インテル・アーキテクチャ
最適化マニュアル

資料番号 242816J-003
1997

目次

図目次

表目次

intel

1

インテル・アーキテクチャ 最適化マニュアルの概要

第 1 章
インテル・アーキテクチャ最適化マニュアルの概要

一般的に、インテル・アーキテクチャ (IA) プロセッサ向けアプリケーションの最適化は難しいことではない。インテル・アーキテクチャとすぐれた開発環境、ツールを理解しているかどうかにより、開発したアプリケーションが高速であるか、潜在的最高速度よりかなり低速であるかの差が生じる。もちろん、8086/8088、80286、Intel386™(DX または SX)、および Intel486™ プロセッサ向けに開発したアプリケーションは、修正も再コンパイルもすることなく、Pentium®、Pentium Pro、および Pentium II プロセッサで動作する。しかし、以降に説明するコード最適化手法を習得し、アーキテクチャに関する説明を理解すれば、アプリケーションをチューニングして最高のパフォーマンスを引き出すことができるようになる。

1.1 アプリケーションのチューニング

インテル・アーキテクチャ (IA) 全体を対象に最高の実行速度が得られるようアプリケーションをチューニングすることは、適切なツールがあれば比較的簡単である。チューニング・プロセスを開始するに当たっては、以下の知識および操作を心得ている必要がある。

- インテル・アーキテクチャに関する知識。第 2 章を参照。
- アプリケーションのパフォーマンスに影響を与える可能性のある重大なストールに関する知識。第 3、4、5 章を参照。
- 使用するコンパイラでどの程度の最適化が可能か、そしてそのコンパイラでより効率的なコードを生成させるためにはどのような設定にすればよいか、といった知識。
- アプリケーション内のボトルネックとそれによるパフォーマンスの低下に関する知識。本書で説明する VTune(パフォーマンス・モニタリング・ツール) を使用する。
- アプリケーションパフォーマンスの監視方法。VTune を使用する。

VTune リリース 2.0 (Visual Tuning Environment Release 2.0) は、アプリケーションの内部を視覚的にとらえ、そのどこからチューニングを開始したらよいかを理解する上で役に立つツールである。Pentium および Pentium Pro プロセッサでは、パフォーマンス・イベント・カウンタでユーザ・コードをモニタすることができる。これらのパフォーマンス・イベント・カウンタには、VTune を使用してアクセスすることができる。本書の各節では、チューニング用のパフォーマンス・カウンタについては補足情報を交えて記載してある。パフォーマンス・イベント・カウンタとそのプログラミングの詳細については、第 7 章に記載してある。VTune の注文案内については、1.4 節に記載してある。

1.2 本書について

本書では、読者がインテル・アーキテクチャ・ソフトウェアおよびアセンブリ言語プログラミングに精通していることを前提としている。

本書では、インテル MMX® テクノロジ搭載の IA プロセッサおよび従来の IA プロセッサに対するソフトウェア・プログラミングの最適化および留意点について説明する。さらに、各プロセッサのインプリメント上の相違点、およびファミリ全体を対象とした最適化の方法についても説明する。

本書は、本章 (第 1 章) を含めて以下の 7 つの章および 4 つの付録で構成されている。

第 1 章 - 本書 (インテル ・アーキテクチャ最適化マニュアル) の概要

第 2 章 - プロセッサ ・アーキテクチャおよびパイプラインの概要 -- IA プロセッサ・アーキテクチャおよびインテル MMX テクノロジの概要を示す。

第 3 章 - 整数ブレンデッド ・コードの最適化手法 -- 整数最適化規則の一覧を示し、高速整数アプリケーションを開発するための最適化手法について説明する。

第 4 章 - インテル MMX® テクノロジ ・コード開発のガイドライン -- インテル MMX テクノロジ最適化規則の一覧を示し、インテル MMX テクノロジに固有の最適化手法およびコーディング例について説明する。

第 5 章 - 浮動小数点アプリケーションの最適化手法 -- 浮動小数点コードに固有の規則の一覧、最適化手法、およびコーディング例を記載する。

第 6 章 - コンパイラ選択のための基準 -- アーキテクチャ上の主な相違を示し、ブレンデッド・コードの使用を推奨する。

第 7 章 - インテル ・アーキテクチャのパフォーマンス ・モニタリング機能の拡張 -- パフォーマンス・モニタリング・カウンタとそれらの機能について詳細に説明する。

付録 A - 整数ペアリング表 -- Pentium プロセッサの IA 整数命令とペアリング情報の一覧を示す。

付録 B - 浮動小数点ペアリング表 -- Pentium プロセッサの IA 浮動小数点命令とペアリング情報の一覧を示す。

付録 C - µop ブレークダウンの概要

付録 D - デコーダ用の Pentium® Pro プロセッサ命令 -- デコーダのスケジューリングを可能にするための IA マクロ命令と Pentium Pro プロセッサのデコード関連情報を示す。

1.3 参考文献

インテル・アーキテクチャおよび本書で言及している個々の手法の詳細については、以下の資料を参照されたい。

- 『Intel Architecture MMX™ Technology Programmer's Reference Manual』資料番号 243007
- 『インテル・アーキテクチャ MMX® テクノロジ・プログラマーズ・リファレンス・マニュアル』資料番号 243007J
- 『Pentium® Processor Family Developer's Manual (Vol.1,2,3)』資料番号 241428、241429、および 241430
- 『Pentium® ファミリー・デベロッパーズ・マニュアル（上巻、中巻、下巻)』資料番号 241428J、241429J、および 241430J
- 『Pentium® Pro Processor Family Developer's Manual (Vol.1,2,3)』資料番号 242690、242691、および 242692
- 『Pentium® Pro ファミリ・デベロッパーズ・マニュアル（上巻、中巻、下巻)』資料番号 242690J、242691J、および 242692J
- 『Developer' Insight』 http://developer.intel.com/sites/developer/
- 『開発者のためのホームページ』 http://www.intel.co.jp/jp/design/

1.4 VTune オーダー情報

最新のオーダー情報については、下記の VTune ホーム・ページを参照されたい。

http://www.intel.com/ial/vtune

米国内およびカナダから発注される場合は、1-800-253-3696 か、または 1-800-445-7899 の Programmer's Paradise に電話されたい。

