式のエラー通知が行われるのは、すべての超越命令、FSCALE、FXTRACT、FPREM などによって生じた x87 FPU スタック・フォルト例外、無効操作例外、デノーマル例外、x87 FPU 格納命令で生じたすべての例外（ただし精度例外は除く）の場合である。据え置き方式によるエラー通知と同様に、即時方式の場合も、次の WAIT 命令または x87 FPU 命令を実行しようとしたときにエラー条件がクリアされていなければ、その直前でプロセッサがフリーズする。

一般に、x87 FPU 例外の発生時に据え置き方式と即時方式のどちらでエラー通知が行われるかは、例外の種類、およびその例外を発生させた命令の種類によって決まる。インテル Pentium プロセッサと Intel486 プロセッサにおけるエラー通知タイプの詳細については、『Pentium® Processor Family Developer's Manual, Volume 1』の 5.1.2.1 項を参照のこと。

NE=0 でも IGNNE# がアクティブならば、マスクされていない x87 FPU 例外が発生しても、プロセッサはその例外を無視し、FERR# をアサートせずに処理を続行する。その後 IGNNE# がディアサート状態になったときに x87 FPU 例外がクリアされていなければ、プロセッサは上記のように応答する（すなわち、即時方式の例外ならば即座に FERR# をアサートし、据え置き方式の例外ならば次の WAIT 命令または x87 FPU 命令の直前でフリーズして FERR# をアサートする）。IGNNE# をアサートするのは、x87 FPU 例外ハンドラの中で例外条件をクリアする前に診断のための非制御 x87 FPU 命令を実行したい場合のみを想定している。例外ハンドラの中で IGNNE# をアサートすると、その前の x87 FPU 例外によって FERR# がすでにアサートされており、さらに外部割り込みハードウェアがすでに応答してはいるが、IGNNE# のアサートによって x87 FPU 命令のところでフリーズすることはない。x87 FPU 例外ハンドラの外部で IGNNE# がアクティブになっていると、x87 FPU 例外を発生した命令の後に別の x87 FPU 命令

が実行されてしまうことがある。この場合、x87 FPU 例外ハンドラが呼び出されても、どちらの命令で発生した例外なのか判定できなくなる。

プロセッサの FERR# 出力と IGNNE# 入力、および PIC の IRQ13 入力の間のインターフェイスを適切に管理するためには、外部ハードウェアが必要である。推奨するハードウェア構成を次の項で説明する。

D.2.1.2. MS-DOS* 互換モードをサポートするための推奨外部ハードウェア

x87 FPU 例外が発生した場合に FERR# と IGNNE# を適切に処理できる外部回路を図 D-1. に示す。この回路では特に、x87 FPU 例外ハンドラの処理順序に関係なく、IGNNE# が x87 FPU 例外ハンドラの内部でしかアクティブにならないことが保証されている。ハードウェアのインプリメントによっては、例外ハンドラが x87 FPU そのものから例外をクリアして FERR# をディアサート状態にする前に、例外ハンドラが自分で PIC への x87 FPU 例外割り込み要求（FP_IRQ 信号）をクリアしなければならない点が問題になる場合がある。図 D-2. には、図 D-1. の回路における IGNNE# 信号の状態を詳しく示す。x87 FPU 例外ハンドラ内部の動作を次に順を追って説明する。

1. x87 FPU 例外によってアクティブになった FERR# 信号により、割り込み要求が PIC を通ってプロセッサの INTR ピンに送信される。
2. x87 FPU 割り込み処理ルーチン（例外ハンドラ）の実行中に、プロセッサが割り込み要求ラッチ（フリップフロップ #1）をクリアする必要が起こったり、x87 FPU から例外をクリアする前に非制御 x87 FPU 命令を実行したい場合がある。そのためには、IGNNE# 信号をローにする必要がある。一般の PC 環境では、ポート 0F0H に I/O アクセスすることにより、外部 x87 FPU 例外割り込み要求（FP_IRQ）がクリアされる。図 D-1 の推奨回路の場合は、この I/O アクセスにより、IGNNE# がアクティブになる。IGNNE# がアクティブになっていれば、有効な x87 FPU 例外によってブロックされることなく、x87 FPU 例外ハンドラは x87 FPU 命令を実行できる。
3. x87 FPU から例外をクリアすると、FERR# 信号が非アクティブになる。その後も IGNNE# をアクティブにしておく必要はない。図 D-1 の推奨回路では、FERR# を非アクティブにすることで IGNNE# も非アクティブになる。図 D-1 と異なる回路を使用する場合は、x87 FPU 例外ハンドラを終了する前に必ず IGNNE# が非アクティブになるようにソフトウェアおよび回路上で保証しなければならない。

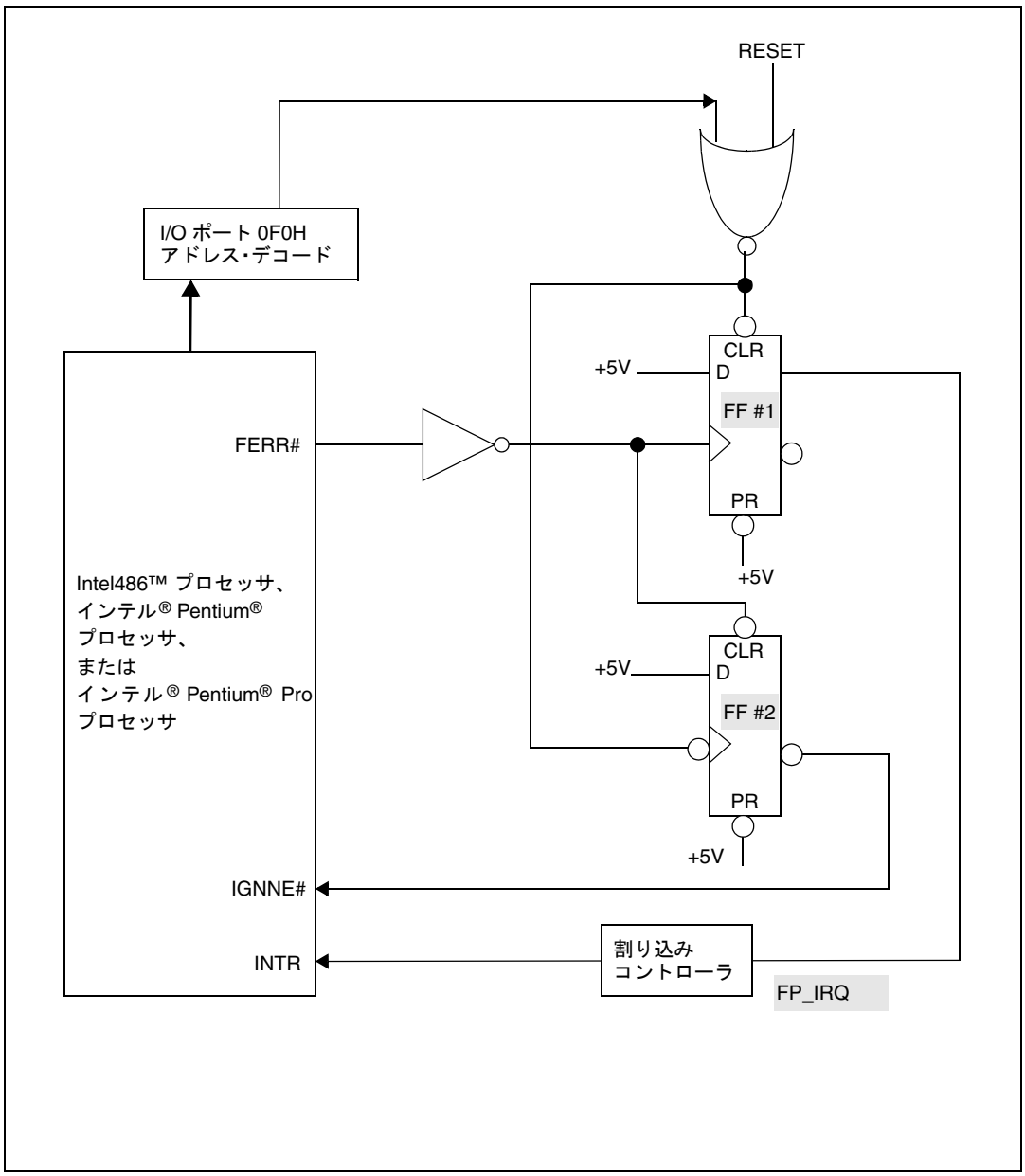


図 D-1. MS-DOS* 互換モードで x87 FPU 例外処理を行う場合の推奨回路

図 D-1. の回路では、x87 FPU 例外ハンドラで I/O ポート 0F0H にアクセスすると、フリップフロップ #1 から出力される IRQ13 割り込み要求がクリアされ、フリップフロップ #2 からの IGNNE# 信号がクロック・アウトされてアクティブになる。したがって、例外ハンドラでは、必要に応じて I/O ポート 0F0H にアクセスすることにより、x87 FPU

例外条件をクリアする（FERR# をディアサートにする）前に IGNNE# をアクティブにできる。

ただし、この図の回路の場合、x87 FPU 例外ハンドラ内の処理順序に関係なく、例外ハンドラ終了時にハードウェアが正しいステートになるよう保証されている。プロセッサに IGNNE# 信号をドライブするフリップフロップ #2 には、FERR# を反転した信号が CLEAR 入力として入っている。そのため、FERR# が非アクティブのときには IGNNE# がアクティブになることは決してない。したがって、例外ハンドラから I/O ポート 0F0H にアクセスする前に x87 FPU 例外条件をクリアすると、IGNNE# はアクティブになることはなく、IGNNE# がアクティブになったまま例外ハンドラを終了することもない。

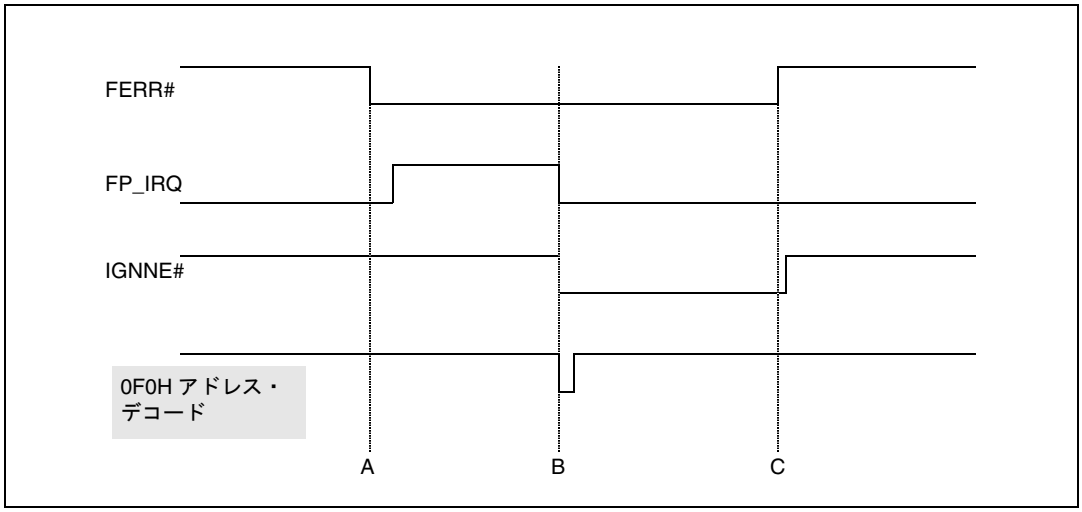


図 D-2. x87 FPU 例外処理時の信号状態

D.2.1.3. 非同期型命令のウインドウ内の x87 FPU 割り込み

インテル® Pentium® プロセッサと Intel486™ プロセッサでは、「非同期型」浮動小数点命令（FNINIT、FNCLEX、FNSTENV、FNSAVE、FNSTSW、FNSTCW、FNENI、FNDISI、FNSETPM）は、MS-DOS* 互換モードで次のように実行される。（非同期型命令の詳細については、の 8.3.11. 項「x87 FPU 制御命令」と 8.3.12. 項「同期型命令と非同期型命令」を参照のこと。）

先に実行した x87 FPU 命令によってマスクされていない数値例外がペンディング状態になっているときにその例外の冒頭で非同期型命令を実行しようとすると、他の x87 FPU 命令を実行しようとした場合と同様に、例外に応答して FERR# ピンがアサートされる。ところが、他の x87 FPU 命令とは違って FERR# はすぐにディアサートに戻される。これは、ペンディング状態の数値例外による割り込みを無視して非同期型命令を実行できるようにするためである。ただし、ごく短時間の FERR# のアサートであって

も、ほとんどのハードウェア・インターフェイス（インテルの推奨回路も含む）において x87 FPU 例外要求はラッチされてしまう。

どの x87 FPU 命令も実行時には、プロセッサが外部割り込みのサンプリングと受信のためのウインドウをオープンする。ペンディング状態の割り込みがあれば、プロセッサは先に割り込みを処理してから命令の実行を再開する。その結果、非同期型の浮動小数点命令が、マスクされていないペンディング状態の数値例外のイベントにおいて FERR# をアサートしたために生じた外部割り込みを受け入れる可能性がある。このことはマニュアルの非同期型命令の説明には明記されていない。この過程を図 D-3. に説明する。

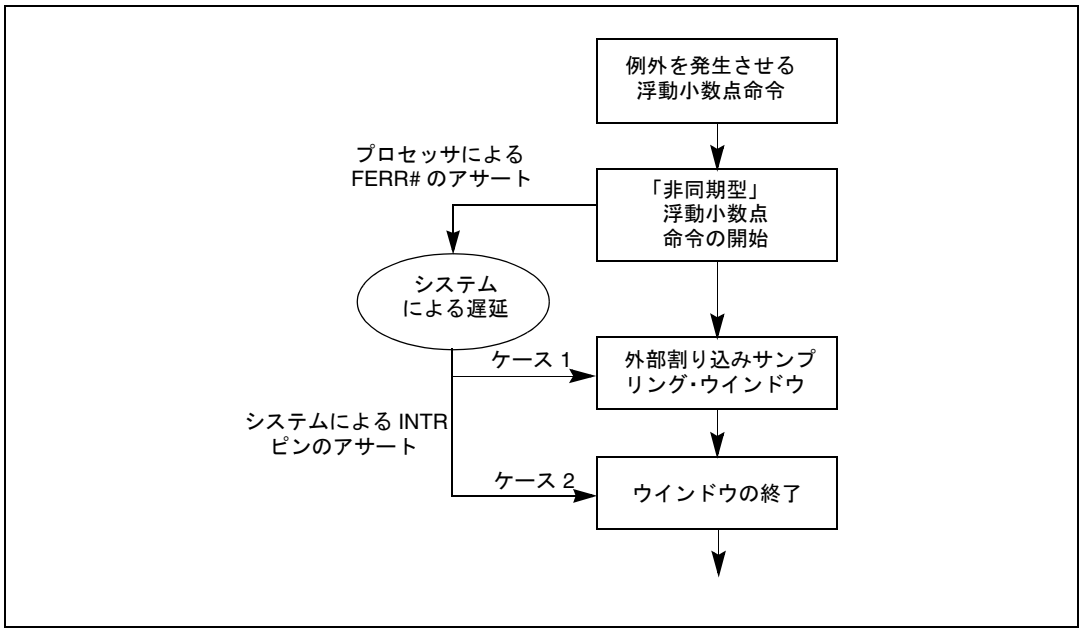


図 D-3. 外部割り込みの受信タイミング

図 D-3. では、D.2.1.1. 項「FERR# 信号発生時の基本規則」説明した「据え置き」方式でエラーを発生する浮動小数点命令が、マスクされていない数値例外を起こした場合を示している。据え置き方式の場合は、次の浮動小数点命令に遭遇したときに初めて FERR# ピンがアサートされる。この命令の次の浮動小数点命令が非同期型の浮動小数点命令の場合には、プロセッサはその非同期型浮動小数点命令に遭遇した時点で FERR# ピンをアサートする。非同期型浮動小数点命令は、FERR# ピンがアサートされた後で、ペンディング状態の外部割り込みをサンプリングするためのウインドウをオープンする。

その後の動作については、プロセッサが INTR ピン（FERR# ピンに応答してシステムがアサートしたもの）を介して割り込みを受信するタイミングによって、次の 2 つのケースが考えられる。

ケース 1　非同期型浮動小数点命令による FERR# ピンのアサートに応答してシステムがこのウインドウ内で INTR ピンをアサートした場合は、非同期型浮動小数点命令の実行を再開するより前に割り込みが処理される。

ケース 2　このウインドウの終了後にシステムが INTR ピンをアサートした場合は、次の命令境界で初めて割り込みが認識される。

ケース 1 以外にも、非同期型浮動小数点命令が自分の割り込みウインドウ内で数値例外を処理する場合が 2 つある。1 つは、最初の浮動小数点エラー条件が D.2.1.1. 項「FERR# 信号発生時の基本規則」で説明した「即時」方式のもので FERR# が即座にアサートされる場合である。システムが INTR をアサートするまでに要する時間が非同期型浮動小数点命令を実行するまでの時間に比べて長いと、INTR は後者の割り込みウインドウ内でアサートされる。もう 1 つは、2 つの非同期型 x87 FPU 命令が連続しているときに、前者の x87 FPU 命令でマスクされていない数値例外が発生した場合である。この場合、最初の非同期型命令でトリガされた FERR# 信号に対する INTR のアサートのタイミングが遅くて最初の命令の割り込みウインドウに入っていなければ、2 番目の割り込みウインドウ内で識別されることになる。

非同期型 x87 FPU 命令がインテルの本来の設計方針どおりに使用されれば、上記のような非同期型 x87 FPU 命令の実行で問題が起こることはない。本来、非同期型命令は x87 FPU 例外ハンドラ内で使用することを前提としており、エラー条件をクリアする前に x87 FPU を操作できるようにするためのものである。すなわち、非同期型命令ならば x87 FPU エラー条件のためにプロセッサがハング状態になることもなく、IGNNE# をアサートする必要もない。非同期型命令が正しく機能するのは、エラー条件がクリアされるまで x87 FPU エラー・ハンドラを呼び出す原因となった FERR# がアサートされたままであるからである。FERR# がすでにアサートされているので、非同期型命令のところで FERR# をごく短時間アサートするロジックによる悪影響は何もないはずである。また、例外ハンドラの中でエラー条件をクリアした後で非同期型命令を実行した場合は、FERR# が全くアサートされないので、やはり問題はない。

x87 FPU 例外ハンドラの外で非同期型命令を実行すると、実際のハードウェア・インターフェイスとプロセッサの種類にもよるが、上記のような問題が起こる場合がある。PUSHFD、CLI、非同期型、POPFD という命令シーケンスの場合、非同期型命令のウインドウ内の割り込みがブロックされることがある（この命令シーケンスでは、CLI により割り込みをブロックし、フラグをプッシュしポップすることにより、割り込みフラグの元の値を保存しリストアする）。ただし、非同期型命令によって FERR# がトリガされれば、ラッチされた値も PIC からの応答も引き続き有効になる。必要に応じて、コード上でこの状態をチェックして修正することもできる。この問題の詳細

と解決方法については、D.3.6. 項「タスク間で x87 FPU を共有する場合の注意事項」を参照のこと。

D.2.2. P6 ファミリおよびインテル® Pentium® 4 プロセッサにおける MS-DOS* 互換モード

CR0 の NE ビットが 0 の場合の、P6 ファミリおよびインテル® Pentium® 4 ファミリ・プロセッサの MS-DOS* 互換モードで提供される FERR# と IGNNE# の機能は、Intel486 プロセッサとインテル® Pentium® プロセッサの場合とほぼ同じである。D.2.1.2. 項「MS-DOS* 互換モードをサポートするための推奨外部ハードウェア」に示した推奨回路は、P6 ファミリおよびインテル Pentium 4 プロセッサでも従来のプロセッサの場合と同様に使用できる。MS-DOS 互換モードの x87 FPU 例外処理において、P6 ファミリおよびインテル Pentium 4 プロセッサが唯一異なるのは、どの x87 FPU 命令の例外でも必ず即時方式でエラー通知が行われることである。つまり、インテル Pentium Pro プロセッサでは、x87 FPU がマスクされていない例外を検出すると同時に FERR# がアサートされる。エラー通知が次の x87 FPU 命令や WAIT 命令まで据え置かれることはない。

（D.2.1.1. 項「FERR# 信号発生時の基本規則」で説明したように、Intel486 プロセッサとインテル Pentium プロセッサの場合は、ほとんどの例外が据え置き方式で通知される。）

マスクされていない x87 FPU エラーを検出すると同時に FERR# がアサートされるといっても、要求された割り込みがコード・シーケンスにおける次の命令より先に処理されるとは限らない。P6 ファミリおよびインテル Pentium 4 プロセッサでは複数の命令を同時に実行するからである。また、実際の外部ハードウェアにもよるが、プロセッサからの FERR# アサートと、それに応答してプロセッサに INTR をアサートするまでの間の遅延時間がある。さらに、PIC への割り込み要求（IRQ13）がオペレーティング・システムにより一時的にブロックされたり、より優先度の高い割り込みによって待機させられたり、オペレーティング・システムが EFLAGS の IF ビットをクリアしたことによりプロセッサの INTR に対する応答そのものがブロックされていたりする場合もある。ただし、ストリーミング SIMD 拡張命令数値例外は、（CR0.NE の値に関係なく）FERR# をアサートしない。また、IGNNE# のアサート/ディアサートを無視する。

もちろん、IGNNE# 入力が非アクティブであれば、Intel486 プロセッサとインテル Pentium プロセッサと同様に、その前の x87 FPU 命令で発生したマスクされていない浮動小数点例外により、次の WAIT 命令や x87 FPU 命令（ただし、非同期型命令は除く）に遭遇した時点でプロセッサが即座にフリーズする。つまり、すでに発生している例外のためにまだ x87 FPU 例外ハンドラが呼び出されていない（したがって、x87 FPU の例外ステートがクリアされていない）場合は、次の WAIT 命令や x87 FPU 命令を実行する前に、プロセッサは例外ハンドラを呼び出して例外処理を待機させられる。

D.2.1.3. 項「非同期型命令のウインドウ内の x87 FPU 割り込み」で説明したように、Intel486 プロセッサとインテル Pentium プロセッサの場合は、非同期型命令を x87 FPU 例外ハンドラの外で使用すると、その前の x87 FPU 命令で発生したマスクされていない例外が、各 x87 FPU 命令実行開始時にオープンされる外部割り込みサンプリング・ウインドウ内で取り込まれて、受け入れられることがある。P6 ファミリおよびインテル Pentium 4 プロセッサの場合は、非同期型の x87 FPU 命令群からサンプリング・ウインドウが削除されたので、このようなことは起こらない。

D.3. MS-DOS* 互換モードのハンドラに対する推奨規則

計算プログラムの動作は大きく 2 つの部分に分けられる。プログラム制御部分と演算部分である。プログラム制御部分では、実行する機能の決定、数値オペランドのアドレス計算、ループ制御などを行う。これに対して演算部分では、単純な加算、減算、乗算などの演算を数値オペランドに対して行う。プロセッサは、この 2 つの部分を独立させて効率よく処理できるように設計されている。x87 FPU 例外ハンドラは、システムによってインプリメントは変わってくるが、プログラム制御コードの中で最も複雑な部分の 1 つといえる。

D.3.1. 浮動小数点例外とそのデフォルト動作

浮動小数点命令の実行中に x87 FPU が識別する浮動小数点例外条件には、次の 6 種類がある。

1. #I － 無効操作
   #IS － スタック・フォルト
   #IA － IEEE 規定による無効操作
2. #Z － ゼロ除算
3. #D － デノーマル・オペランド
4. #O － 数値オーバーフロー
5. #U － 数値アンダーフロー
6. #P － 不正確結果（精度）

各例外の詳細とそのデフォルト動作については、8.4. 節「x87 FPU 浮動小数点例外処理」と 8.5. 節「x87 FPU 浮動小数点例外条件」を参照のこと。

D.3.2. 数値例外処理の 2 つのオプション

数値例外が発生したときにプロセッサが次のどちらの方法で例外を処理するかは、ソフトウェアのシステム設計者が決定する。

- x87 FPU 自身が、ほとんどの場合に有効なデフォルトの処理方法にしたがって選択された例外を処理する。この方法を取ると、例外の発生によって中断されることなく計算プログラムの実行を続行できる。プログラム上では、各例外タイプを個別にマスクして、例外が発生してもこのように安全かつ妥当な結果が x87 FPU から得られるようにできる。デフォルトの例外処理動作は例外を発生させた命令の一部として x87 FPU が行うので、マスクされている例外の処理に時間がかかり過ぎない限り、外部には例外の発生は通知されない。マスクされている例外が検出されると、数値ステータス・レジスタのフラグがセットされるが、いつどこでセットされたかに関する情報は保存されない。
- x87 FPU 自身が例外を処理するのではなく、ソフトウェア例外ハンドラを呼び出して例外を処理する方法もある。数値例外がマスクされていないときに数値例外が発生すると、x87 FPU は以降の数値命令の実行を停止し、ソフトウェア例外ハンドラに分岐する。例外ハンドラでは、x87 FPU が検出した数値例外に合わせて任意の回復プロシージャを実行できる。

D.3.2.1. マスクによる自動例外処理

x87 FPU ステータス・ワードには、上記の 6 種類の例外条件のそれぞれに対応するフラグビットがあり、x87 FPU 制御ワードには各例外条件に対応するマスクビットがある。制御ワードの対応するマスクビットが 1 にセットされて例外がマスクされると、プロセッサはデフォルトの例外処理を適宜行い、計算を続行する。

プロセッサでは、発生し得るすべての例外条件に対するデフォルトの例外処理が決められている。マスクされている例外処理は、ほとんどの数値計算アプリケーションに安全に適用できるように設計されている。

例えば、不正確結果（精度）例外をマスクした場合、正確に表現できない演算結果を x87 FPU がどのように処理するかについて、システムは 4 つのモード、すなわち、通常の丸め、ゼロ方向への切り捨て、切り上げ、切り下げから選択できる。また、アンダーフロー例外をマスクした場合、x87 FPU は小さすぎて正規形式で表現できない数値を非正規（デノーマル）形式で、あるいは、デノーマルでも小さすぎる場合はゼロとして格納する。例外がマスクされていると、x87 FPU は 1 つの命令で複数の例外を検出する場合があることに注意しなければならない。x87 FPU はマスクされている例外を処理した後でその命令を続行するからである。例えば、x87 FPU がデノーマル・オペランドを検出すると、この例外に対してマスクされている例外処理を行い、その結果アンダーフローを検出する。

デフォルトの例外処理を使用して重大な例外を安全で自動的に処理する例として、異なる値の抵抗を並列につないだ回路の抵抗値を 1 つの計算式で求める場合を考えてみる（図 D-4. を参照）。R1 がゼロであると、この回路の抵抗値もゼロになってしまう。ゼロ除算例外と精度例外をマスクしておけば、プロセッサから正しい結果が返される。R1 を 1 にして FDIV 命令を実行すると結果は無限大になり、（無限大 +R2+R3）を 1 にして FDIV 命令を実行すると結果はゼロになる。

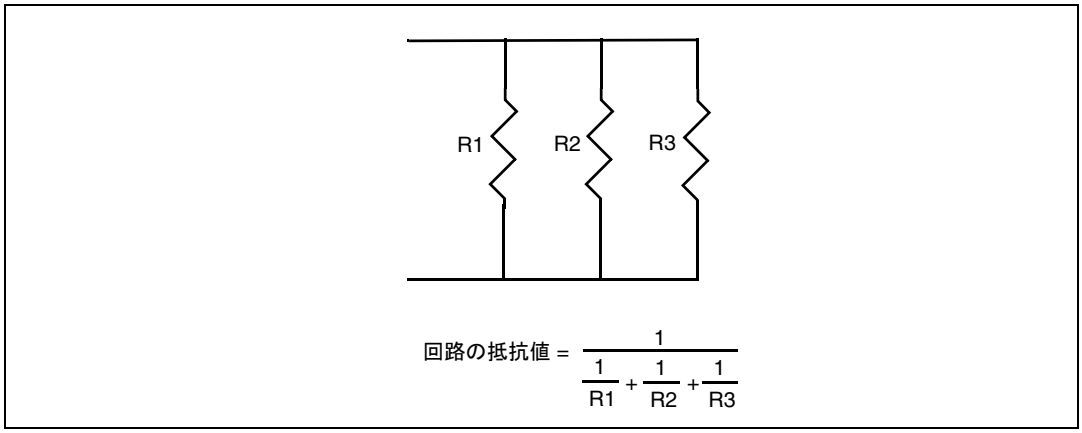


図 D-4. 無限大を使用する計算の例

プログラマは x87 FPU 制御ワードの個々の数値例外をマスクするかどうかを決定するときには、ほとんどの例外処理をプロセッサにまかせて、特に重大な例外だけに対して例外ハンドラをプログラムして処理できる。例外ハンドラ・ソフトウェアを作成するのは難しいので、例外をマスクしてしまえば各例外条件に対して十分妥当な結果を簡単に得ることができる。ほとんどのアプリケーションでは、すべての例外をマスクすることにより、最小のプログラミング労力で満足できる結果が得られる。ソフトウェア開発時のデバッグ段階では、いくつかの例外をマスクしないでテストし、ソフトウェアが完成してからマスクする方法も有効である。例えば、無効操作例外は一般にプログラム・エラーであり、プログラムを修正しなければならないからである。

x87 FPU ステータス・ワードの例外フラグには、各フラグが最後にクリアされてから発生した例外の記録が残っている。例外フラグは一度セットされると、FCLEX/FNCLEX（clear exceptions）命令を実行するか、FINIT/FNINIT 命令か FSAVE/FNSAVE 命令で x87 FPU を再初期化するか、FRSTOR 命令または FLDENV 命令でフラグを上書きするまでクリアされない。したがって、プログラマは例外をすべてマスクして計算を実行してから、ステータス・ワードを調べて計算中に何らかの例外が検出されたかを確認できる。

D.3.2.2. ソフトウェアによる例外処理

インテル® 286 プロセッサ以降の IA-32 プロセッサで使用する x87 FPU が、MS-DOS* 互換モードで動作するシステムで IGNNE# がアサートされていないときにマスクされていない例外条件を検出すると、PIC およびプロセッサの INTR ピンを通してソフトウェア例外ハンドラが呼び出される。エラー条件が最初に検出された時点、またはプロセッサが次の WAIT 命令か x87 FPU 命令に遭遇した時点で、x87 FPU からの FERR#（または ERROR#）出力により、プロセッサは例外ハンドラを呼び出す。どの時点で例外ハンドラが呼び出されるかは、プロセッサの種類、例外の種類、例外をトリガした x87 FPU 命令の種類によって決まる（D.1. 節「MS-DOS* 互換モードの x87 FPU 例外処理メカニズムの由来」と、D.2. 節「Intel486™ プロセッサ、インテル® Pentium® プロセッサ、P6 プロセッサ・ファミリおよびインテル® Pentium® 4 プロセッサにおける MS-DOS* 互換モード」を参照）。最初のエラー信号から x87 FPU 例外ハンドラの呼び出しまでに要する時間は、当然、外部ハードウェア・インターフェイスによって異なるが、x87 FPU エラーによる外部割り込みが許可されているかどうかによっても異なる。ただし、インテル® アーキテクチャでは、マスクされていない浮動小数点例外が発生すると次の WAIT 命令か浮動小数点命令が実行される直前にプロセッサがフリーズするので（ただし、IGNNE# 入力がアクティブな場合と、次の浮動小数点命令が非同期型 x87 FPU 命令の場合は除く）、遅くとも次の WAIT 命令か浮動小数点命令が実行される前に例外ハンドラが呼び出されるようになっている。

フリーズ状態のプロセッサは外部割り込みを待機するが、外部割り込みは、プロセッサ（またはコプロセッサ）からの FERR#（または ERROR#）出力に応答して、通常は外部ハードウェアによって「スレーブ」PIC の IRQ13 および INTR を通してなされなければならない。この外部割り込みにより、例外ハンドラルーチンが呼び出される。プロセッサが x87 FPU 命令を実行するときに x87 FPU エラーによる外部割り込みがディスエーブルになっている場合は、マスクされていない x87 FPU 例外条件が有効になっていると、プロセッサはイネーブルの別の割り込みが発生するまでフリーズする。NE=0 の場合でも IGNNE# 入力がアクティブならば、プロセッサは例外を無視して処理を続行する。外部割り込みによるエラー通知は、MS-DOS との互換性のためにサポートされている。MS-DOS との互換性の詳細については、『IA-32 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、下巻』の第 18 章「インテル® アーキテクチャの互換性」を参照のこと。

上記の説明の中で x87 FPU からの ERROR# 出力というのは、インテル® 387 と
インテル® 287 数値演算コプロセッサ（NPX チップ）の場合である。このどちらかのコプロセッサがマスクされていない例外条件を検出すると、プロセッサとコプロセッサをつなぐ ERROR# ステータス・ラインを使用して、インテル 286 プロセッサと Intel386™ プロセッサに例外を通知する。x87 FPU の種類による例外処理の違いについては、この付録の D.1. 節「MS-DOS* 互換モードの x87 FPU 例外処理メカニズムの由来」と、『IA-32 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、下巻』の第 18 章「インテル® アーキテクチャの互換性」を参照のこと。

例外処理ルーチンは、一般に、システム・ソフトウェアに属するものである。例外処理ルーチンでは、ペンディング状態の浮動小数点例外が存在すると、完了できない浮動小数点命令を実行する前に、x87 FPU ステータス・ワード内のアクティブな例外フラグをクリアする（ディスエーブルにする）必要がある。そうしないと、浮動小数点命令により再びx87 FPU割り込みがトリガされ、システムは浮動小数点例外がネストした無限ループに入ってハングしてしまう。どのようなイベントの場合も、例外処理ルーチンは例外処理を行った後、IRET(D）より前に、x87 FPUステータス・ワード内のアクティブな例外フラグをクリアする必要がある。一般に、例外に対する処理として行う作業を次に示す

- 後に表示/印刷ができるように、例外カウンタをインクリメントする。
- 診断情報（例えば、x87 FPU環境とレジスタ状態）を表示/印刷する。
- 以降の実行を中止する。または、例外ポインタを使用して例外を発生させない命令を構築して実行する。

アプリケーション・プログラマは、オペレーティング・システムのリファレンス・マニュアルで数値例外に対するシステムの対処法が適切かどうかを調べる必要がある。ソフトウェア例外ハンドラの作成方法については、本付録の D.3.4. 項「x87 FPU 例外ハンドラの例」の他、『IA-32 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、下巻』の第5章「割り込みと例外の処理」も参照のこと。

D.2.1.2. 項「MS-DOS* 互換モードをサポートするための推奨外部ハードウェア」で説明したように、FERR# と INTR を結ぶハードウェア・インターフェイスの初期のものの中には、推奨回路ほど安全でないものがある。その理由は、例外ハンドラがx87 FPUそのものから例外をクリアすることでFERR#をディアサートする前に、例外ハンドラがポート0F0Hにアクセスして PICへのx87 FPU例外割り込み要求をクリアしなければならないからである。このような初期のハードウェアで発生し得る問題を避けるために、インテルでは、x87 FPU 例外ハンドラでx87 FPU からエラー条件をクリアする前に必ずポート0F0Hにアクセスすることを推奨している。