その他の外国から発注される場合は、503-264-2203 に電話されたい。

intel

2

プロセッサ・アーキテクチャおよびパイプラインの概要

第 2 章
プロセッサ ・アーキテクチャおよびパイプラインの概要

本章では、インテル MMX テクノロジ搭載および従来の Pentium および Pentium Pro、Pentium II の各プロセッサについてパイプラインおよびアーキテクチャ上の機能の概要を示す。コードがプロセッサのパイプラインを通ってどのように流れるかを理解することにより、個々の最適化によってコードの速度が向上する理由をよりよく理解できる。本章では、最適化を最大限に利用する方法を説明する。

2.1 Pentium® プロセッサ

Pentium プロセッサは、高機能のスーパースケーラ型プロセッサであり、2 つの整数パイプラインとパイプライン化された浮動小数点ユニットを中核として構築されている。Pentium プロセッサでは、2 つの整数命令を同時に実行することができる。ソフトウェアが直接関与しない形で動的に行われる分岐予測により、分岐によるパイプラインのストールが最小限に抑えられる。

2.1.1 整数パイプライン

Pentium プロセッサには、図 2-1 に示すように、並行処理が可能な 2 つの整数パイプラインがある。メイン ・パイプ (U) には、プリフェッチ (PF)、デコード・ステージ 1 (D1)、デコード・ステージ 2 (D2)、実行 (E)、ライトバック (WB) という 5 つのステージがある。第 2 パイプ (V) は、実行できる命令にいくつかの制限がある点を除いてメイン ・パイプと同じである。これらの制限については、後の節で詳細に説明する。

Pentium プロセッサは、1 サイクルごとに命令を 2 つまで発行することができる。最初の 2 つの命令の実行中に、次の 2 つの命令がチェックされ、V パイプが使える (空いている) のであればそこへ投入される。つまり、U パイプに最初の 2 つの命令、V パイプに次の 2 つの命令という具合に振り分けられる。V パイプが使えない場合は、次の命令は U パイプに発行され、V パイプには命令は発行されない。

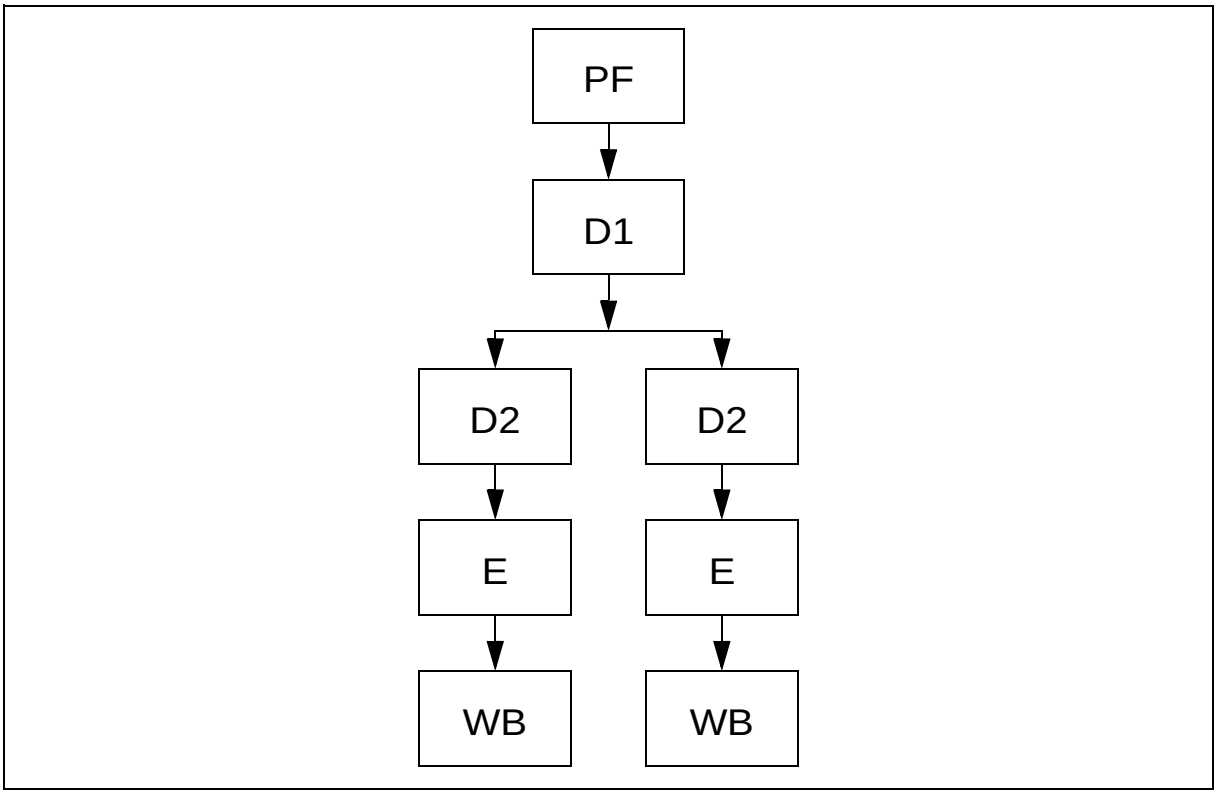


図 2-1. Pentium® プロセッサの整数パイプライン

命令が 2 つのパイプで並行的に実行される場合でも、命令の振舞いは、各命令を順次に実行した場合とまったく同じである。ストールが発生すると、どちらのパイプでも後続の命令はストールした命令を越えて先に進むことはできない。Pentium プロセッサのパイプラインでは、D2 ステージでメモリ・オペランドのアドレスが計算されるが、このステージではマルチウェイ加算を実行できるので、Intel486 プロセッサのパイプラインの場合とは異なり、1 クロックのインデックス・ペナルティはない。