D.3.3. x87 FPU 例外ハンドラの使用時に必要な同期

x87 FPU例外ハンドラを使用する場合は、同時性あるいは同期を管理して、x87 FPUが使用した値をプロセッサが変更する前に例外をチェックする必要がある。誤ったコンテキストを使用すると、ほとんどの数値命令で数値例外が発生する可能性があることに注意しなければならない。

D.3.3.1. 例外処理で同期の必要な対象、理由、タイミング

例外処理で同期を取るのは、例外ハンドラが例外を処理するときに、例外が起こったコンテキストで例外を調べて処理することである。並列処理がなされている場合は、プロセッサが例外を認識した時点のプロセッサ状態（コンテキスト）と、例外が発生した時点のコンテキストが異なっている場合が多い。プロセッサが内部レジスタの内容を変えてしまっていたり、例外発生時とは全く別のプログラムが実行されていたりする場合もある。例外ハンドラが元のコンテキストをリストアできなければ、例外の原因を突き止めたり、例外から適切に回復したりすることはできない。このような問題を解決するために、x87 FPU には特殊なレジスタがある。これらのレジスタは数値命令を開始した時点で更新され、エラーが起こった命令実行時の数値プログラム状態が記録されている。

例外ハンドラではこれを利用して例外発生時のコンテキストをリストアできるが、アプリケーション・コードを開発するときも同期を念頭に置いておく必要がある。結局、例外処理において同期を取ることにより、マスクされていない数値例外が発生して例外ハンドラが呼び出されたときに、x87 FPU などコンテキストに関係する各要素が定義されているとおりの状態になっていることが保証できなければならない。

x87 FPU がマスクされていない例外条件を通知するのは、助けを求めていることである。すなわち、例外をマスクしないのは、x87 FPU の数値演算規則とプログラミング規則にしたがってそれ以降の計算プログラムを実行しても正しい結果を出せない意味である。そのような例外が発生した場合は、正しく同期させて例外を処理しなければ、プログラムの実行結果が信頼できなくなる。

高級言語においては、コンパイラによって自動的に必要な同期が取られるが、アセンブリ言語においては、プログラム上で例外処理の同期を取らなければならない。計算プログラムを十分テストしデバッグして数値例外が発生しなくなったとしても、異なるシステムや計算環境に移行すると例外が多発する可能性があることは周知の事実である。その一例として、プログラムの設計とテストを行ったシステムと、実際にプログラムを走らせるシステムとで使用可能な数値範囲が異なる場合が考えられる。D.3.3.2. 項「例外処理の同期の例」の例 D-1. と例 D-2. には、予測不可能な例外に対する巧緻な対処例を示す。

D.3.1. 項「浮動小数点例外とそのデフォルト動作」で説明したように、ソフトウェアのシステム設計者の決定にしたがって、数値例外が発生するとプロセッサは次のどちらかの方法で例外を処理する。

- x87 FPU 自身がデフォルトの処理方法にしたがって選択された例外を処理する。すべての例外に対して x87 FPU がデフォルトの処理を行う場合、例外の同期を取る必要性は明確ではない。ただし、コードは設計時とは別の環境やオペレーティング・システムに移植されることが多いので、x87 FPU を使用してコードを作成するときは、次の例のように、常に例外処理の同期を考慮するのが安全である。

- ソフトウェア例外ハンドラを呼び出して例外を処理する。数値例外がマスクされていないときに数値例外が発生すると、x87 FPU はそれ以降の数値命令の実行を停止して、ソフトウェア例外ハンドラに分岐する。x87 FPU 例外ハンドラを呼び出す場合は、常に同期を考慮しないと信頼性のある結果は得られない。

数値計算コードを作成するときは、当初は数値例外をマスクして実行する予定であっても、常に例外の同期を考慮しなければならないことを次の例 D-1. と例 D-2. で説明する。

D.3.3.2. 例外処理の同期の例

次の例は、3 つの命令を使用して、整数値をロードし、平方根を計算し、整数値をインクリメントするものである。FILD 命令で例外が発生しない限り、x87 FPU の同期化された処理により、INC COUNT がプロセッサで並列に実行され、どちらのプログラムでも正しい結果が得られる。ところが、例外がマスクされていない環境にこのコードを移して実行すると、例 D-1. に示すコードは正しく処理されない。

例 D-1. 誤ったエラー同期の例

```
FILD   COUNT  ; x87 FPU 命令
INC    COUNT  ; 整数命令でオペランドを変更する
FSQRT         ; 次の x87 FPU 命令 --
              ; 先の x87 FPU 命令によるエラーはここで検出される
```

例 D-2. 適切なエラー同期の例

```
FILD   COUNT  ; x87 FPU 命令
FSQRT         ; 次の x87 FPU 命令 --
              ; 直前の x87 FPU 命令によるエラーはここで検出される
INC    COUNT  ; 整数命令でオペランドを変更する
```

x87 FPU をサポートするオペレーティング・システムの中には、数値レジスタスタックがメモリに拡張されるものがある。x87 FPU スタックをメモリに拡張する場合は、無効操作例外はマスクされない。フル・レジスタにプッシュするか、空レジスタからポップすると、SF（Stack Fault）フラグがセットされ、無効操作例外が発生する。この例外に対する回復ルーチンでは、原因を調べてスタックを修復し、元の操作を再開しなければならないが、例 D-1. ではこの回復ルーチンが正しく機能しない。なぜなら、例外ハンドラが呼び出される前に COUNT の値がインクリメントされるために、回復ルーチンが誤った COUNT 値をロードして、誤った結果あるいは信頼できない結果になるからである。

D.3.3.3. 例外処理の一般的な同期方法

D.2.1.2. 項「MS-DOS* 互換モードをサポートするための推奨外部ハードウェア」で説明したように、x87 FPU がマスクされていない例外条件を検出すると、次の WAIT 命令または浮動小数点命令を実行する前にソフトウェア例外ハンドラが呼び出される。これは、マスクされていない浮動小数点例外が発生すると、WAIT 命令や浮動小数点命令を実行する直前にプロセッサがフリーズするからである（ただし、IGNNE# 入力がアクティブな場合と、非同期型 x87 FPU 命令の場合は除く）。例外が検出されてから次の WAIT 命令または x87 FPU 命令までのインターバルのどの時点で例外ハンドラが呼び出されるかは、プロセッサの種類、システム、x87 FPU 命令と例外の種類よって異なる。

例外処理を確実に同期させるには、例外ハンドラがこのインターバルの最後に呼び出された場合を想定する。そうすると、プログラム上では、例外ハンドラが必要とする可能性のある値（例えば、例 D-1. と例 D-2. の COUNT 値）はすべて、エラーが発生したかもしれない x87 FPU 命令の次の x87 FPU 命令が終了するまで変更できなくなる。このような値を次の x87 FPU 命令より前に変更しなければならない場合（あるいは、次の x87 FPU 命令でもエラーが発生するかもしれない場合）は、その値を変更する前に WAIT 命令を挿入する。WAIT 命令は、アプリケーション内の最後の浮動小数点命令の後にも入れておくとよい。こうすると、アプリケーションが完了する前に、マスクされていない例外がすべて処理されるからである。

D.3.4. x87 FPU 例外ハンドラの例

例外ハンドラの作成にはさまざまな手法が考えられる。その一例として、例外ハンドラ・プロシージャを「プロローグ」、「ボディ」、「エピローグ」の 3 つの部分で構成する方法を説明する。

INTR、NMI、SMI が原因で制御が例外ハンドラに渡されると、ハードウェアにより外部割り込みがディスエーブルにされる。プロローグ部では、優先度が高いソースの割り込みから保護しなければならない処理を行う。すなわち、レジスタ状態を保存したり、診断情報を x87 FPU からメモリにコピーしたりする。プロローグ部でクリティカルな処理が完了したら、再び割り込みを可能にして、優先度が高い割り込みハンドラが例外ハンドラに割り込めるようにする。標準的なプロローグ部では、レジスタ状態をセーブし診断情報を x87 FPU からメモリにコピーするだけでなく、ステータス・ワード内の浮動小数点例外フラグもクリアする。あるいは、例外ハンドラを再入可能にする必要がなければ別の手法も利用できる。この場合、プロローグ部で例外フラグはクリアしないため、ボディ部には、ペンディング状態の浮動小数点例外が存在すると完了できない浮動小数点命令は入れてはならない。（非同期型命令については、8.3.12. 項「同期型命令と非同期型命令」を参照のこと。）ただし、その場合も IRET 命令を実行する前に例外フラグをクリアしなければならない。このどちらの手法にも従わなければ、システムは浮動小数点例外がネストした無限ループに入ってハングしてしまう。

例外ハンドラのボディ部では、診断情報を調べ、そのアプリケーションで必要な処理を行う。実際には、アプリケーションの実行を停止する、メッセージを表示する、問題を解決して通常の実行を再開する、などのさまざまな処理が考えられる。エピローグ部では、基本的にはプロローグ部と反対の処理を行う。すなわち、プロセッサ状態をリストアして、通常の実行を再開できるようにする。エピローグ部では、マスクされていない例外フラグをx87 FPUにロードしてはいけない。そうすると、ロードした時点で別の例外が発生してしまうからである。

次の3つのコードは、例外ハンドラの骨組みをASM386/486でコーディングした例である。保存空間としては32ビット・プロテクト・モードで適正なサイズを確保している。次の3つの例では、さまざまな状況に対応するためのプロローグ部とエピローグ部の作成方法を示すが、アプリケーションに応じて異なるボディ部についてはその挿入位置だけをコメントで示している。

最初の2つの例はよく似ており、大きな違いはx87 FPUのセーブとリストアに使用する命令だけである。FNSAVE　命令を使用すると詳しい診断情報が得られるが、FNSTENV命令を使用する方が高速である（さらに、FNSAVE命令は元のx87 FPU情報をセーブした後でx87 FPUを再初期化するが、FNSTENV命令は全x87 FPU例外をマスクするだけである）。割り込み処理に要する時間が問題になるようなアプリケーションや、レジスタ内容を調べる必要がないアプリケーションの場合は、FNSTENV命令を使用する方が、プロセッサが別の割り込み要求を受け入れない「クリティカル・リージョン」時間を短縮できる（x87 FPUセーブのイメージの詳細については、8.1.9.項「FSTENV/FNSTENV命令およびFSAVE/FNSAVE命令によるx87 FPUのステートのセーブ」を参照のこと）。プロセッサとオペレーティング・システムの両方がストリーミングSIMD拡張命令をサポートしている場合は、FNSAVE命令の代わりにFXSAVE命令を使用するべきである。FXSAVE命令を使用する場合は、ステート全体をセーブできるように、セーブ領域が512バイトに拡大され、アライメントが16バイトに合わされている必要がある。これらのステップにより、すべてのコンテキストが確実にセーブされる。

例外ハンドラのボディ部の後のエピローグ部では、割り込み発生ポイント（すなわち、マスクされていない例外を発生した命令の直後の命令）から実行を再開できるようにプロセッサを準備する。x87 FPUにロードされるメモリ・イメージ内の例外フラグは、再ロードする前にクリアしておかなければならないことに注意する（ここに示す例では、ステータス・ワード・イメージ全体をクリアしている）。

例D-3.と例D-4.では、例外ハンドラそのものがマスクされていない例外を発生させることはないものと想定している。例外が起こる可能性があれば、例D-5.の手法を採り入れる。すなわち、プロローグ部で完全なx87 FPUステートをセーブしてから新たに制御ワードをロードする。このタイプの例外ハンドラを設計する場合は、例外ハンドラが無限に再入されることのないように注意しなければならない。

例 D-3. FPU 全ステートをセーブするための例外ハンドラ

```
SAVE_ALL    PROC
;
; レジスタをセーブし、x87 FPU ステート・イメージをセーブするためのスタック空間を確保する
    PUSH    EBP
    .
    .
    MOV     EBP, ESP
    SUB     ESP, 108   ; 108 バイトの空間を確保する（32 ビット・プロテクト・モードのサイズ）
; x87 FPU 全ステートをセーブし、割り込み許可フラグ（IF）をリストアする
    FNSAVE  [EBP-108]
    PUSH    [EBP + OFFSET_TO_EFLAGS] ; 元のフラグをスタックのトップにコピーする
    POPFD               ; IF を x87 FPU 例外の前の値にリストアする
;
; アプリケーションに応じた例外処理コードをここに入れる
;
; メモリに入っているステータス・ワード内の例外フラグをクリアする
; 変更したステート・イメージをリストアする
    MOV     BYTE PTR [EBP-104], 0H
    FRSTOR  [EBP-108]
; スタック空間を解放し、レジスタの内容をリストアする
    MOV     EBP
    .
    .
    POP     EBP
;
; 割り込まれた計算を再開する
    IRETD
SAVE_ALL    ENDP
```

例 D-4. レイテンシを短縮するための例外ハンドラ

```
SAVE_ENVIRONMENTPROC
;
; レジスタをセーブし、x87 FPU 環境をセーブするためのスタック空間を確保する
    PUSH    EBP
    .
    .
    MOV     EBP, ESP
    SUB     ESP, 28   ; 28 バイトの空間を確保する（32 ビット・プロテクト・モードのサイズ）
; 環境をセーブし、割り込み許可フラグ（IF）をリストアする
    FNSTENV [EBP-28]
    PUSH    [EBP + OFFSET_TO_EFLAGS]   ; 元のフラグをスタックのトップにコピーする
    POPFD              ; IF を x87 FPU 例外の前の値にリストアする

;
; アプリケーションに応じた例外処理コードをここに入れる
;
; メモリに入っているステータス・ワード内の例外フラグをクリアする
; 変更した環境イメージをリストアする
    MOV     BYTE PTR [EBP-24], 0H
    FLDENV  [EBP-28]
; スタック空間を解放し、レジスタの内容をリストアする
    MOV     ESP, EBP
    .
    .
    POP     EBP
;
; 割り込まれた計算を再開する
    IRETD
SAVE_ENVIRONMENT ENDP
```

例 D-5. 再入可能な例外ハンドラ

```
    .
    .
LOCAL_CONTROL DW ? ; 初期化を想定
    .
    .
REENTRANT PROC
;
; レジスタをセーブし、x87 FPU ステート・イメージをセーブするためのスタック空間を確保する
    PUSH    EBP
    .
    .
    MOV     EBP, ESP
    SUB     ESP, 108   ; 108 バイトの空間を確保する（32 ビット・プロテクト・モードのサイズ）

; ステートをセーブし、新たに制御ワードをロードし、割り込み許可フラグ（IF）をリストアする
    FNSAVE  [EBP-108]
    FLDCW   LOCAL_CONTROL
    PUSH    [EBP + OFFSET_TO_EFLAGS] ; 以前のフラグをスタックのトップにコピーする
    POPFD               ; IF を x87 FPU 例外の前の値にリストアする

    .
    .
;
; アプリケーションに応じた例外処理コードをここに入れる

; 例外ハンドラを再入可能にするコードをここに入れる
; ローカルな保存領域が必要な場合は、スタック上に割り当てる
;
    .
    .
; メモリに入っているステータス・ワード内の例外フラグをクリアする
; 変更したステート・イメージをリストアする
    MOV     BYTE PTR [EBP-104], 0H
    FRSTOR  [EBP-108]
; スタック空間を解放し、レジスタの内容をリストアする
    MOV     ESP, EBP
    .
    .
    POP     EBP
;
; 割り込まれた場所に戻る
    IRETD
REENTRANT ENDP
```

D.3.5. x87 FPU と SMM を使用する場合の IGNNE# 回路ステートのセーブ

Intel486™ プロセッサ以降のプロセッサで MS-DOS* 互換モードの x87 FPU 例外処理を行う場合の推奨回路（図 D-1. を参照）には、2 つのフリップフロップが入っている。x87 FPU 例外ハンドラから I/O ポート 0F0H にアクセスすると、フリップフロップ #1 から出力される IRQ13 割り込み要求がクリアされ、さらにフリップフロップ #2 からの IGNNE# 信号もクロックアウトされアクティブになる。

IGNNE# のアサートは、例外ハンドラがペンディング状態の x87 FPU エラーを無視して x87 FPU 命令を実行させる場合に利用できる。ここで問題となるのは、フリップフロップ #2 のステートが隠れた追加ステータス・ビットとしてプロセッサの動作に影響することである。そのため、理想的には SMM に入るときにそのステートをセーブしておき、通常操作を再開するときにリストアできるとよい。これが行われず、また SMM コードでも x87 FPU ステートがセーブされ、かつ IGNNE# のアサートに依存する x87 FPU エラー・ハンドラが使用されていると（このような状況は非常にまれであるが）、x87 FPU ハンドラが自分自身の内部でネストされて正しく機能しなくなる。このような状況が発生する例を次に示す。

x87 FPU 例外ハンドラに次のコード・シーケンスが含まれているものとする。

```
FNSTSW save_sw      ; 非同期型 x87 FPU 命令を使用して x87 FPU ステータス・
                    ; ワードをセーブする
OUT     0F0H, L     ; IRQ13 をクリアし、IGNNE# をアクティブにする
    . . . .
FLDCW new_cw        ; IGNNE# がアクティブであると想定しているので、
                    ; x87 FPU エラーを無視して新たに CW をロードする。
                    ; IGNNE# がアクティブでなければ、
                    ; 非同期型でない他の x87 FPU 命令でも同じ問題が発生する。
     . . . .
FCLEX               ; x87 FPU エラー条件をクリアし、FERR# をオフにして、IGNNE# FF をリセットする
```

この例では、OUT 命令と FLDCW 命令の間でプロセッサが SMM に入った場合にのみ問題が起こる。このとき、SMM コードで FNSAVE 命令を使用して x87 FPU ステートをセーブすると、IGNNE# フリップフロップがクリアされる（FNSAVE 命令は x87 FPU エラーをクリアするので、FERR# がディアサートされるからである）。プロセッサが SMM から戻って FRSTOR 命令で x87 FPU ステートをリストアすると、FERR# は再びアサートされるが、IGNNE# フリップフロップはセットされない。その後で x87 FPU エラー・ハンドラが FLDCW 命令を実行すると、アクティブなエラー条件によりこの x87 FPU エラー・ハンドラが最初から再入され、正しく機能できなくなる。

この問題を解決するために、インテルでは次の 2 つの方法を推奨している。

1. SMM コード内での計算では x87 FPU を使用しない。（SMM が提供する通常のパワー管理機能およびセキュリティ機能では、x87 FPU 計算は不要である。x87 FPU 計算が必要

になる特殊な場合には、x87 FPU のかわりにスケーリングやエミュレーションを使用する。）そうすると、0V サスペンド状態に入る場合を除いて、SMM コード内で FNSAVE/FRSTOR 命令を使用しなくて済む（0V サスペンド状態の場合は、電力節約のために CPU を完全にオフにするので、完全な CPU ステートをセーブする必要がある）。

2. プロセッサが x87 FPU 計算を行っている途中、または割り込みが発生した直後には、システムが SMM コードを呼び出してプロセッサを 0 V サスペンド状態にしないようにする。通常のパワー管理プロトコルでは、この問題を避けるために、システム・アクティビティが一定期間発生していない場合にのみパワー・ダウン・ステートに入るようにしている。

D.3.6. タスク間で x87 FPU を共有する場合の注意事項

IA-32 アーキテクチャでは、タスクの切り替えと同時に x87 FPU ステートを切り替えることはせず、見込みによって据え置くことができる。そのため、切り替えられた別のタスクで実際に x87 FPU 命令が実行されるまでは、x87 FPU ステートを切り替えなくてよい。カーネル・タスクでは浮動小数点はほとんど使用せず、また、アプリケーションには浮動小数点を使用しないものと頻繁に使用するものが混在しているため、不必要に x87 FPU ステートの格納を行わないことで大幅に時間を節約できる。ただし、x87 FPU ステートの保存を見込みによって据え置くために、カーネルには、次の 3 つの余分な負担がかかることになる。

1. 現在実行中のスレッドが x87 FPU を所有しているとは限らないので、カーネルはどのスレッドが x87 FPU を所有しているのか常に追跡していなければならない。
2. カーネルは浮動小数点例外とそれを発生させたタスク関連づけなければならない。浮動小数点例外は他のシステム・アクティビティとは非同期に通知されるので、そのための特殊な処理が必要になる。
3. 疑似的な浮動小数点例外割り込みが発生することがあるので、カーネルはそれを識別して破棄する必要がある。

D.3.6.1. x87 FPU ステート保存の見込みによる据え置きの概要

マルチタスクをサポートするには、システム内の各スレッドに汎用レジスタの保存領域が必要である。さらに、各タスクに浮動小数点の使用を許可する場合には、x87 FPU スタック全体および制御ワードやステータス・ワードなどの関連する x87 FPU ステートをセーブできる大きさの x87 FPU 保存領域も必要になる（x87 FPU セーブのイメージの詳細については、8.1.9. 項「FSTENV/FNSTENV 命令および FSAVE/FNSAVE 命令による x87 FPU のステートのセーブ」を参照のこと）。プロセッサとオペレーティング・システムの両方がストリーミング SIMD 拡張命令をサポートしている場合は、x87 FPU ステートとストリーミング SIMD 拡張命令ステートを格納できるように、セーブ領域が拡大され、アライメントが合わされている必要がある。

タスクスイッチにおいては、サスペンドされるスレッドの保存領域に汎用レジスタがスワップ・アウトされ、再開されるスレッドの汎用レジスタがロードされる。この時点ではまだx87 FPUステートをセーブする必要はない。再開されるスレッドが次にサスペンド状態になるまでx87 FPUを使用しなければ、x87 FPUステートをセーブしたりロードしたりする必要はないからである。いくつものスレッドがx87 FPUを全く使用しないまま実行されるのは、よくあることである。

プロセッサはx87 FPUステートのセーブに対して見込みによって据え置くことができるように、割り込み7（Device Not Available: DNA）とCR0のビット3（Task Switched: TS）を使用する（『IA-32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、下巻』の第2章の「制御レジスタ」を参照）。ハードウェアがサポートするタスク・スイッチング・メカニズム（『IA-32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、下巻』の第6章の「タスク・スイッチング」を参照）を使用してタスクを切り替えると、TSビットがセットされる。ソフトウェアによるタスク・スイッチング[2]を使用するマルチスレッド・カーネルでは、CR0を読み込んでTSビットをセットし、1とビット3のORを計算してCR0に書き戻す[3]。切り替え後の新スレッドのコンテキストで浮動小数点命令を実行しようとすると、フォルトが起こって割り込み7が発生する。

その結果、DNAハンドラは古い浮動小数点コンテキストをセーブし、現在のスレッドのx87 FPUステートを再ロードする。DNAハンドラは終了前にCLTS命令を使用してTSビットをクリアする。DNAハンドラからリターンすると、フォルトを起こしたスレッドは、浮動小数点命令のところから実行を再開する。

オペレーティング・システムによっては、タスクスイッチによりタスク間のリニアアドレス空間も変わるという理由で、タスクスイッチのたびにx87 FPUのコンテキストをセーブするものもある。以降の各項で説明する問題点やその解決方法は、そのようなオペレーティング・システムにも当てはまる。

D.3.6.2. x87 FPU 所有者の追跡

x87 FPUの内容は現在実行しているスレッドのものとは限らないため、最後にx87 FPUを使用したユーザのスレッドIDを別にセーブしておく必要がある。この作業は簡単である。カーネルには、現在実行しているスレッドのIDを格納する変数とは別にx87 FPU所有者のスレッドIDを格納するための変数を持たせればよい。x87 FPU所有者の

2. ソフトウェアによるタスクスイッチの場合、オペレーティング・システムはサスペンドするスレッドのステートをセーブし、再開するスレッドのステートをリストアするのに、IA-32 アーキテクチャが提供する割り込み不可能な長期間の単一のタスクスイッチ操作ではなく、一連の命令シーケンスを使用する。

3. CR0 のビット 2（emulation flag: EM）も DNA 例外を発生するが、EM ビットを TS ビットのかわりに使用してはいけない。EM とは x87 FPU が使用できないので浮動小数点命令をエミュレーションしなければならないという意味である。EM を使用してタスクスイッチをトラップすると、IA の MMX テクノロジを使用する場合の互換性がなくなる。EM フラグがセットされると、MMX 命令では無効オペランド例外が発生する。

変数は、DNA 例外ハンドラが更新する。DNA 例外ハンドラは、この変数を使用して新旧スレッドの x87 FPU 保存領域を見付ける。DNA 例外ハンドラの概要を次に示す。

1. 「x87 FPU 所有者」変数を使用して、最後に x87 FPU を使用したスレッドの x87 FPU 保存領域を見付ける。
2. x87 FPU の内容を旧スレッドの x87 FPU 保存領域にセーブする。これには、通常、FNSAVE 命令または FXSAVE 命令を使用する。
3. 「x87 FPU 所有者」変数を、現在実行しているスレッドの ID に変更する。
4. 新スレッドの x87 FPU 保存領域から x87 FPU の内容をロードする。これには、通常、FRSTOR 命令または FXSAVE 命令を使用する。
5. CLTS 命令を使用して TS ビットをクリアし、DNA 例外ハンドラを終了する。

ここに示したのは見込みによる据え置き方式で x87 FPU ステートを切り替える場合（ステートのスワップ）の大筋であり、実際に安全かつ確実に行うには、さらに細かい配慮が必要である。

D.3.6.3. x87 FPU ステートのセーブと浮動小数点例外の関係

IA-32 アーキテクチャにおけるあらゆるインプリメントやあらゆる浮動小数点命令において、浮動小数点例外が発生するのは、例外を起こした浮動小数点命令の実行中から次の浮動小数点命令の直前までの間のどこかの時点であることが、これまでに説明されてきた。ここで、「次の」浮動小数点命令が、タスクスイッチのために x87 FPU ステートをセーブする FNSAVE 命令である場合を考える。FNSAVE 命令のような非同期型命令の場合、先に例外を起こした命令からの割り込み（NE=0 の場合）が非同期型命令の実行直前、実行中、実行後（システムによって異なる遅延時間のため）のどこで起こるかわからない。

したがって、x87 FPU ステートの切り替え中に浮動小数点例外が発生する可能性を考慮し、カーネルと浮動小数点例外割り込みハンドラでそれに対応できるようにする必要がある。

x87 FPU ステート切り替え中に受信した例外に対処する簡単な方法として、カーネルも x87 FPU 所有者スレッドの 1 つとみなす方法がある。カーネルが x87 FPU 所有者であることを示すには予約スレッド ID を使用する。「x87 FPU 所有者」変数は、x87 FPU ステート切り替え中は現在の所有者としてカーネルを設定しておき、x87 FPU ステート切り替えが完了してから新スレッドに変更する。数値例外ハンドラでは、x87 FPU 所有者を調べてカーネルが x87 FPU 所有者である場合には数値例外を破棄する。この状況に対処する DNA 例外ハンドラの概略フローを図 D-5. に示す。

x87 FPU ステート切り替えのためにカーネルが x87 FPU を所有しているときに数値例外を受信すると、カーネルはハンドラをディスパッチせずにこの数値例外を破棄しなければならない。数値例外ディスパッチ・ルーチンのフローを図 D-6. に示す。

このフローを見ると、x87 FPU ステート切り替え中に破棄された例外のために、浮動小数点例外が失われると思われるかもしれないが、x87 FPU ステートを再ロードするとこの例外が再発行される。ペンディング状態の数値例外がある場合とない場合の x87 FPU ステート切り替えを個別に見てみると、この 2 つのハンドラの動作の違いが分かる。

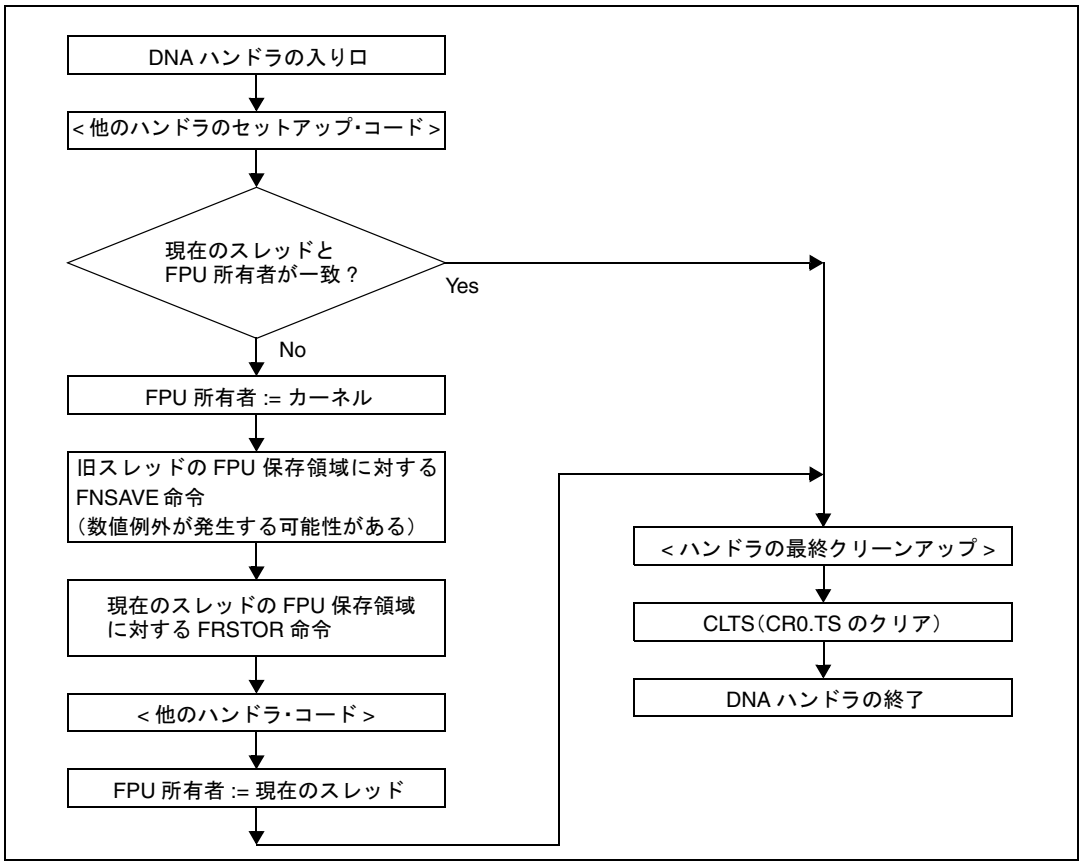


図 D-5. DNA 例外ハンドラの概略フロー

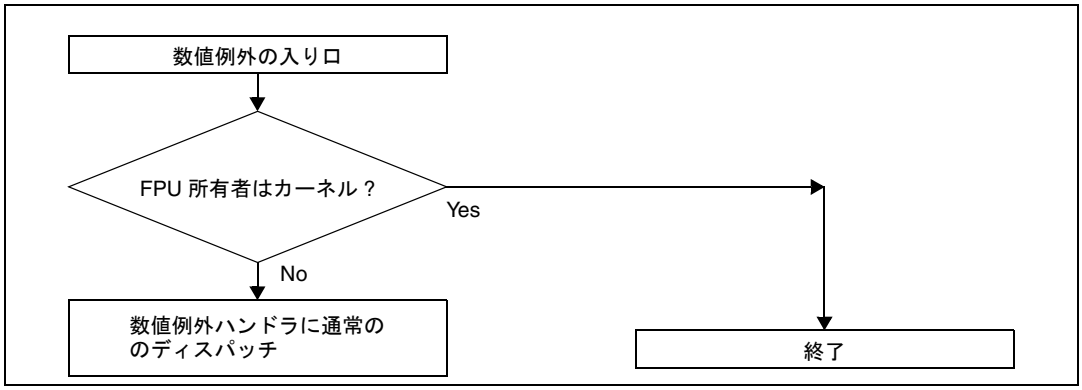


図 D-6. 数値例外ディスパッチ・ルーチンのプログラム・フロー

ケース 1: 数値例外がない場合の x87 FPU ステート切り替え

スレッド A とスレッド B の 2 つがあり、どちらも x87 FPU を使用するものとする。最後に浮動小数点命令を実行したのはスレッド A で、ペンディング状態の数値例外はないものとする。

スレッド B が現在実行中のスレッドとすると、スレッド A をサスペンドするときに CR0.TS がセットされる。スレッド B が浮動小数点命令を実行しようとすると、TS がセットされているので DNA 例外のフォルトが起こる。

ここで例外ハンドラが呼び出され、現在の x87 FPU 所有者と現在実行中のスレッドが異なることが分かる。x87 FPU ステート切り替えを無関係な数値例外から保護するため、x87 FPU 所有者をカーネルに設定する。旧所有者の x87 FPU ステートを FNSAVE 命令でセーブし、現在実行中のスレッドの x87 FPU ステートを FRSTOR 命令で再ロードする。例外ハンドラは終了前に x87 FPU 所有者をスレッド B に戻し、TS ビットをクリアする。

例外ハンドラが終了すると、スレッド B はフォルトした浮動小数点命令の実行を再開する。

ケース 2: 数値例外がある場合の x87 FPU ステート切り替え

同じく、スレッド A とスレッド B の 2 つがあり、どちらも FPU を使用するものとする。最後に浮動小数点命令を実行したのはスレッド A だが、今回はペンディング状態の数値例外が存在する場合を考える。現在実行中のスレッド B が浮動小数点命令を実行しようとすると、DNA 例外のフォルトが発生して、DNA ハンドラが呼び出される（数値例外と DNA 例外の両方がペンディング状態の場合は、数値例外を正しいコンテキストで処理できるように DNA 例外の方が優先される）。

ペンディング状態の数値例外があるので、FNSAVE 命令の開始時に FERR# により割り込みがトリガされる。システムによって異なる遅延時間後に数値例外ハンドラが実行されるので、FNSAVE 命令の実行前なのか実行直後なのかはわからない。このとき x87 FPU 所有者はカーネルなので、数値例外ハンドラは直ちに終了し、数値例外は破棄される。DNA ハンドラの実行が再開され、FNSAVE 命令によるスレッド A の古い FPU コンテキストのセーブと FRSTOR 命令によるスレッド B の x87 FPU コンテキストのリストアが完了する。

タスクスイッチ中に破棄された例外は、最終的にスレッド A で処理される。一定時間が経過すると、スレッド B はサスペンドされ、スレッド A の実行が再開される。スレッド A で浮動小数点命令を実行しようとすると、再び DNA 例外ハンドラが実行される。スレッド B の x87 FPU ステートには問題がないが、スレッド A の x87 FPU ステートには問題がある。ここで、スレッド A の保存領域から x87 FPU ステートをリストアすると、ペンディング状態の数値例外フラグが浮動小数点ステータス・ワードに再ロードされることに注意しなければならない。DNA 例外ハンドラからリターンすると、スレッド A がフォルトした浮動小数点命令の実行を再開し、即座に数値例外が発生し、通常どおり処理される。タスクスイッチと DNA 例外ハンドラによる x87 FPU ステート切り替えの結果、もう一度数値例外が発生するので、切り替え時に数値例外が破棄されても問題はない。

D.3.6.4. カーネルからの割り込みルーチン

MS-DOS* では、数値例外を処理するアプリケーションにおいては、割り込みベクタ・テーブルに自分のハンドラ・アドレスを書き込んで割り込み 16 をフックし、終了するときに割り込み 16 の元のハンドラにジャンプする。MS-DOS アプリケーションをサブシステムで実行するプロテクト・モードのシステムでは、このような例外処理メカニズムを次のように処理できる。例えば、CR.NE=1 で実行しているプロテクト・モードの OS が仮想マシン・サブシステムで MS-DOS プログラムを実行する場合を考える。MS-DOS プログラムは、仮想の割り込みテーブルを持つ仮想マシン上でセットアップされる。MS-DOS プログラムは仮想マシンで通常どおりに割り込み 16 をフックする。数値例外はカーネルのリング 0 にあるリアル INT 16 によりカーネルにトラップする。