スーパースケーラ型インプリメントでは、2 つの整数パイプラインの使用率を最大にするように命令ストリームをスケジュールすることが重要である。

2.1.2 キャッシュ

オンチップ・キャッシュ・サブシステムは、32 バイト・キャッシュ・ライン長 8K バイト 2 ウェイ・セット・アソシエイティブ・キャッシュ 2 つ (命令キャッシュとデータ・キャッシュ) で構成されている。64 ビット幅の外部データ・バス・インタフェースがある。これらのキャッシュは、ライトバック・メカニズムと LRU 置換アルゴリズムを採用している。データ・キャッシュは、4 バイト境界でインターリーブしている 8 つのバンクからなっている。参照先のバンクが異なっていれば、データ・キャッシュには両方のパイプから同時にアクセスすることができる。キャッシュ・ミスに対する最小遅延は 4 クロックである。

2.1.3 命令プリフェッチャ

命令プリフェッチャには、4 つの 32 バイト・バッファがある。プリフェッチ (PF) ステージでは、独立した 2 対のライン・サイズのプリフェッチ・バッファが分岐ターゲット・バッファと連動する。常に、一度に 1 つのプリフェッチ・バッファだけからプリフェッチを能動的に要求する。プリフェッチは、分岐命令がフェッチされるまで順次に要求される。分岐命令がフェッチされると、分岐が行われるかどうかを分岐ターゲット・バッファ (BTB) が予測する。分岐が行われないと予測された場合は、プリフェッチの要求がその後も続けられる。分岐が行われると予測された場合は、他方のプリフェッチ・バッファがイネーブルにされ、そのバッファが分岐が行われた場合と同様にプリフェッチを開始する。分岐予測が外れた場合は、命令パイプラインがフラッシュされ、最初からプリフェッチ操作がやり直される。プリフェッチャは、2 つのキャッシュ・ラインにまたがる 1 つの命令をペナルティなしでフェッチすることができる。命令キャッシュとデータ・キャッシュは独立しているので、命令のプリフェッチがキャッシュ参照のデータ・アクセスと衝突することはない。

2.1.4 分岐ターゲット・バッファ

Pentium プロセッサは、256 エントリの BTB (分岐ターゲット・バッファ) による動的分岐予測スキームを採用している。予測が正しい場合は、ペナルティなしで分岐命令が実行される。分岐が誤って予測された場合、条件付き分岐が U パイプで実行されていれば 3 サイクルのペナルティが生じ、V パイプで実行されていれば 4 サイクルのペナルティが生じる。コールおよび無条件ジャンプ命令が誤って予測された場合は、どちらのパイプでも 3 クロックのペナルティが生じる。

注記

分岐しない分岐は、誤って予測されるまで BTB 内には挿入されない。

2.1.5 ライト・バッファ

Pentium プロセッサには、2 つのライト・バッファがある。1 つの整数パイプラインに 1 つのバッファが対応しており、メモリへの連続書き込みのパフォーマンス向上を図っている。これらのライト・バッファは、1 クワッド・ワード幅 (64 ビット) であり、たとえば 2 つの命令パイプラインで 2 つの書き込みミスが同時に発生した場合に、1 クロックで同時にフィルされることができる。これらのバッファへの書き込みは、プロセッサ・コアで発生した順に外部バスに送出される。(キャッシュ・ミスの結果として) 読み取りがあっても、それ以前に発生していて両ライト・バッファに挿入されている書き込みの前後の順序は決して変わることはない。Pentium プロセッサは強力に書き込み順序の管理をしている。すなわち、書き込みは書き込みデータが発生した順序で行われる。

2.1.6 パイプライン化された浮動小数点ユニット

Pentium プロセッサは、整数パイプラインに 3 ステージの浮動小数点パイプを付加した、高性能の浮動小数点ユニットを備えている。浮動小数点命令は、E ステージまでパイプラインの中を進行する。その後、命令は、浮動小数点ステージ、すなわち X1 ステージ、X2 ステージ、WF ステージのそれぞれで最低 1 クロックを費やす。大部分の浮動小数点命令には 1 クロックを超える実行レイテンシがあるが、ほとんどは、パイプライン化によりパイプラインの別のステージにおける他の命令の実行によってレイテンシを覆い隠すことができる。さらに、FDIV などのレイテンシの大きい浮動小数点命令の実行中に整数命令を発行することができる。図 2-2 に、整数および浮動小数点の両パイプラインの構成を示す。

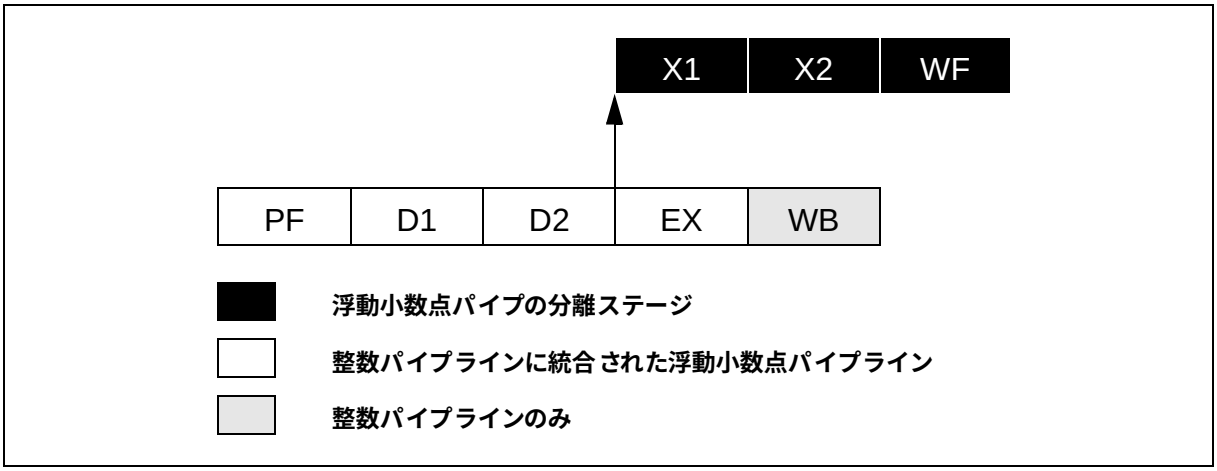


図 2-2. 整数パイプラインと浮動小数点パイプラインの統合

使用頻度の高い命令の大部分は、パイプラインに投入 され、各サイクルごとに新 しい 1 対 (ペア) の命令を受け付けられるようになっている。したがって、うまくコーディングすれば、ほぼ 1 サイクル当たり 2 命令のスループットを達成することができる (並行性については、人為的操作を加えないプログラムを想定)。FXCH 命令は、よく使用 される浮動小数点命令 と並行 して実行でき、パフォーマンスをほとんど低下 させることなく浮動小数点スタックを通常のレジスタ ・セットのように扱 うことができる。