カーネルの INT 16 ハンドラは、正しい MS-DOS 仮想マシンを探し出して、その仮想マシン・モニタに割り込みを反映させる。仮想マシン・モニタは、仮想の割り込みテーブルに入っているアドレスを使用してジャンプし、アプリケーションの数値例外ハンドラに到達する。

D.3.6.5. オペレーティング・システムがストリーミング SIMD 拡張命令をサポートしている場合の考慮事項

インテル® Pentium® III プロセッサで導入されたストリーミング SIMD 拡張命令をサポートするオペレーティング・システムは、FXSAVE 命令と FXRSTOR 命令を使用して、既存の浮動小数点ステートと新しい SIMD 浮動小数点命令レジスタ・ステートのセーブとリストアを実行する。このようなオペレーティング・システムでは、次の事項を考慮に入れる必要がある。

1. **ステート・セーブ領域の拡大**：FNSAVE/FRSTOR 命令は、16 ビット・モードでは 94 バイトのメモリ領域を操作し、32 ビット・モードでは 108 バイトのメモリ領域を操作する。FXSAVE/FXRSTOR 命令は、512 バイトのメモリ領域を操作する。
2. **アライメントの必要条件**：FXSAVE/FXRSTOR 命令は、処理するメモリ領域が 16 バイト・アライメントであることを要求する。メモリ領域のアライメントが合っていない場合に発生する例外については、『IA-32 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、中巻 A』の第 3 章「命令セット・リファレンス A-M」の各命令の説明を参照のこと。
3. **従来のアプリケーション / ライブラリとの互換性の維持**：ストリーミング SIMD 拡張命令をサポートするためのオペレーティング・システムの変更点が、浮動小数点命令だけを扱う従来のアプリケーションやライブラリに影響を与えないようにする必要がある。FXSAVE/FXRSTOR 命令が操作するメモリ領域のレイアウトは、FNSAVE/FRSTOR 命令が操作するメモリ領域のレイアウトとは異なる。具体的には、x87 FPU タグワードのフォーマットと、メモリ領域内の各種のフィールドの長さが異なる。x87 FPU ステートを従来のアプリケーションに返すときは（例えば、浮動小数点例外を報告する場合など）、従来のアプリケーションが受け入れるフォーマットを使用しなければならない。
4. **命令セマンティクスの相違点**：FXSAVE 命令と FSAVE/FNSAVE 命令の間には、動作のセマンティクスに若干の相違点がある。FSAVE/FNSAVE 命令は、x87 FPU ステートをセーブした後に x87 FPU をクリアする。FXSAVE 命令は、x87 FPU/ ストリーミング SIMD 拡張命令ステートをセーブした後にそのステートをクリアしない。したがって、オペレーティング・システムが、浮動小数点ユニットを他のスレッドに渡す前に（例えば、スレッド・スイッチ時間中に）、FXSAVE 命令を使用して x87 FPU ステートをセーブする場合は、「汚れた」x87 FPU を他のアプリケーションに渡さないように注意する必要がある。

D.4. ネイティブ・モードのハンドラとの相違点

8087 の INT ピンは、マスクされていない例外が発生するとこの INT ピンがアサートする。ところが 8086 または 8088 には、それに接続するための専用の割り込み入力ピンも、x87 FPU エラー・アサート専用の割り込みベクタ番号もなかった。インテル 286 プロセッサとインテル 287 プロセッサの組み合わせにおいて初めて x87 FPU 例外に対するサポートが行われ、割り込みベクタ 16 が割り当てられた。

D.4.1. インテル® 286 プロセッサとインテル® 287 プロセッサ、Intel386™ プロセッサとインテル® 387 プロセッサの場合

インテル® 286 プロセッサとインテル® 287 プロセッサ、および Intel386™ プロセッサとインテル 387 プロセッサ / コプロセッサのペアでは、プロセッサと x87 FPU の ERROR# ピン同士を接続するよう推奨している。こうすると、マスクされていない x87 FPU 例外が発生した場合、x87 FPU が例外を記録し ERROR# ピンをアサートする。プロセッサは、命令ストリームにおける次の WAIT 命令または x87 FPU 命令（ただし、非同期型命令は除く）の直前で ERROR# ステータス・ラインがアクティブになったことを識別し、割り込みベクタ 16 のルーチンに分岐する。このようにして、エラーを起こした x87 FPU 命令の次の x87 FPU 命令より前に、x87 FPU 例外が処理される（非同期型命令の場合は x87 FPU 例外割り込みをトリガせずに実行されるが、例外はペンディング状態のままである）。

x87 FPU 例外処理専用の割り込み 16 を使用するのがネイティブ・モードである。これは最も単純な割り込み処理方法であり、インテルはこのネイティブ・モードを推奨している。

D.4.2. CR0.NE=1 の Intel486™ プロセッサ、インテル® Pentium® プロセッサ、インテル® Pentium® Pro プロセッサの場合

IA-32 アーキテクチャの第三世代では、それぞれの x87 FPU に対し、さらに機能拡張と高速化がなされた。また x87 FPU がプロセッサと同一チップに組み込まれたので、x87 FPU は統合システムの構成要素としてさらに高速に動作できるようになった。そのため、CR0 レジスタの NE ビットを 1 にセットすることにより、ネイティブ・モードでの x87 FPU 例外処理も完全に内部処理として実行できるようになった。

x87 FPU 命令実行中にマスクされていない例外が発生すると、x87 FPU は内部的に例外を記録し、次の WAIT 命令または x87 FPU 命令の直前で割り込み 16 で例外ハンドラをトリガする（ただし、非同期型命令は除く。D.4.1. 項「インテル® 286 プロセッサとインテル® 287 プロセッサ、Intel386™ プロセッサとインテル® 387 プロセッサの場合」を参照のこと）。

NE=1 の場合も、NE=0 の場合にアサートされるプログラム・フローと同じタイミングで FERR# 出力がアクティブになる。ただし、ネイティブな内部モードの場合は、FERR# から PIC への接続によってシステムが INTR を発生することはない。（システムのハードウェア上で、MS-DOS のサポートのために FERR# を IRQ13 をトリガするように接続しているが、実際にはネイティブ・モードでシステムを動作させている場合には、OS によってスレーブ PIC の IRQ13 を無効にしなければならない。）このようなシステム構成においては、D.2.1.3. 項「非同期型命令のウインドウ内の x87 FPU 割り込み」で説明したような、非同期型 x87 FPU 命令中に x87 FPU 例外を受け取るような問題は起こらない。

D.4.3. ネイティブ・モードでタスク間で x87 FPU を共有する場合の注意事項

MS-DOS* 互換モードの x87 FPU 例外ハンドラをタスク間で共有させるために D.3.6. 項「タスク間で x87 FPU を共有する場合の注意事項」で説明した内容は、ネイティブ・モードの場合にも当てはまる。ただし、ネイティブ・モードの場合はカーネル実行中に疑似的な浮動小数点例外割り込みが発生しないので、ネイティブ・モード専用のハンドラで守るべき規則の方が簡単である。

MS-DOS 互換モードでの例外ハンドラ・コードでは、DNA ハンドラで FNSAVE 命令を使用して x87 FPU コンテキストを切り替えるときに実際に問題が生じることがある。x87 FPU 例外がアクティブなときに FNSAVE 命令が FERR# を短時間トリガすると、x87 FPU 例外ハンドラが DNA ハンドラ内部で呼び出される。ネイティブ・モードの場合は、FNSAVE 命令であっても非同期型命令であっても割り込み 16 をトリガすることはない（すでに説明したように、NE ビットの設定に関係なく FERR# がアサートされるが、PIC 経由の割り込みを OS によって無効にする）。また、ごくまれにカーネルの実行中に浮動小数点例外割り込みが発生することがあるが、これは、即時方式の x87 FPU 例外割り込みが外部ハードウェアによる遅延でカーネルに切り替えられてから発生する場合である。ネイティブ・モードの場合は外部ハードウェアによる遅延がないので、この問題も発生しない。

したがって、ネイティブ・モードの場合、x87 FPU 例外ハンドラは x87 FPU 所有者がカーネルかどうかを調べる必要がなく、DNA ハンドラも最初に x87 FPU 所有者をカーネルに設定する必要がない。ただし、この操作を省略してもわずかなステップしか違わないので、MS-DOS 互換モードが広く使用されていることを考えると、常に MS-DOS 互換モードに必要なステップを入れておく習慣にするとよい。

インテル® Pentium® プロセッサでの特殊な DP（デュアル・プロセッシング）モード、および複数のインテル Pentium プロセッサ、P6 ファミリまたはインテル® Pentium® 4 プロセッサが組み込まれているシステム向けのより汎用性のあるインテル・マルチプロセッサ仕様においては、x87 FPU 例外処理はネイティブ・モードでのみサポートしていることに注意しなければならない。複数のプロセッサが組み込まれているシステムで MS-DOS 互換の FPU モードを使用することはお勧めできない。

E

SIMD 浮動小数点例外ハンドラを作成する際のガイドライン

付録 E
SIMD 浮動小数点例外ハンドラを作成する際のガイドライン

E

SIMD浮動小数点例外の説明の詳細は、11.5.節「SSE、SSE2、SSE3の例外」を参照のこと。

本章では、数値（SIMD浮動小数点）例外を発生させるSSE、SSE2、SSE3について検討し、このような例外を処理するための必要条件の概要を示す。ここでは、RSQRTSS、RSQRTPS、RCPSS、RCPPS など、浮動小数点例外を発生しない命令、 x 87 命令、一覧にない命令は対象としない。

どの命令が数値例外を発生させるか、またそれらの命令の一覧については、付録C「浮動小数点例外の要約」を参照のこと。非数値例外は、標準的な IA-32 命令の非数値例外と同じように処理される。

E.1. 浮動小数点例外処理の 2 つのオプション

SSE、SSE2、SSE3によって浮動小数点例外が発生した場合、x87 FPU浮動小数点例外の場合と同じように、プロセッサは次のいずれかの処置を実行する。

- 発生した例外がマスクされている（すなわち、MXCSRレジスタの対応するマスクビットが1にセットされている）場合は、デフォルトの結果が生成される。ほとんどの場合は、この処理で問題はない。例外が外部に表示されることはないが、MXCSRレジスタの対応する例外フラグがセットされるので、後でそれらのフラグを確認できる。ただし、パックドデータの演算の場合、MXCSRにセットされた例外フラグを見ても、サブオペランドのうちどれが例外イベントを発生させたのかはわからない。
- 発生した例外がマスクされていない（すなわち、MXCSRレジスタの対応するマスクビットが0にセットされている）場合は、SIMD浮動小数点例外（#XF、ベクタ19）によって、あらかじめユーザが登録した例外ハンドラがオペレーティング・システムにサポートされて起動される。この処理については、次のE.2. 節「ソフトウェアによる例外処理」で説明する。

E.2. ソフトウェアによる例外処理

割り込みベクタ 19によって起動される例外処理ルーチンは、通常はシステム・ソフトウェア（オペレーティング・システムのカーネル）の一部になっている。ただし、割り込みディスクリプタ・テーブル（IDT）の 1 つのエントリが、このベクタ用にあらかじめ設定されていなければならない（『IA-32 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、下巻』の第 5 章「割り込みと例外処理」を参照）。コンパイラによっては、特定のランタイム・ライブラリを使用して浮動小数点例外処理を支援するものがある。浮動小数点例外を発生させる可能性がある x87 FPU 浮動小数点演算を実行する場合は、例外処理ルーチンが、すべての浮動小数点例外をディスエーブルにしなければならない（例えば、FLDCW 命令を使ってローカル制御ワードをロードする）。あるいは、例外処理ルーチンが、再入可能として実装されていなければならない（x87 FPU 例外の処理フローの例については、付録 D「x87 FPU 例外ハンドラを作成する際のガイドライン」の例 D-1. を参照のこと）。再入可能として実装されていない場合は、例外処理ルーチンは、x87 FPU 例外のステータス・フラグをクリアするか、すべての x87 FPU 浮動小数点例外をマスクする必要がある。しかし、SIMD 浮動小数点例外の場合は、例外フラグがマスクされていないとき、MXCSR レジスタの例外フラグをクリアする必要はない（例外フラグをクリアしてもかまわない）。SIMD 浮動小数点例外は正確であり、直ちに発生する。このため、対応する例外がマスクされていないときに SIMD 浮動小数点例外ステータス・フラグがセットされても、例外が発生することはない。

この下位レベルの例外処理ルーチンが実行する一般的な処置は次のとおりである。

- 後で表示/印刷ができるように、例外カウンタをインクリメントする。
- 診断情報（例えば、MXCSR レジスタと XMM レジスタ）を表示/印刷する。
- これ以降の実行を中止する。または、例外ポインタを使用して例外を発生させない命令を構築し、実行する。
- 例外に関する情報を、上位レベルのユーザ例外ハンドラに渡されるデータ構造にストアする。

ほとんどの場合（SSE、SSE2、SSE3 の場合も含む）、下位レベルの浮動小数点例外ハンドラは、「プロローグ」、「本体」、および「エピローグ」の 3 つの部分で構成される。

プロローグ部では、優先順位の高いソースからの割り込みから保護しなければならない処理を実行する。通常は、レジスタの状態をセーブし、診断情報をプロセッサからメモリに転送する。この重要な処理が完了すると、プロローグ部は、再び割り込みを可能にして、優先順位の高い割り込みハンドラが例外ハンドラに割り込めるようにする。ただし、割り込みハンドラは割り込みゲートを介して呼び出されたものとする。これは、プロセッサが EFLAGS レジスタの割り込みイネーブル（IF）フラグをクリア

したという意味である。6.4.1. 項「割り込み / 例外処理プロシージャのコール操作とリターン操作」を参照のこと。

例外ハンドラの本体は、診断情報を検討し、アプリケーションに応じた応答を実行する。具体的には、アプリケーションの実行を停止する、メッセージを表示する、問題を解決して通常の実行を再開する、データ構造をセットアップする、上位レベルのユーザ例外ハンドラを呼び出し、例外ハンドラからのリターン時に実行を再開する、などの応答がある。E.4. 節「2 進浮動小数点計算に関する IEEE-754 規格と SIMD 浮動小数点例外」では、ユーザ例外ハンドラを呼び出す場合を考える。

最後に、エピローグ部では、プロローグ部とは基本的に反対の動作を行う。すなわち、プロセッサの状態をリストアして、通常の実行を再開できるようにする。

次のコード例は、一般的な例外ハンドラを示している。この例外ハンドラと、E.4.3. 項「SIMD 浮動小数点エミュレーションのコード例」に示した例 E-2. のコードを組み合わせるには、次の処理を実行する必要がある。まず、（ここでは詳しく示していない）例外ハンドラの本体が、セーブされたステートを他のルーチンに渡す。そのルーチンが、例外を発生させた命令のすべてのサブオペランドを検査する。次に、特定のサブオペランドが原因で、マスクされていない（イネーブルになっている）例外が発生した場合は、ユーザの浮動小数点例外ハンドラを起動する。それ以外の場合は、問題の命令をエミュレートする。

例 E-1. SIMD 浮動小数点例外ハンドラ

```
SIMD_FP_EXC_HANDLER PROC
;
;;; PROLOGUE
; SAVE REGISTERS THAT MIGHT BE USED BY THE EXCEPTION HANDLER
   PUSH EBP                          ; SAVE EBP
   PUSH EAX                          ; SAVE EAX
   ...
   MOV EBP, ESP                      ; SAVE ESP in EBP
   SUB ESP, 512                      ; ALLOCATE 512 BYTES
   AND ESP, 0fffffff0h               ; MAKE THE ADDRESS 16-BYTE ALIGNED
   FXSAVE [ESP]                      ; SAVE FP, MMX, AND SIMD FP STATE
   PUSH [EBP+EFLAGS_OFFSET]          ; COPY OLD EFLAGS TO STACK TOP
   POPFD                             ;RESTORE THE INTERRUPT ENABLE FLAG IF
                                     ;TO VALUE BEFORE SIMD FP EXCEPTION
;
;;; BODY
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
   LDMXCSR LOCAL_MXCSR               ; LOAD LOCAL MXCSR VALUE IF NEEDED
   ...
   ...
;
;;; EPILOGUE
   FXRSTOR [ESP]                     ; RESTORE MODIFIED STATE IMAGE
   MOV ESP, EBP                      ; DE-ALLOCATE STACK SPACE
   ...
   POP EAX                           ; RESTORE EAX
   POP EBP                           ; RESTORE EBP
   IRET                              ; RETURN TO INTERRUPTED CALCULATION
```

E.3. 例外の同期

SSE、SSE2、SSE3 は、他の SSE、SSE2、SSE3 整数命令、浮動小数点命令 /MMX® 命令と並行して実行できる。その際、x87 命令の場合と異なり、例外の同期に関する特別な注意は不要である。SSE、SSE2、SSE3 の浮動小数点例外は、次の浮動小数点命令が実行されるまで遅延することはなく、すぐに発生するからである。ただし、マスクされていない浮動小数点例外が生成されると、浮動小数点エミュレーションが必要になる場合もある。

E.4. 2 進浮動小数点計算に関する IEEE-754 規格と SIMD 浮動小数点例外

SSE、SSE2、SSE3 は、「2 進浮動小数点算術演算に関する IEEE 規格 754 に 100% 適合しており、この規格の必要条件をすべて満たしている（ゼロへのフラッシュ・モードまたはゼロへのデノーマル・モードがイネーブルになっていない場合）。ただし、SSE、SSE2、SSE3 を含むプログラミング環境は、許容されるハードウェアとソフトウェアの組み合わせとしてのみ、浮動小数点例外処理に関する IEEE 規格 754 の必要条件と推奨条件に適合する。この規格は、5 つの浮動小数点例外のうち任意の例外が発生したとき、ユーザはトラップを要求できなければならないと定めている（デノーマル例外は IA-32 で追加されたことに注意）。また、この規格は、例外ハンドラに渡される値（オペランドまたは結果）も指定している。

主な問題は、計算後型の例外（トラップ、すなわちオーバーフロー例外、アンダーフロー例外、または不正確例外）を発生させる SSE、SSE2、SSE3 の場合、x87 FPU 命令の場合とは異なり、プロセッサは IEEE 規格 754 が推奨する結果をユーザハンドラに提供しないことである。ユーザ・プログラムが、計算後型の例外を発生させた命令の結果を必要とする場合は、ソフトウェア側の責任で、フォルトを発生させた SSE、SSE2、または SSE3 をエミュレートし、その結果を求めなければならない。もう 1 つの問題は、IEEE 規格は、複数の浮動小数点例外が同時に発生した場合の処理について明確に指定していないことである。パックドデータの演算の場合は、各サブオペランドの演算によってセットされるフラグを OR（論理和）演算することで、MXCSR レジスタの例外フラグがセットされる。以下の各項では、これらの問題を解決する 1 つの方法について説明する。

E.4.1. 浮動小数点エミュレーション

すべてのオペレーティング・システムは、カーネルレベルの浮動小数点例外ハンドラを備えていなければならない（E.2. 節「ソフトウェアによる例外処理」にテンプレートを示した）。以下の説明では、ユーザモードの浮動小数点例外フィルタが（例えばC 関数のライブラリの一部として）SIMD 浮動小数点例外用に提供される場合を考える。ユーザ・プログラムは、マスクされていない例外を処理するために、この例外フィルタを起動できる。ユーザモードの浮動小数点例外フィルタ（ここには示していない）は、数値例外を発生させる SSE、SSE2、SSE3 をエミュレートできなければならない。また、この例外フィルタは、浮動小数点例外に対して、ユーザが提供した浮動小数点例外ハンドラを起動できなければならない。SSE、SSE2、SSE3 によって、マスクされていない浮動小数点例外が発生すると、下位レベルの浮動小数点例外ハンドラが呼び出される。この下位レベルのハンドラは、ユーザモードの浮動小数点例外フィルタを呼び出す。ハードウェアが結果を提供しないため、このフィルタ関数は、計算後型 / 計算前型のどちらの例外が発生した場合でも、例外を発生させた命令の元のオペランドを受け取る。例外フィルタは、受け取ったオペランドを最大 4 組のサブオペランドにアンパックし、一度に 1 組ずつエミュレーション関数に渡す（E.4.3. 項「SIMD 浮動小数点エミュレーションのコード例」の例 E-2. を参照）。エミュレーション関数は、サブオペランドを検査し、必要な計算を再実行する。

ここで、次の 2 つの場合が考えられる。

- この処理で、マスクされていない（イネーブルになっている）例外が発生した場合は、エミュレーション関数は、呼び出し元（フィルタ関数）に制御を戻し、適切な情報を渡す。フィルタは、問題のサブオペランドに対して、あらかじめ登録されているユーザの浮動小数点例外ハンドラを起動し、ユーザハンドラからのリターン時に結果を記録する（ユーザハンドラが実行を続けることを認めた場合）。
- マスクされていない（イネーブルになっている）例外が発生しなかった場合は、エミュレーション関数は、現在のサブオペランドの演算結果を求めて、呼び出し元に返す（この結果は IEEE 規格 754 に適合している必要がある）。フィルタ関数は、この結果（および新しいフラグの設定）を記録する。

次に、ユーザレベルのフィルタ関数は、サブオペランドの次の組に対してエミュレーション関数を呼び出す。この処理が完了すると、部分結果がパックされる（例外を発生させた命令が、パックド浮動小数点形式の結果を生成する場合。ほとんどの SSE、SSE2、SSE3 数値命令は、この条件に該当する）。ここで、例外フィルタは、下位レベルの例外ハンドラに制御を戻す。例外ハンドラは、割り込みからアプリケーションに制御を戻して、実行を再開させる。ただし、実行を正しく再開するためには、命令ポインタ（EIP）が、例外を発生させた命令の次の命令を指すように変更されていなければならない。

ユーザモードの浮動小数点例外フィルタが使用できない場合は、例外を発生させた命令をデコーディングし、その命令のオペランドを読み取り、マスクされていない浮動小数点例外に対応しない結果の構成要素について命令をエミュレートし、合成した結果を返すまでのすべての作業を、ユーザが提供した浮動小数点例外ハンドラが行わなければならない。

スカラ演算では、1 つのオペランドまたは 1 対のオペランドに対して、実際にエミュレーションを実行する必要がある。パックド演算では、4 つのオペランドすべてまたはサブ・オペランドすべてに対して、エミュレーションを実行する必要がある。これを行うには、次の手順を実行する必要がある。

- 例外を発生させた命令をデコーディングし、セーブされたコンテキストからオペランドを読み取る。
- 各サブオペランド（またはサブオペランドの各ペア）について、命令をエミュレートする。浮動小数点例外が発生しなかった場合は、部分結果をセーブする。マスクされた浮動小数点例外が発生した場合は、例外がマスクされているときの結果をエミュレーションによって求め、その結果をセーブし、適切なステータス・フラグをセットする。マスクされていない浮動小数点例外が発生した場合は、ユーザが提供した浮動小数点例外ハンドラが結果を生成し、適切なステータス・フラグをセットする。
- 4 つの部分結果を組み合わせて、コンテキストに書き込む。このコンテキストは、アプリケーションプログラムの実行が再開されるときにリストアされる。

マスクされていない浮動小数点例外の処理の制御フロー図を次に示す。

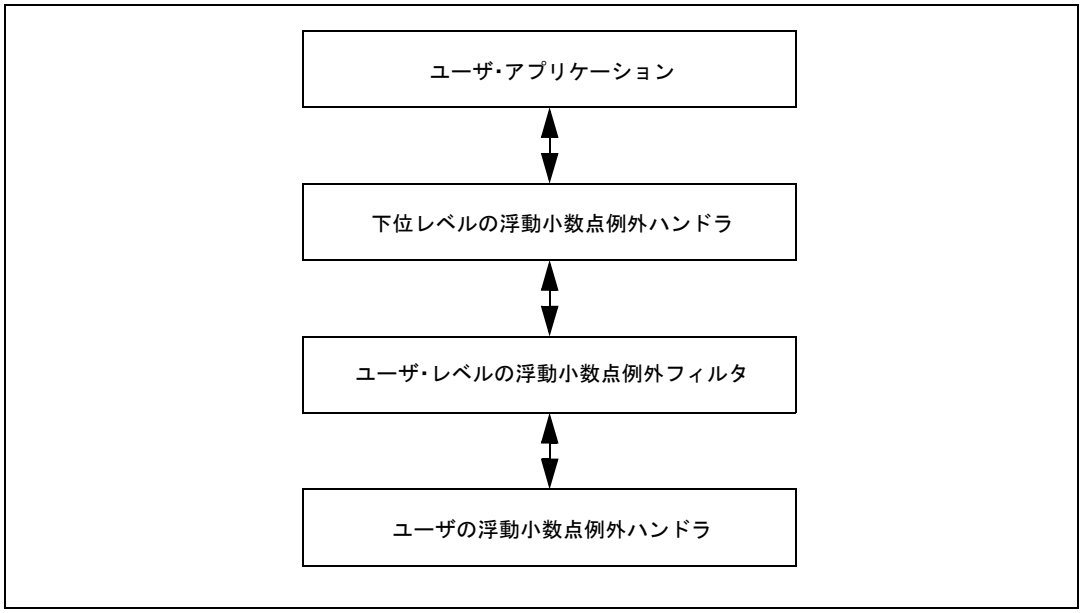


図 E-1. マスクされていない浮動小数点例外の処理の制御フロー

E.4.3. 項「SIMD 浮動小数点エミュレーションのコード例」の例 E-2. は、ユーザレベルの浮動小数点フィルタから、浮動小数点エミュレーションの部分だけを示している。これに関連する処理を理解するには、すべての SSE、SSE2、SSE3 の数値命令について、例外がイネーブルになっている場合（例外がマスクされていないときの結果）と例外がディスエーブルになっている場合（例外がマスクされているときの結果）の両方の場合に関して、例外に対する予想される応答を知っている必要がある。例外がマスクされているときの応答については、6.4. 節「割り込みと例外」を参照のこと。例外を発生させない NaN オペランドに対する応答については、4.8.3.4. 項「NaN（Not a Number)」を参照のこと。NaN オペランドに対する応答と、マスクされていない/マスクされている浮動小数点例外に対する応答については、次の項でも詳しく説明する。

E.4.2. 浮動小数点例外に対する SSE、SSE2、SSE3 の応答

この項では、SSE、SSE2、SSE3 がマスクされていない浮動小数点例外を発生させたときの、予想される応答について説明する。それと共に、例外がマスクされているときの応答について説明する（この応答は、マスクされていない浮動小数点例外を発生させる命令のエミュレーション・プロセスに必要である）。また、NaN オペランドに対する応答についても、4.8.3.4. 項「NaN（Not a Number)」より詳しく説明する。浮動小数点例外の優先順位については、『IA-32 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、下巻』の第 5 章「割り込みと例外処理」の「同時に発生した例外と割り込みの優先順位」を参照のこと。

E.4.2.1. 数値例外

数値（浮動小数点）例外条件には、無効操作（#I）、ゼロ除算（#Z）、デノーマル・オペランド（#D）、数値オーバーフロー（#O）、数値アンダーフロー（#U）、不正確結果（精度）（#P）の 6 つのクラスがある。#I、#Z、#D は、計算前型の例外（浮動小数点フォルト）であり、算術演算を実行する前に検出される。#O、#U、#P は、計算後型の例外（浮動小数点トラップ）である。

ユーザは、MXCSR レジスタのマスク / アンマスク・ビットの設定により、SSE、SSE2、SSE3 の浮動小数点例外の処理方法を制御できる。マスクされた例外は、同じ命令で発生したマスクされていない例外と組み合わされた場合にのみ、プロセッサまたはソフトウェアによって処理される。マスクされていない例外は、通常は下位レベルの例外ハンドラとユーザレベルのソフトウェアが協調して処理する。

E.4.2.2. SSE、SSE2、SSE3 数値命令で NaN オペランドまたは NaN 結果を含む演算の結果

以下の表（E-1. ～ E-10.）は、NaN の入力値（または NaN の結果を生じさせる NaN でない入力値）に対する、SSE、SSE2、SSE3 の応答を示している。

単精度 QNaN 不定値は 0xffc00000 であり、倍精度 QNaN 不定値は 0xfff8000000000000 であり、整数不定値は 0x80000000 である。この値は浮動小数点値ではないが、浮動小数点値から整数への変換命令の結果になりうる。

マスクされていない例外が発生した場合は、ハードウェアによって結果がユーザハンドラに提供されない。ユーザが登録した浮動小数点例外ハンドラが起動された場合は、そのハンドラが、例外を発生させた命令の結果を求める。この結果は、割り込みからのリターン後にアプリケーション・コードの実行が再開された場合に使用される。

表 E-1. ～表 E-12. では、指定したオペランドは、通常は無効例外を発生させる。ただし、「マスクされていないときの結果」の欄に「例外ではない」と記載されている場合は、マスクされていないときの結果とマスクされているときの結果は同じになる。

表 E-1. ADDPS、ADDSS、SUBPS、SUBSS、MULPS、MULSS、DIVPS、DIVSS、ADDPD、ADDSD、SUBPD、SUBSD、MULPD、MULSD、DIVPD、DIVSD、ADDSUBPS、ADDSUBPD、HADDPS、HADDPD、HSUBPS、HSUBPD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
SNaN1 op[1] SNaN2	SNaN1 | 00400000H または SNaN1 | 0008000000000000H[2]	なし
SNaN1 op QNaN2	SNaN1 | 00400000H または SNaN1 | 0008000000000000H[2]	なし
QNaN1 op SNaN2	QNaN1	なし
QNaN1 op QNaN2	QNaN1	QNaN1（例外ではない）
SNaN op 実数値	SNaN | 00400000H または SNaN1 | 0008000000000000H[2]	なし
実数値 op SNaN	SNaN | 00400000H または SNaN1 | 0008000000000000H[2]	なし
QNaN op 実数値	QNaN	QNaN（例外ではない）
実数値 op QNaN	QNaN	QNaN（例外ではない）
いずれのソース・オペランドも SNaN ではないにもかかわらず、#I が通知された（例えば、Inf - Inf、Inf * 0、Inf/Inf、0/0 の場合）	単精度 QNaN 不定値または 倍精度 QNaN 不定値	なし

注 1. 表 E-1. ～表 E-2. : op は、実行対象の演算を示す。

注 2. SNaN | 0x00400000 は単精度フォーマットのクワイエット型 NaN であり（SNaN が単精度の場合）、SNaN | 0008000000000000H は倍精度フォーマットのクワイエット型 NaN である（SNaN が倍精度の場合）。それぞれ、シグナル型 NaN が入力として与えられた場合に得られる。

注 3. クワイエット型 NaN だけを含む演算は、浮動小数点例外を発生させない。

表 E-2. CMPPS.EQ、CMPSS.EQ、CMPPS.ORD、CMPSS.ORD、CMPPD.EQ、CMPSD.EQ、CMPPD.ORD、CMPSD.ORD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
NaN op Opd2（任意の Opd2）	00000000H または 0000000000000000H[1]	00000000H または 0000000000000000H[1]（例外ではない）
Opd1 op NaN（任意の Opd1）	00000000H または 0000000000000000H[1]	00000000H または 0000000000000000H[1]（例外ではない）

注 1. 32 ビットの結果は単精度操作用であり、64 ビットの結果は倍精度操作用である。

表 E-3. CMPPS.NEQ、CMPSS.NEQ、CMPPS.UNORD、CMPSS.UNORD、CMPPD.NEQ、CMPSD.NEQ、CMPPD.UNORD、CMPSD.UNORD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
NaN op Opd2（任意の Opd2）	FFFFFFFFH または FFFFFFFFFFFFFFFFH[1]	FFFFFFFFH または FFFFFFFFFFFFFFFFH[1] （例外ではない）
Opd1 op NaN（任意の Opd1）	FFFFFFFFH または FFFFFFFFFFFFFFFFH[1]	FFFFFFFFH または FFFFFFFFFFFFFFFFH[1] （例外ではない）

注 1. 32 ビットの結果は単精度操作用であり、64 ビットの結果は倍精度操作用である。

表 E-4. CMPPS.LT、CMPSS.LT、CMPPS.LE、CMPSS.LE、CMPPD.LT、CMPSD.LT、CMPPD.LE, CMPSD.LE

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
NaN op Opd2（任意の Opd2）	00000000H または 0000000000000000H[1]	なし
Opd1 op NaN（任意の Opd1）	00000000H または 0000000000000000H[1]	なし

注 1. 32 ビットの結果は単精度操作用であり、64 ビットの結果は倍精度操作用である。

表 E-5. CMPPS.NLT、CMPSS.NLT、CMPSS.NLT、CMPSS.NLE、CMPPD.NLT、CMPSD.NLT, CMPPD.NLE、CMPSD.NLE

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
NaN op Opd2（任意の Opd2）	FFFFFFFFH または FFFFFFFFFFFFFFFFH[1]	なし
Opd1 op NaN（任意の Opd1）	FFFFFFFFH または FFFFFFFFFFFFFFFFH[1]	なし

注 1. 32 ビットの結果は単精度操作用であり、64 ビットの結果は倍精度操作用である。

表 E-6. COMISS、COMISD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
SNaN op Opd2（任意の Opd2）	OF,SF,AF=000 ZF,PF,CF=111	なし
Opd1 op SNaN（任意の Opd1）	OF,SF,AF=000 ZF,PF,CF=111	なし
QNaN op Opd2（任意の Opd2）	OF,SF,AF=000 ZF,PF,CF=111	なし
Opd1 op QNaN（任意の Opd1）	OF,SF,AF=000 ZF,PF,CF=111	なし

表 E-7. UCOMISS、UCOMISD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
SNaN op Opd2（任意の Opd2）	OF,SF,AF=000 ZF,PF,CF=111	なし
Opd1 op SNaN（任意の Opd1）	OF,SF,AF=000 ZF,PF,CF=111	なし
QNaN op Opd2 （任意の Opd2 ≠ SNaN）	OF,SF,AF=000 ZF,PF,CF=111	OF,SF,AF=000 ZF,PF,CF=111 （例外ではない）
Opd1 op QNaN （任意の Opd1 ≠ SNaN）	OF,SF,AF=000 ZF,PF,CF=111	OF,SF,AF=000 ZF,PF,CF=111 （例外ではない）

表 E-8. CVTPS2PI、CVTSS2SI、CVTTPS2PI、CVTTSS2SI、CVTPD2PI、CVTSD2SI、CVTTPD2PI, CVTTSD2SI、CVTPS2DQ、CVTTPS2DQ、CVTPD2DQ、CVTTPD2DQ

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
SNaN	80000000H または 8000000000000000[1]（整数不定値）	なし
QNaN	80000000H または 8000000000000000[1]（整数不定値）	なし

注 1. 32 ビットの結果は単精度操作用であり、64 ビットの結果は倍精度操作用である。

表 E-9. MAXPS、MAXSS、MINPS、MINSS、MAXPD、MAXSD、MINPD、MINSD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
Opd1 op NaN2（任意の Opd1）	NaN2	なし
NaN1 op Opd2（任意の Opd2）	Opd2	なし