2.2 Pentium® Pro プロセッサ

Pentium Pro プロセッサ ・ファミリでは、アウトオブオーダ実行と見込み実行をハードウェア ・レジスタ・リネーミングおよび分岐予測と融合させた動的実行方式を採用している。これらのプロセッサは、IA プロセッサのマクロ命令を単純なマイクロ ・オペレーション (µop) に分解するインオーダ発行パイプラインと、µop を実行するアウトオブオーダ型のスーパースケーラ ・プロセッサ ・コアを備えている。プロセッサのアウトオブオーダ ・コアは、整数、分岐、浮動小数点、メモリ・アクセスの各実行ユニットが接続されている複数のパイプラインで構成されている。同一のパイプラインに複数の異なる実行ユニットがクラスタ状に接続されている場合もある。たとえば、整数算術論理演算ユニットと浮動小数点用の各実行ユニット (加算器、乗算器、除算器) は同一のパイプラインを共有している。データ・キャッシュは、1 つのポートをロード専用、もう 1 つのポートをストア専用にしてインター リーブした、疑似デュアル ・ポート構成になっている。単純な演算 (整数 ALU、浮動小数点加算、浮動小数点乗算など) のほとんどはパイプライン化により、1 クロック・サイクル当たり 1 演算または 2 演算のスループットを得ることができるが、浮動小数点除算器はパイプライン化されない。レイテンシが長い演算は、レイテンシが短い演算と並行して処理することができる。

Pentium Pro プロセッサのパイプラインは、(1) インオーダ発行フロントエンド、(2) アウトオブオーダ ・コア、(3) インオーダ・リタイアメント・ユニットという 3 つの部分で構成されている。図 2-3 に、Pentium Pro プロセッサ・パイプライン全体の詳細を示す。

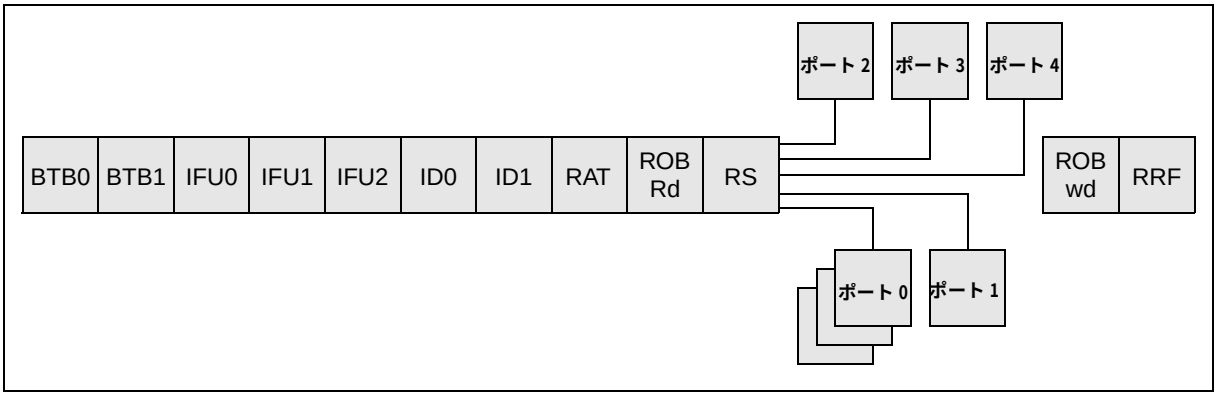


図 2-3. Pentium® Pro プロセッサのパイプライン

図 2-4 に、インオーダ発行フロントエンドの詳細図解を示す。

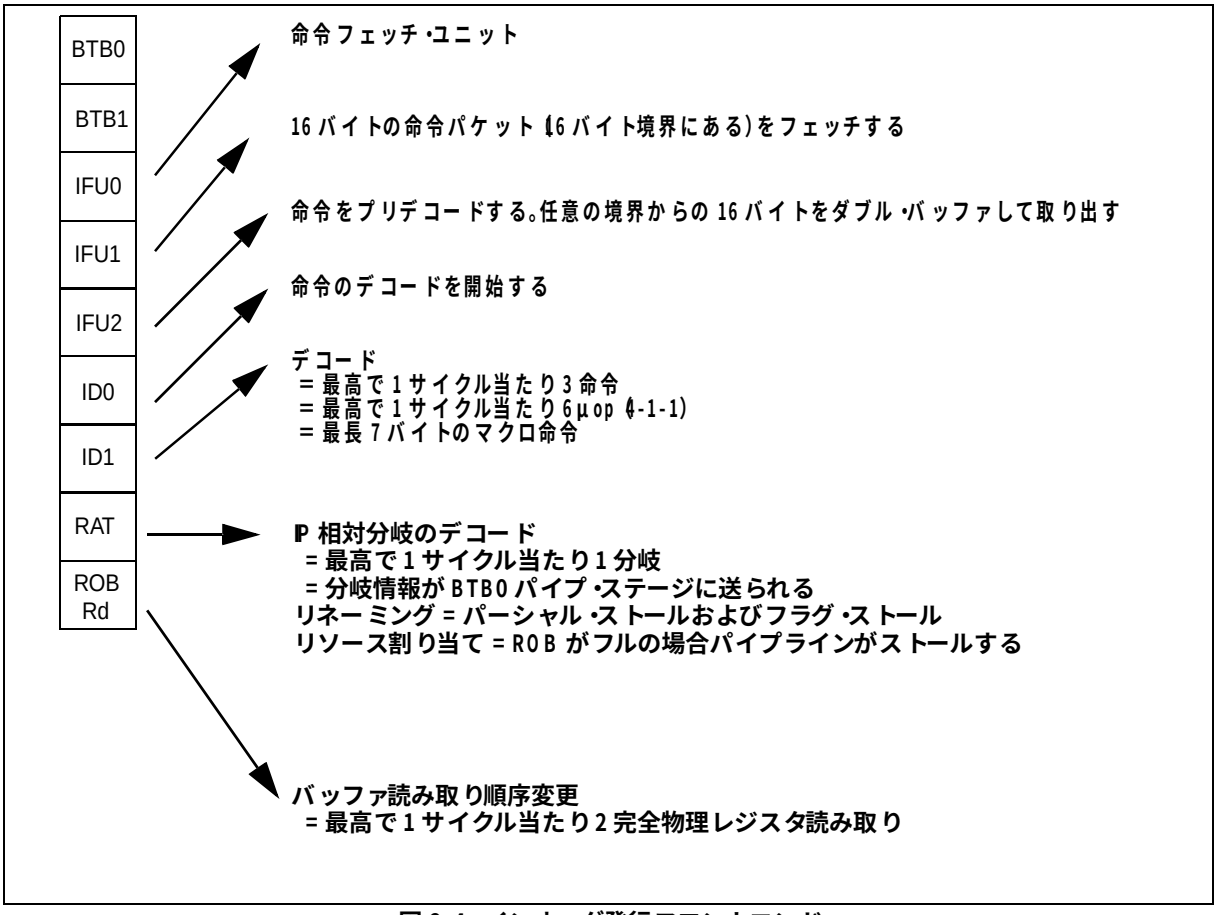


図 2-4. インオーダ発行フロントエンド

Pentium Pro プロセッサでは、プログラムされた命令の順序を変えて実行できるので、パフォーマンスのチューニングにおいては、十分な数の µop を実行可能にすることが何よりも大切である。インオーダ・フロントエンドから最高のパフォーマンスを得るには、正しい分岐予測と高速デコードが必須である。分岐予測と分岐ターゲット・バッファについては、3.2 節で詳細に説明する。デコードについては、以下に説明する。

ID1 パイプステージでは、1 クロック ・サイクルで最高 3 つのマクロ命令をデコードすることができる。ただし、命令が複雑であるか、または長さが 7 バイトを超える場合は、デコードできる命令の数はそれより少なくなる。

ID1 パイプステージのデコーダは、以下をデコードすることができる。

- 1 クロック ・サイクル当たり最高 3 マクロ命令
- 1 クロック ・サイクル当たり最高 6 µop
- 最長 7 バイトのマクロ命令

Pentium Pro プロセッサには、D1 パイプステージに 3 つのデコーダがある。第 1 のデコーダは、1 クロック ・サイクルごとに 4 µop 以下のマクロ命令を 1 つデコードすることができる。その他の 2 つのデコーダは、1 クロック ・サイクルごとに 1 µop のマクロ命令を 1 つデコードすることができる。5 µop 以上のマクロ命令の場合は、デコードに複数のサイクルを要する。アセンブリ言語によるプログラミングの場合は、命令を 4-1-1 の µop シーケンスにスケジュールすることにより、各クロック ・サイクル当たりにデコードできる命令数が増加する。一般的に、以下のことがいえる。

- レジスタ - レジスタ形式の単純な命令の µop の数は 1 つだけである。
- ロード命令の µop の数は 1 つだけである。
- ストア命令の µop の数は 2 つである。
- 単純な読み取り - 修正命令の µop の数は 2 つである。
- 単純なレジスタ - メモリ形式の命令の µop の数は 2 つまたは 3 つである。
- 読み取り - 修正 - 書き込み命令の µop の数は 4 つである。
- 複雑な命令の µop の数は一般に 5 つ以上であり、したがってデコードに複数のサイクルを要する。

付録 C に、インテル ・アーキテクチャ命令セットの各命令を構成する µop の数を示す。

µop は、デコードされると、インオーダ ・フロントエンドからアウトオブオーダ ・コアの最初のパイプステージである予約ステーション (RS) に発行される。µop は、RS 内でそれぞれのデータ ・オペランドが揃うまで待つ。すべてのデータ ・オペランドが揃うと、µop は RS から実行ユニットにディスパッチされる。µop がデータ ・レディ状態で (すなわち、すべてのデータが揃った状態で) RS に入り、そのときに使える実行ユニットがあると、µop はただちにその実行ユニットにディスパッチされる。この場合、µop は RS 内で余分なクロック ・サイクルを費やさない。すべての実行ユニットは、RS から出ているポートでクラスタ状に接続されている。

µop は、実行されるとリオーダ ・バッファ (ROB) にストアされ、リタイアを待つ。このパイプステージでは、すべてのデータ値がメモリにライトバックされ、すべての µop が一度に 3 つずつ順番にリタイアされる。図 2-5 に、アウトオブオーダ ・コアとインオーダ ・リタイアメントのパイプステージに関する詳細を示す。