注： SNaN オペランドと QNaN オペランドは無効操作例外を発生させる。

表 E-10. SQRTPS、SQRTSS、SQRTPD、SQRTSD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
QnaN	QNaN	QNaN（例外ではない）
SNaN	SNaN | 00400000H または SNaN | 0008000000000000H[1]	なし
ソース・オペランドが SNaN ではないにもかかわらず、#I が通知された （例えば、sqrt(-1.0) の場合）	単精度 QNaN 不定値または 倍精度 QNaN 不定値	なし

注 1. SNaN | 00400000H は単精度フォーマットのクワイエット型 NaN であり（SNaN が単精度の場合）、SNaN | 0008000000000000H は倍精度フォーマットのクワイエット型 NaN である（SNaN が倍精度の場合）。それぞれ、シグナル型 NaN が入力として与えられた場合に得られる。

表 E-11. CVTPS2PD, CVTSS2SD

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
QNaN	QNaN1[1]	QNaN1[1]（例外ではない）
SNaN	QNaN1[2]	なし

注 1. 倍精度出力 QNaN1 は、単精度入力 QNaN から次のように作成される。符号ビットの保存後、8 ビット指数部 FFH を 11 ビット指数部 7FFH に置き換え、0 を 29 ビット追加することで 24 ビット仮数部を 53 ビット仮数部に延長する。

注 2. 倍精度出力 QNaN1 は、単精度入力 SNaN から次のように作成される。符号ビットの保存後、8 ビット指数部 FFH を 11 ビット指数部 7FFH に置き換え、0 を 29 ビット追加することで 24 ビット仮数部を 53 ビット仮数部に延長する。シグナル型 NaN をクワイエット型 NaN に変換する際には、仮数部の 2 番目の上位ビットが 0 から 1 に変更される。

表 E-12. CVTPD2PS, CVTSD2SS

ソース・オペランド	マスクされているときの結果	マスクされていないときの結果
QNaN	QNaN1[1]	QNaN1[1]（例外ではない）
SNaN	QNaN1[2]	なし

注 1. 単精度出力 QNaN1 は、倍精度入力 QNaN から次のように作成される。符号ビットの保存後、11 ビット指数部 7FFH を 8 ビット指数部 FFH に置き換え、下位 29 ビットを削除することで 53 ビット仮数部を 24 ビット仮数部に切り捨てる。

注 2. 単精度出力 QNaN1 は、倍精度入力 SNaN から次のように作成される。符号ビットの保存後、11 ビット指数部 7FFH を 8 ビット指数部 FFH に置き換え、下位 29 ビットを削除すれば 53 ビット仮数部を 24 ビット仮数部に切り捨てる。シグナル型 NaN をクワイエット型 NaN に変換する際には、仮数部の 2 番目の上位ビットが 0 から 1 に変更される。

E.4.2.3. マスクされた数値例外とマスクされていない数値例外に対する条件コード、例外フラグ、応答

以下の表で、「マスクされているときの応答」とは、SSE、SSE2、またはSSE3数値命令がマスクされた例外を発生させたとき、プロセッサが提供する情報を示す。4 つの要素から成る入力オペランドの特定の要素が原因で、マスクされた例外が発生した場合は、ストリーミング SIMD 拡張数値命令の浮動小数点エミュレータが、これと同じ応答を生成する（浮動小数点例外が発生しない場合は、このエミュレータが、IEEE 規格 754 の規定にしたがって適切な答も生成する）。「マスクされていないときの応答」とは、SSE、SSE2、SSE3 のパックド・オペランドのうち、マスクされていない例外の原因になった要素について、エミュレータがユーザハンドラに提供する情報である。（ただし、COMISS、UCOMISS、COMISD、UCOMISD では、デスティネーションは EFLAGS レジスタである）。

以下の表では、操作の結果を 'res' で示す。実際の命令では、デスティネーションは第 1 ソース・オペランドと一致する（ただし、COMISS、UCOMISS、COMISD、UCOMISD では、デスティネーションは EFLAGS レジスタである）。

表 E-13. #I - 無効操作

命令	条件	マスクされているときの応答	マスクされていないときの応答と例外コード
ADDPS ADDPD ADDSS ADDSD HADDPS HADDPD	src1 または src2[1] = SNaN	NaN オペランドについては表 E-1. を参照、#IA=1	src1、src2 変更なし、#IA=1
ADDSUBPS（加算コンポーネント） ADDSUBPD（加算コンポーネント）	src1=+Inf、src2 = -Inf または src1=-Inf、src2 = +Inf	res[1] = QNaN 不定値、#IA=1	
SUBPS SUBPD SUBSS SUBSD HSUBPS HSUBPD	src1 または src2 = SNaN	NaN オペランドについては表 E-1. を参照、#IA=1	src1、src2 変更なし、#IA=1
ADDSUBPS（減算コンポーネント） ADDSUBPD（減算コンポーネント）	src1=+Inf、src2 = +Inf または src1=-Inf、src2 = -Inf	res = QNaN 不定値、#IA=1	
MULPS MULPD	src1 または src2 = SNaN	NaN オペランドについては表 E-1. を参照、#IA=1	src1、src2 変更なし、#IA=1
MULSS MULSD	src1=±Inf、src2 = ±0 または src1=±0、src2 = ±Inf	res = QNaN 不定値、#IA=1	
DIVPS DIVPD	src1 または src2 = SNaN	NaN オペランドについては表 E-1. を参照、#IA=1	src1、src2 変更なし、#IA=1
DIVSS DIVSD	src1=±Inf、src2 = ±Inf または src1=±0、src2 = ±0	res = QNaN 不定値、#IA=1	
SQRTPS SQRTPD SQRTSS SQRTSD	src = SNaN	NaN オペランドについては表 E-10. を参照、#IA=1	src 変更なし、#IA=1
	src < 0 (note that -0 < 0 is false)	res = QNaN 不定値、#IA=1	
MAXPS MAXSS MAXPD MAXSD	src1 = NaN または src2 = NaN	res = src2、#IA=1	src1、src2 変更なし、#IA=1

表 E-13. #I - 無効操作（続き）

命令	条件	マスクされているときの応答	マスクされていないときの応答と例外コード
MINPS MINSS MINPD MINSD	src1 = NaN または src2 = NaN	res = src2、#IA=1	src1、src2 変更なし、#IA=1
CMPPS.LT CMPPS.LE CMPPS.NLT CMPPS.NLE CMPSS.LT CMPSS.LE CMPSS.NLT CMPSS.NLE CMPPD.LT CMPPD.LE CMPPD.NLT CMPPD.NLE CMPSD.LT CMPSD.LE CMPSD.NLT CMPSD.NLE	src1 = NaN または src2 = NaN	NaN オペランドについては表 E-4. と表 E-5. を参照、#IA=1	src1、src2 変更なし、#IA=1
COMISS COMISD	src1 = NaN または src2 = NaN	NaN オペランドについては表 E-6. を参照	src1、src2、EFLAGS 変更なし、#IA=1
UCOMISS UCOMISD	src1 = SNaN または src2 = SNaN	NaN オペランドについては表 E-7. を参照	src1、src2、EFLAGS 変更なし、#IA=1
CVTPS2PI CVTSS2SI CVTPD2PI CVTSD2SI CVTPS2DQ CVTPD2DQ	src = NaN、±Inf、または $\lvert(src)_{rnd}\rvert > 7FFFFFFFH$ および $(src)_{rnd} \neq 80000000H$ rnd については、注 2 を参照	res = 整数不定値、#IA=1	src 変更なし、#IA=1
CVTTPS2PI CVTTSS2SI CVTTPD2PI CVTTSD2SI CVTTPS2DQ CVTTPD2DQ	src = NaN、±Inf、または $\lvert(src)_{rz}\rvert > 7FFFFFFFH$ および $(src)_{rz} \neq 80000000H$ rz については、注 2 を参照	res = 整数不定値、#IA=1	src 変更なし、#IA=1
CVTPS2PD CVTSS2SD	src = NAN	NaN オペランドについては表 E-11. を参照	src 変更なし、#IA=1
CVTPD2PS CVTSD2SS	src = NAN	NaN オペランドについては表 E-12. を参照	src 変更なし、#IA=1

注 1. 表 E-13. ～表 E-18.

- src は、単項演算の単一のソース・オペランドを示す。
- src1 と src2 は、それぞれ二項演算の 1 番目および 2 番目のソース・オペランドを示す。
- res は、演算の数値結果を示す。

注 2. rnd は、MXCSR レジスタで指定されたユーザ指定の丸めモードを示す。rz は、浮動小数点値を整数に丸めるときのゼロ側への丸め（切り捨て）モードを示す。詳細については、表 4-8. を参照のこと。

注 3. NaN のエンコーディングについては、表 4-3. を参照のこと。

表 E-14. #Z - ゼロ除算

命令	条件	マスクされているときの応答	マスクされていないときの応答と例外コード
DIVPS DIVSS DIVPD DIVPS	src1 = デノーマル[1] または src2 = デノーマル（および MXCSR の DAZ ビットは 0）	res = ±Inf #ZE=1	src1、src2 変更なし、 #ZE=1

表 E-15. #D - デノーマル・オペランド

命令	条件	マスクされているときの応答	マスクされていないときの応答と例外コード
ADDPS ADDPD ADDSUBPS ADDSUBPD HADDPS HADDPD SUBPS SUBPD HSUBPS HSUBPD MULPS MULPD DIVPS DIVPD SQRTPS SQRTPD MAXPS MAXPD MINPS MINPD CMPPS CMPPD ADDSS ADDSD SUBSS SUBSD MULSS MULSD DIVSS DIVSD SQRTSS SQRTSD MAXSS MAXSD MINSS MINSD CMPSS CMPSD COMISS COMISD UCOMISS UCOMISD CVTPS2PD CVTSS2SD CVTPD2PS CVTSD2SS	src1 = デノーマル[1] または src2 = デノーマル （および MXCSR の DAZ ビットは 0）	res = 有界の指数を使用して、デスティネーションの精度に合わせて丸められた結果（ただし、マスクされていない計算後型の例外が発生していない場合のみ）。	src1、src2 変更なし #DE=1 SQRT、CVTPS2PD、CVTSS2SD、CVTPD2PS、CVTSD2SS は src を 1 つだけとることに注意。

注 **1.** デノーマル数のエンコーディングについては、4.8.3.2. 項「ノーマル型有限数とデノーマル型有限数」を参照のこと。

表 E-16. #O - 数値オーバーフロー

命令	条件	マスクされているときの応答			マスクされていないときの応答と例外コード
ADDPS ADDSUBPS HADDPS SUBPS HSUBPS MULPS DIVPS ADDSS SUBSS MULSS DIVSS CVTPD2PS CVTSD2SS	丸められた結果 > 最大の単精度有限ノーマル値	**丸め**	**符号**	**結果とステータス・フラグ**	res =（境界のない指数を使用して計算され、デスティネーションの精度に合わせて丸められた結果）/ 2^{192} • #OE=1 • #PE=1（結果が不正確な場合）

		直近値への丸め	+
			-
				#OE=1, #PE=1
				res = + ∞
				res = −∞	

		- ∞方向	+
			-
				#OE=1, #PE=1
				res = 1.11…1 * 2^{127}
				res = −∞	

		+ ∞方向	+
			-
				#OE=1, #PE=1
				res = + ∞
				res = -1.11…1 * 2^{127}	

		ゼロ方向	+
			-
				#OE=1, #PE=1
				res = 1.11…1 * 2^{127}
				res = -1.11…1 * 2^{127}	

ADDPD ADDSUBPD HADDPD SUBPD HSUBPD MULPD DIVPD ADDSD SUBSD MULSD DIVSD	丸められた結果 > 最大の単精度有限ノーマル値	**丸め**	**符号**	**結果とステータス・フラグ**	res =（境界のない指数を使用して計算され、デスティネーションの精度に合わせて丸められた結果）/ 2^{1536} • #OE=1 • #PE=1（結果が不正確な場合）

		直近値への丸め	+
			-
				#OE = 1, #PE = 1
				res = + ∞
				res = −∞	

		- ∞方向	+
			-
				#OE = 1, #PE = 1
				res = 1.11…1 * 2^{1023}
				res = −∞	

		+ ∞方向	+
			-
				#OE = 1, #PE = 1
				res = + ∞
				res = -1.11…1 * 2^{1023}	

		ゼロ方向	+
			-
				#OE = 1, #PE = 1
				res = 1.11…1 * 2^{1023}
				res = -1.11…1 * 2^{1023}	

表 E-17. #U - 数値アンダーフロー

命令	条件	マスクされているときの応答	マスクされていないときの応答と例外コード
ADDPS ADDSUBPS HADDPS SUBPS HSUBPS MULPS DIVPS ADDSS SUBSS MULSS DIVSS CVTPD2PS CVTSD2SS	境界のない指数を使用して計算され、デスティネーションの精度に合わせて丸められた結果＜最小の単精度有限ノーマル値	res = ± 0、デノーマル、またはノーマル #UE=1 および #PE=1（ただし、結果が不正確な場合のみ）。	res =（境界のない指数を使用して計算され、デスティネーション・オペランドの精度に合わせて丸められた結果）* 2^{192} • #UE=1 • #PE=1（結果が不正確な場合）
ADDPD ADDSUBPD HADDPD SUBPD HSUBPD MULPD DIVPD ADDSD SUBSD MULSD DIVSD	境界のない指数を使用して計算され、デスティネーションの精度に合わせて丸められた結果＜最小の倍精度有限ノーマル値	res = ± 0、デノーマル、またはノーマル #UE=1 および #PE=1（ただし、結果が不正確な場合のみ）。	res =（境界のない指数を使用して計算され、デスティネーション・オペランドの精度に合わせて丸められた結果）* 2^{1536} • #UE=1 • #PE=1（結果が不正確な場合）

表 E-18. #P - 不正確結果（精度）

命令	条件	マスクされているときの応答	マスクされていないときの応答と例外コード
ADDPS ADDPD ADDSUBPS ADDSUBPD HADDPS HADDPD SUBPS SUBPD HSUBPS HSUBPD MULPS MULPD DIVPS DIVPD SQRTPS SQRTPD CVTDQ2PS CVTPI2PS CVTPS2PI CVTPS2DQ CVTPD2PI CVTPD2DQ CVTPD2PS CVTTPS2PI CVTTPD2PI CVTTPD2DQ CVTTPS2DQ ADDSS ADDSD SUBSS SUBSD MULSS MULSD DIVSS DIVSD SQRTSS SQRTSD CVTSI2SS CVTSS2SI CVTSD2SI CVTSD2SS CVTTSS2SI CVTTSD2SI	結果がデスティネーションのフォーマットで正確に表現できない。	res = 有界の指数を使用して計算され、デスティネーションの精度に合わせて丸められた結果。ただし、マスクされていないアンダーフロー条件またはオーバーフロー条件が発生していない場合のみ（この例外は、マスクされたアンダーフローまたはオーバーフローがある場合に発生する）。#PE=1	アンダーフロー / オーバーフロー条件が発生していない場合か、または対応する例外がマスクされている場合のみ。 • マスクされたオーバーフローの場合は、#OE をセットし、マスクされたオーバーフローについての上記の説明にしたがって結果を設定する。 • マスクされたアンダーフローの場合は、#UE をセットし、マスクされたアンダーフローについての上記の説明にしたがって結果を設定する。 アンダーフローでもオーバーフローでもない場合は、res = 有界の指数を使用して計算され、デスティネーションの精度に合わせて丸められた結果。 #PE=1

E.4.3. SIMD 浮動小数点エミュレーションのコード例

以下に示すコード例は、SSE、SSE2、SSE3 数値命令のユーザレベルの浮動小数点例外フィルタの一部である。このフィルタ関数は、下位レベルの例外ハンドラによって起動される（下位レベルの例外ハンドラは、マスクされていない浮動小数点例外が発生したとき、割り込みベクタ 19 によって起動される）。また、このフィルタ関数は、E.4.1. 項「浮動小数点エミュレーション」の説明にしたがって動作する。このコード例は、SSE の加算、減算、乗算、除算のエミュレーションのみを実行する。これを行うために、C コードと x87 FPU 操作が使用される。他の SSE、SSE2、SSE3 数値命令に対応する操作も同様にエミュレートできる。この例では、エミュレーション関数が、多くの入力パラメータを指定するデータ構造を指すポインタを受け取るものとする。入力パラメータには、例外を発生させた操作、1 対のサブオペランド（float 型のアンパックされた要素）、丸めモード（精度は常に単精度になる）、例外マスク（MXCSR レジスタ内と同じ相対ビット位置を持ち、符号なし整数のビット 0 から始まる）、およびゼロフラッシュ・インジケータとデノーマル・ゼロ・インジケータがある。

出力パラメータは、浮動小数点の結果（float 型）、例外の原因（以下で明確に定義されていない定数によって示される）、および例外ステータス・フラグである。対応する C の定義は次のとおりである。

```
typedef struct {
  unsigned int operation; // SSE or SSE2 operation: ADDPS, ADDSS, ...
  unsigned int operand1_uint32; // first operand value
  unsigned int operand2_uint32; // second operand value (if any)
  float result_fval; // result value (if any)
  unsigned int rounding_mode; // rounding mode
  unsigned int exc_masks; // exception masks, in the order P, U, O, Z, D, I
  unsigned int exception_cause; // exception cause
  unsigned int status_flag_inexact; // inexact status flag
  unsigned int status_flag_underflow; // underflow status flag
  unsigned int status_flag_overflow; // overflow status flag
  unsigned int status_flag_divide_by_zero; // divide by zero status flag
  unsigned int status_flag_denormal_operand; // denormal operand status flag
  unsigned int status_flag_invalid_operation; // invalid operation status flag
  unsigned int ftz; // flush-to-zero flag
unsigned int daz; // denormals-are-zeros flag
} EXC_ENV;
```

例に示した算術演算は、次のようにエミュレートされる。

1. デノーマル・ゼロ・モードが有効になっている（MXCSR の DAZ ビットが 1 にセットされている）場合は、すべてのデノーマル入力を同じ符号の 0 で置き換える（ただし、この処理は、デノーマル・フラグには影響を与えない）。