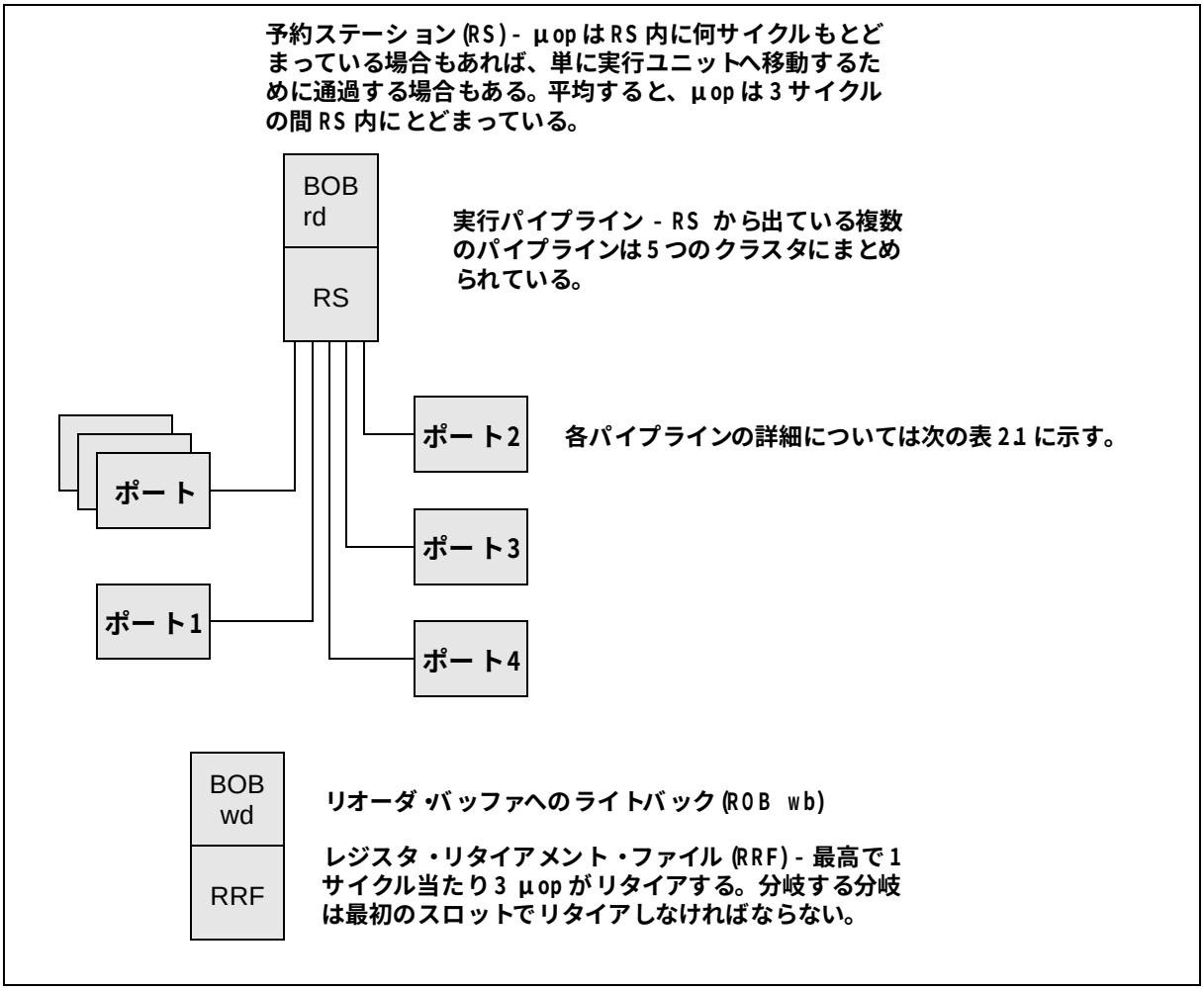


図2-5. アウトオブオーダ・コアとリタイアメント・パイプライン

表 2-1. Pentium® Pro プロセッサの実行ユニット

ポート	実行ユニット	レイテンシ / スループット
0	整数 ALU ユニット: LEA 命令 シフト命令 整数乗算命令 浮動小数点ユニット: FADD 命令 FMUL 命令 FDIV 命令	レイテンシ 1、スループット 1/ サイクル レイテンシ 1、スループット 1/ サイクル レイテンシ 1、スループット 1/ サイクル レイテンシ 4、スループット 1/ サイクル レイテンシ 3、スループット 1/ サイクル レイテンシ 5、スループット 1/2 サイクル [1,2] レイテンシ - 単精度 17 サイクル、倍精度 36 サイクル、拡張精度 56 サイクル、スループット - 非パイプライン化
1	整数 ALU ユニット	レイテンシ 1、スループット 1/ サイクル
2	ロード・ユニット	キャッシュ・ヒット時レイテンシ 3、スループット 1/ サイクル [3]
3	ストア・アドレス・ユニット	レイテンシ 3(該当しない)、スループット 1/ サイクル [3]
4	ストア・データ・ユニット	レイテンシ 1(該当しない)、スループット 1/ サイクル

注記:

1. FMUL ユニットは、最初の FMUL を受け付けた後、そのサイクルでは次の FMUL を受け付けることはできない。だからといって、FMUL が偶数番目のクロック・サイクルでしか実行できないというわけではない。FMUL は、2 クロック・サイクルごとに 1 つずつパイプライン化される。
2. ストアのレイテンシは、データフローの観点からはなんら重要ではない。レイテンシが問題になるのは、特定の µop がいつリタイアでき、完了できるかの判定に関してである。ストアの µop のレイテンシは、ロード・フォワーディングに関しても異なる。たとえば、特定アドレス、例としてストア・アドレス 100 とストア・データがクロック・サイクル 10 でディスパッチされる場合、同じアドレス 100 への (同じサイズと形の) ロードが同じクロック・サイクル 10 でディスパッチでき、ストールは生じない。
3. 同じアドレスからのロードと同じアドレスへのストアは、同一のクロック・サイクルでディスパッチすることができる。

2.2.1 キャッシュ

オンチップ・レベル 1(L1) キャッシュは、32 バイト・キャッシュ・ライン長の 8K バイト 4 ウェイ・セット・アソシエイティブ命令キャッシュ・ユニット 1 つと、8K バイト 2 ウェイ・セット・アソシエイティブ・データ・キャッシュ・ユニット 1 つで構成されている。L1 キャッシュ内のすべてのミスが最大のメモリ・アクセス遅延につながるわけではない。レベル 2(L2) キャッシュは、L1 キャッシュ・ミスによって生じる最大のメモリ・アクセス遅延をマスクする、すなわちその表面化を抑止する。L1 および L2 キャッシュ・ミスに対する最小遅延は 11 ～ 14 サイクルであるが、その実際の長さは DRAM ページがヒットまたはミスによって変わる。データ・キャッシュには、参照先のキャッシュ・バンクが異なれば、ロード命令とストア命令で同時にアクセスすることができる。

2.2.2 命令プリフェッチャ

命令プリフェッチャは、直線的処理を行うコードについては積極的にプリフェッチする。分岐でフォール・スルーしがちなループを構成しないでこのプリフェッチ機能を有効利用するようにコードを配置されたい。さらに、実行頻度の低いコードは、無駄にプリフェッチされないように、プロシージャの一番下またはプログラムの終わりに隔離して配置する。