2. マスクされていないときの応答と例外コード x87 FPU 命令を使用して、例外をディスエーブルにし、ユーザが指定した元の丸めモードと単精度の設定で、この演算を実行する。例外条件が存在する場合は、これによって無効、デノーマル、またはゼロ除算例外が検出される。この結果を倍精度値としてメモリにストアする（この値の指数の範囲は、単精度計算の結果にとって「境界がない」ように見えるほど大きい）。
3. マスクされていないときの応答と例外コードマスクされていない例外が検出されなかった場合は、結果が単精度フォーマットで表現できる最小のノーマル型数より小さい数（極小数）か、単精度フォーマットで表現できる最大のノーマル型数より大きい数（極大数）かを判定する。マスクされていないオーバーフローまたはアンダーフロー例外が発生した場合は、IEEE-754 規格の指定にしたがって、スケーリングされた結果を計算する。この結果は、ユーザ例外ハンドラに渡される。
4. マスクされていないときの応答と例外コード上の手順で例外が発生しなかった場合は、「有界の」指数を含む結果を計算する。結果が極小数である場合は、デノーマライズ処理を行う必要がある（仮数を右にシフトして、指数をインクリメントし、単精度浮動小数点値の [-126,+127] の許容範囲内の値にする）。

   ステップ 2 で得られた結果は、二重丸め誤差を含む可能性があるため、使用できない（この結果は、ステップ 2 で 24 ビットに丸められ、デノーマライズ処理でもう一度丸められている可能性がある）。この問題を解決するには、結果を倍精度値として計算し、単精度フォーマットでメモリにストアすればよい。

   最初に仮数内 53 ビットに丸め、次に 24 ビットに丸めれば、二重丸め誤差は発生しない（二重丸め誤差が発生した場合について厳密に規定するプロパティもあるが、基本的な算術演算の場合、原則として、限りなく正確な結果を 2p + 1 ビットに丸め、さらに p ビットに丸めた場合、得られる結果は、直接 p ビットに丸めた場合と同じ値になる。つまり、二重丸め誤差は発生しない）。
5. マスクされていないときの応答と例外コード結果が不正確であり、不正確例外がマスクされていない場合は、計算された結果がユーザの浮動小数点例外ハンドラに転送される。
6. 結果が極小数である場合は、ゼロへのフラッシュの場合が処理される。
7. エミュレーション関数は、例外を発生させる必要がある場合は、フィルタ関数に RAISE_EXCEPTION を返す（exception_cause フィールドに原因が示される）。例外を発生させる必要がない場合は、DO_NOT_RAISE_EXCEPTION を返す。RAISE_EXCEPTION を返した場合は、フィルタ関数によって呼び出されたユーザ例外ハンドラが結果を生成する。DO_NOT_ RAISE_EXCEPTION を返した場合は、エミュレーション関数が結果を生成する。フィルタ関数は、すべての部分結果を集めて、スカラ値またはパックド値の結果を作成する。この結果は、実行が再開される場合に使用される。

例 E-2. SIMD 浮動小数点エミュレーション

```
// masks for individual status word bits
#define PRECISION_MASK 0x20
#define UNDERFLOW_MASK 0x10
#define OVERFLOW_MASK 0x08
#define ZERODIVIDE_MASK 0x04
#define DENORMAL_MASK 0x02
#define INVALID_MASK 0x01

// 32-bit constants
static unsigned ZEROF_ARRAY[] = {0x00000000};
#define  ZEROF *(float *) ZEROF_ARRAY
   // +0.0
static unsigned NZEROF_ARRAY[] = {0x80000000};
#define  NZEROF *(float *) NZEROF_ARRAY
   // -0.0
static unsigned POSINFF_ARRAY[] = {0x7f800000};
#define POSINFF *(float *)POSINFF_ARRAY
   // +Inf
static unsigned NEGINFF_ARRAY[] = {0xff800000};
#define NEGINFF *(float *)NEGINFF_ARRAY
   // -Inf

// 64-bit constants
static unsigned MIN_SINGLE_NORMAL_ARRAY [] = {0x00000000, 0x38100000};
#define MIN_SINGLE_NORMAL *(double *)MIN_SINGLE_NORMAL_ARRAY
   // +1.0 * 2^-126
static unsigned MAX_SINGLE_NORMAL_ARRAY [] = {0x70000000, 0x47effffff};
#define MAX_SINGLE_NORMAL *(double *)MAX_SINGLE_NORMAL_ARRAY
   // +1.1...1*2^127
static unsigned TWO_TO_192_ARRAY[] = {0x00000000, 0x4bf00000};
#define TWO_TO_192 *(double *)TWO_TO_192_ARRAY
   // +1.0 * 2^192
static unsigned TWO_TO_M192_ARRAY[] = {0x00000000, 0x33f00000};
#define TWO_TO_M192 *(double *)TWO_TO_M192_ARRAY
   // +1.0 * 2^-192

// auxiliary functions
static int isnanf (unsigned int); // returns 1 if f is a NaN, and 0 otherwise
static float quietf (unsigned int); // converts a signaling NaN to a quiet NaN, and
                        // leaves a quiet NaN unchanged
static float check_for_daz (unsigned int); // converts denormals to zeroes of the same sign;
                        // does not affect any status flags

// emulation of SSE and SSE2 instructions using
// C code and x87 FPU instructions

unsigned int
simd_fp_emulate (EXC_ENV *exc_env)

{

  float opd1; // first operand of the add, subtract, multiply, or divide
  float opd2; // second operand of the add, subtract, multiply, or divide
  float res; // result of the add, subtract, multiply, or divide
  double dbl_res24; // result with 24-bit significand, but "unbounded" exponent
     // (needed to check tininess, to provide a scaled result to
     // an underflow/overflow trap handler, and in flush-to-zero mode)
  double dbl_res;  // result in double precision format (needed to avoid a
```

```
   // double rounding error when denormalizing)
unsigned int result_tiny;
unsigned int result_huge;
unsigned short int sw; // 16 bits
unsigned short int cw; // 16 bits


// have to check first for faults (V, D, Z), and then for traps (O, U, I)

// initialize x87 FPU (floating-point exceptions are masked)
_asm {
  fninit;
}

result_tiny = 0;
result_huge = 0;

switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
  case SUBPS:
  case SUBSS:
  case MULPS:
  case MULSS:
  case DIVPS:
  case DIVSS:

    uiopd1 = exc_env->operand1_uint32; // copy as unsigned int
          // do not copy as float to avoid conversion of SNaN to QNaN by compiled code
    uiopd2 = exc_env->operand2_uint32;
          // do not copy as float to avoid conversion of SNaN to QNaN by compiled code
    uiopd1 = check_for_daz (uiopd1); // operand1 = +0.0 * operand1 if it is denormal
          // and DAZ=1
    uiopd2 = check_for_daz (uiopd2); // operand2 = +0.0 * operand2 if it is denormal
          // and DAZ=1

    // execute the operation and check whether the invalid, denormal, or
    // divide by zero flags are set and the respective exceptions enabled

    // set control word with rounding mode set to exc_env->rounding_mode,
    // single precision, and all exceptions disabled
    switch (exc_env->rounding_mode) {
      case ROUND_TO_NEAREST:
        cw = 0x003f; // round to nearest, single precision, exceptions masked
        break;
      case ROUND_DOWN:
        cw = 0x043f; // round down, single precision, exceptions masked
        break;
      case ROUND_UP:
        cw = 0x083f; // round up, single precision, exceptions masked
        break;
      case ROUND_TO_ZERO:
        cw = 0x0c3f; // round to zero, single precision, exceptions masked
        break;
      default:
        ;
    }
    __asm {
      fldcw WORD PTR cw;
```

```
}

// compute result and round to the destination precision, with
// "unbounded" exponent (first IEEE rounding)
switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
    // perform the addition
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set the denormal or invalid status flags
      faddp st(1), st(0); // may set the inexact or invalid status flags
      // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;

  case SUBPS:
  case SUBSS:
    // perform the subtraction
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set the denormal or invalid status flags
      fsubp st(1), st(0); // may set the inexact or invalid status flags
      // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;

  case MULPS:
  case MULSS:
    // perform the multiplication
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set the denormal or invalid status flags
      fmulp st(1), st(0); // may set the inexact or invalid status flags
      // store result
      fstp  QWORD PTR dbl_res24; // exact
    }
    break;

  case DIVPS:
  case DIVSS:
    // perform the division
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set the denormal or invalid status flags
      fdivp st(1), st(0); // may set the inexact, divide by zero, or
                          // invalid status flags
      // store result
      fstp  QWORD PTR dbl_res24; // exact
```

```
        }
        break;

      default:
        ; // will never occur

    }

    // read status word
    __asm {
      fstsw WORD PTR sw;
}

    if (sw & ZERODIVIDE_MASK)
    sw = sw & ~DENORMAL_MASK; // clear D flag for (denormal / 0)

    // if invalid flag is set, and invalid exceptions are enabled, take trap
    if (!(exc_env->exc_masks & INVALID_MASK) && (sw & INVALID_MASK)) {
      exc_env->status_flag_invalid_operation = 1;
      exc_env->exception_cause = INVALID_OPERATION;
      return (RAISE_EXCEPTION);
    }

    // checking for NaN operands has priority over denormal exceptions; also fix for the
    // differences in treating two NaN inputs between the SSE and SSE2
    // instructions and other IA-32 instructions
    if (isnanf (uiopd1) || isnanf (uiopd2)) {

      if (isnanf (uiopd1) && isnanf (uiopd2))
          exc_env->result_fval = quietf (uiopd1);
      else
          exc_env->result_fval = (float)dbl_res24; // exact

      if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
      return (DO_NOT_RAISE_EXCEPTION);
    }

    // if denormal flag is set, and denormal exceptions are enabled, take trap
    if (!(exc_env->exc_masks & DENORMAL_MASK) && (sw & DENORMAL_MASK)) {
      exc_env->status_flag_denormal_operand = 1;
      exc_env->exception_cause = DENORMAL_OPERAND;
      return (RAISE_EXCEPTION);
    }

    // if divide by zero flag is set, and divide by zero exceptions are
    // enabled, take trap (for divide only)
    if (!(exc_env->exc_masks & ZERODIVIDE_MASK) && (sw & ZERODIVIDE_MASK)) {
      exc_env->status_flag_divide_by_zero = 1;
      exc_env->exception_cause = DIVIDE_BY_ZERO;
      return (RAISE_EXCEPTION);
    }

    // done if the result is a NaN (QNaN Indefinite)
    res = (float)dbl_res24;
    if (isnanf (res)) {
      exc_env->result_fval = res; // exact
      exc_env->status_flag_invalid_operation = 1;
      return (DO_NOT_RAISE_EXCEPTION);
    }
```

```
// dbl_res24 is not a NaN at this point

if (sw & DENORMAL_MASK) exc_env->status_flag_denormal_operand = 1;

// Note: (dbl_res24 == 0.0 && sw & PRECISION_MASK) cannot occur
if (-MIN_SINGLE_NORMAL < dbl_res24 && dbl_res24 < 0.0 ||
     0.0 < dbl_res24 && dbl_res24 < MIN_SINGLE_NORMAL) {
  result_tiny = 1;
}

// check if the result is huge
if (NEGINFF < dbl_res24 && dbl_res24 < -MAX_SINGLE_NORMAL ||
    MAX_SINGLE_NORMAL < dbl_res24 && dbl_res24 < POSINFF) {
  result_huge = 1;
}

// at this point, there are no enabled I, D, or Z exceptions; the instr.
// might lead to an enabled underflow, enabled underflow and inexact,
// enabled overflow, enabled overflow and inexact, enabled inexact, or
// none of these; if there are no U or O enabled exceptions, re-execute
// the instruction using IA-32 double precision format, and the
// user's rounding mode; exceptions must have been disabled before calling
// this function; an inexact exception may be reported on the 53-bit
// fsubp, fmulp, or on both the 53-bit and 24-bit conversions, while an
// overflow or underflow (with traps disabled) may be reported on the
// conversion from dbl_res to res

// check whether there is an underflow, overflow, or inexact trap to be
// taken

// if the underflow traps are enabled and the result is tiny, take
// underflow trap
if (!(exc_env->exc_masks & UNDERFLOW_MASK) && result_tiny) {
  dbl_res24 = TWO_TO_192 * dbl_res24; // exact
  exc_env->status_flag_underflow = 1;
  exc_env->exception_cause = UNDERFLOW;
  exc_env->result_fval = (float)dbl_res24; // exact
  if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
  return (RAISE_EXCEPTION);
}

// if overflow traps are enabled and the result is huge, take
// overflow trap
if (!(exc_env->exc_masks & OVERFLOW_MASK) &&  result_huge) {
  dbl_res24 = TWO_TO_M192 * dbl_res24; // exact
  exc_env->status_flag_overflow = 1;
  exc_env->exception_cause = OVERFLOW;
  exc_env->result_fval = (float)dbl_res24; // exact
  if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
  return (RAISE_EXCEPTION);
}

// set control word with rounding mode set to exc_env->rounding_mode,
// double precision, and all exceptions disabled
cw = cw | 0x0200; // set precision to double
__asm {
  fldcw WORD PTR cw;
}
```

```
switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
    // perform the addition
    __asm {
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal status flag
      fld DWORD PTR uiopd2; // may set the denormal status flag
      faddp st(1), st(0); // rounded to 53 bits, may set the inexact
                          // status flag
      // store result
      fstp  QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

  case SUBPS:
  case SUBSS:
    // perform the subtraction
    __asm {
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal status flag
      fld DWORD PTR uiopd2; // may set the denormal status flag
      fsubp st(1), st(0); // rounded to 53 bits, may set the inexact
                          //  status flag
      // store result
      fstp  QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

  case MULPS:
  case MULSS:
    // perform the multiplication
    __asm {
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal status flag
      fld DWORD PTR uiopd2; // may set the denormal status flag
      fmulp st(1), st(0); // rounded to 53 bits, exact
      // store result
      fstp  QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

  case DIVPS:
  case DIVSS:
    // perform the division
    __asm {
      // load input operands
      fld DWORD PTR uiopd1; // may set the denormal status flag
      fld DWORD PTR uiopd2; // may set the denormal status flag
      fdivp st(1), st(0); // rounded to 53 bits, may set the inexact
                          // status flag
      // store result
      fstp  QWORD PTR dbl_res; // exact, will not set any flag
    }
    break;

  default:
    ; // will never occur
```

```
}

// calculate result for the case an inexact trap has to be taken, or
// when no trap occurs (second IEEE rounding)
res = (float)dbl_res;
    // may set P, U or O; may also involve denormalizing the result

// read status word
__asm {
  fstsw WORD PTR sw;
}

// if inexact traps are enabled and result is inexact, take inexact trap
if (!(exc_env->exc_masks & PRECISION_MASK) &&
    ((sw & PRECISION_MASK) || (exc_env->ftz && result_tiny))) {
  exc_env->status_flag_inexact = 1;
  exc_env->exception_cause = INEXACT;
  if (result_tiny) {
    exc_env->status_flag_underflow = 1;

    // if ftz = 1 and result is tiny, result = 0.0
    // (no need to check for underflow traps disabled: result tiny and
    // underflow traps enabled would have caused taking an underflow
    // trap above)
    if (exc_env->ftz) {
      if (res > 0.0)
        res = ZEROF;
      else if (res < 0.0)
        res = NZEROF;
      // else leave res unchanged
    }
  }
  if (result_huge) exc_env->status_flag_overflow = 1;
  exc_env->result_fval = res;
  return (RAISE_EXCEPTION);
}

// if it got here, then there is no trap to be taken; the following must
// hold: ((the MXCSR U exceptions are disabled  or
//
// the MXCSR underflow exceptions are enabled and the underflow flag is
// clear and (the inexact flag is set or the inexact flag is clear and
// the 24-bit result with unbounded exponent is not tiny)))
// and (the MXCSR overflow traps are disabled or the overflow flag is
// clear) and (the MXCSR inexact traps are disabled or the inexact flag
// is clear)
//
// in this case, the result has to be delivered (the status flags are
// sticky, so they are all set correctly already)

// read status word to see if result is inexact
__asm {
  fstsw WORD PTR sw;
}

if (sw & UNDERFLOW_MASK) exc_env->status_flag_underflow = 1;
if (sw & OVERFLOW_MASK) exc_env->status_flag_overflow = 1;
if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;

// if ftz = 1, and result is tiny (underflow traps must be disabled),
```

```
    // result = 0.0
    if (exc_env->ftz && result_tiny) {
      if (res > 0.0)
        res = ZEROF;
      else if (res < 0.0)
        res = NZEROF;
      // else leave res unchanged

      exc_env->status_flag_inexact = 1;
      exc_env->status_flag_underflow = 1;
    }

    exc_env->result_fval = res;
    if (sw & ZERODIVIDE_MASK) exc_env->status_flag_divide_by_zero = 1;
    if (sw & DENORMAL_MASK) exc_env->status_flag_denormal= 1;
    if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
    return (DO_NOT_RAISE_EXCEPTION);

    break;

  case CMPPS:
  case CMPSS:

    ...

    break;

  case COMISS:
  case UCOMISS:

    ...

    break;

  case CVTPI2PS:
  case CVTSI2SS:

    ...

    break;

  case CVTPS2PI:
  case CVTSS2SI:
  case CVTTPS2PI:
  case CVTTSS2SI:

    ...

    break;

  case MAXPS:
  case MAXSS:
  case MINPS:
  case MINSS:

    ...

    break;

  case SQRTPS:
```

```
    case SQRTSS:
      ...
      break;
    case UNSPEC:
      ...
      break;
    default:
      ...
  }
}
```

索 引

索 引

記号・数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

あ

い

と

に

の

は

ひ

ふ

へ

ほ

ま

む

ろ

わ