命令のフェッチは常にアライメントが合った 16 バイト・ブロックに対して行われるということに注意されたい。Pentium Pro プロセッサは、16 バイトのアライメント境界から命令を読み取る。したがって、たとえば分岐先アドレス (ラベルのアドレス) が 14 モジュロ 16 に等しい場合は、最初のサイクルでは 2 つの有効な命令バイトだけがフェッチされる。残りの命令バイトは、それ以降のサイクルでフェッチされる。

2.2.3 分岐ターゲット・バッファ

512 エントリの分岐ターゲット・バッファ(BTB)には、以前に現れた分岐とそれらのターゲット、すなわち分岐先アドレスの履歴が保持されている。分岐がプリフェッチされると、BTB は分岐先アドレスを命令フェッチ・ユニット(IFU)に直接通知する。分岐が実行されると、BTB はその分岐先アドレスで更新される。以前に現れたことがある分岐は、分岐ターゲット・バッファ内の履歴に基づいて動的に予測される。分岐ターゲット・バッファの予測アルゴリズムでは、パターン・マッチングと各分岐先アドレスごとに最高 4 つの予測履歴ビットが使用される。たとえば、繰り返し回数 4 のループの予測はほぼ 100% 的中するはずである。以下のガイドラインに従うことにより、分岐予測の精度が向上する。

条件付き分岐(ループの場合を除く)については、最も多く実行される分岐が分岐命令の直後に来る(すなわち、フォール・スルーする)ようにプログラムする。

さらに、Pentium Pro プロセッサにはリターン・スタック・バッファ(RSB)があるが、このバッファは異なる位置から相次いで呼び出されるプロシージャのリターン・アドレスを正確に予測することができる。これにより、関数コールが介在するループをアンロールすることの利点が増大し、特定のプロシージャをインラインに配置する必要がなくなる。

Pentium Pro プロセッサでは、分岐に伴うペナルティのレベルは、浪費されるサイクル数によって次の 3 つに分けられている。

1. 分岐しない分岐はペナルティを生じない。これに該当するのが、BTB により分岐しないとして正しく予測された分岐、および BTB 内に存在せず、デフォルトにより分岐しないとして予測された前方分岐である。
2. BTB により分岐するとして正しく予測された分岐は、マイナーなペナルティ(約 1 サイクル)を生じる。つまり、命令のフェッチが 1 サイクル中断される。その間、プロセッサは以降の命令をデコードせず、結果として発行される μop の数が 4 を下回る可能性が高くなる。このマイナーなペナルティは、以前に現れたことがある(すなわち、BTB 内に存在する)無条件分岐にあてはまる。分岐するとして正しく予測された分岐に対するマイナーなペナルティは、命令フェッチで 1 サイクル浪費されることと、分岐後に命令が発行されないことである。
3. 誤って予測された分岐は大きなペナルティを生じる。誤って予測された分岐のペナルティは、最低 9 サイクル(インオーダ発行パイプラインの長さ)にわたって命令のフェッチが中断されることと、誤って予測された分岐命令がリタイアするまでの待ち時間が長くなることである。このペナルティは、実行状況に依存する。一般的に、分岐予測ミスのために浪費されるサイクル数は平均 10 ～ 15 であり、最高では 26 サイクルに及ぶこともある。

2.2.3.1 静的予測

BTB 内に存在しない分岐、すなわち静的予測メカニズムによって正しく予測された分岐は、約 5 または 6 サイクル(この点までのパイプラインの長さ)の小さなペナルティを生じる。このペナルティが生じるのは、以前にはまったく現れたことのない無条件直接分岐にあてはまる。

ループを閉じる分岐などディスプレースメントが負である条件付き分岐は、静的予測メカニズムによって分岐すると予測される。このような分岐は、初めて現れたときだけ小さいペナルティ(約6サイクル)を生じ、2回目以降の繰り返しでは負の分岐がBTBによって正しく予測され、ペナルティはマイナー(約1サイクル)になる。

BTB内には存在しないで、デコーダによって正しく予測された分岐に対する小さいペナルティでは、正しく予測されなかった分岐または予測なしの分岐に対する10～15サイクルのペナルティとは対照的に命令フェッチが約5サイクルにわたって中断される。

2.2.4 ライト・バッファ

Pentium Proプロセッサは、メモリへの各書き込み(ストア)を一時的にライト・バッファにストアする。ライト・バッファにより、プロセッサはメモリまたはキャッシュ(あるいはその両方)への書き込み中にも命令の実行を続けることができ、そのパフォーマンスが向上する。さらに、書き込みを遅らせて一括処理できるため、メモリ・アクセス・バス・サイクルの利用効率が向上する。ライト・バッファにストアされた書き込みは、常にプログラム順どおりにメモリに書き込まれる。Pentium Proプロセッサでは、その順序管理機能により、プログラムのデータ読み取り(ロード)および書き込み(ストア)の順序と、プロセッサが実際に読み取りおよび書き込みを実行する順序の整合性が維持される。このタイプの順序管理では、見込みによる任意の順序で読み取りを実行することと、バッファ利用書き込み中も読取りを実行することができ、メモリへの書き込みは常にプログラム順どおりに実行される。

2.3 インテルMMX® テクノロジ搭載IAプロセッサ

インテルMMXテクノロジは、インテル・アーキテクチャ(IA)の拡張命令セットである。このテクノロジは、SIMD(Single Instruction, Multiple Data)方式で複数のデータ要素を並行処理することにより、マルチメディアおよび通信ソフトウェアの処理速度を向上させる。MMX命令セットでは、57の新しいオペコードと64ビットのクワッドワード・データ型を追加されている。新しい64ビット・データ型は、図2-6に示すように、MMX命令が操作するパックド整数値を保持する。

さらに、8つの新しい64ビットMMXレジスタが追加されている。これにより。レジスタ名MM0～MM7を使用して直接アドレス指定することができる。図2-7に、8つの新しいMMXレジスタのレイアウトを示す。

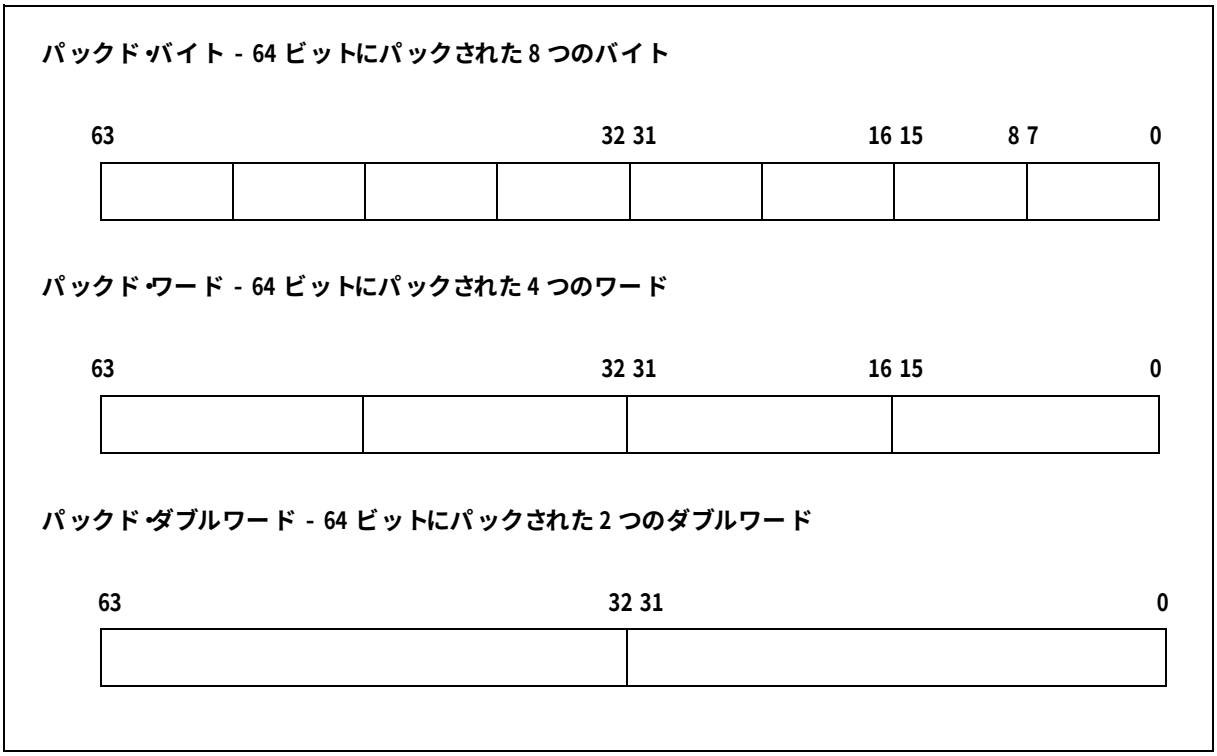


図 2-6. 新しいデータ型

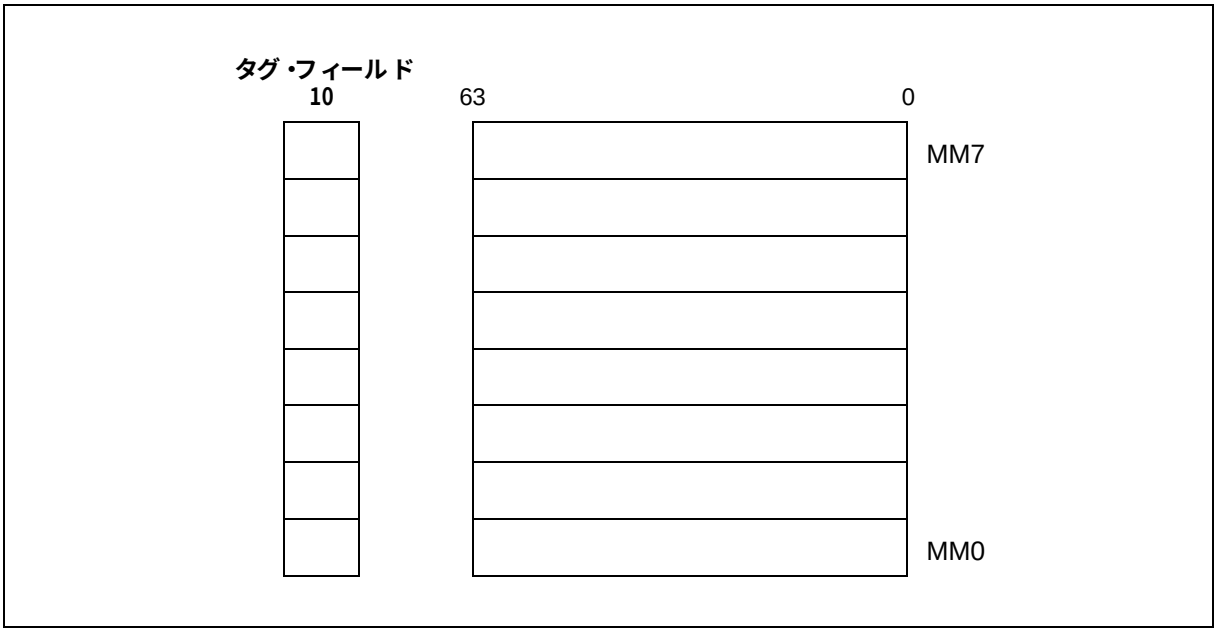


図 2-7. MMX® レジスタ・セット

インテル MMX テクノロジは、オペレーティング・システムには透過であり、すべての既存インテル・アーキテクチャ・ソフトウェアと 100% 互換である。インテル MMX テクノロジ搭載のプロセッサでは、すべてのアプリケーションをそのまま引き続き実行することができる。MMX の命令、データ型、レジスタの詳細は、『インテル・アーキテクチャ MMX® テクノロジ・プログラマーズ・リファレンス・マニュアル』資料番号 243007J) に記載されている。

2.3.1 スーパースケーラ(Pentium® プロセッサ・ファミリ)パイプライン

MMX テクノロジ Pentium プロセッサでは、パイプラインに新しいステージが追加されている。この MMX パイプラインと整数パイプラインとの統合は、浮動小数点パイプのそれに非常によく似ている。

図 2-8 に、このスキームのパイプライン構造を示す。

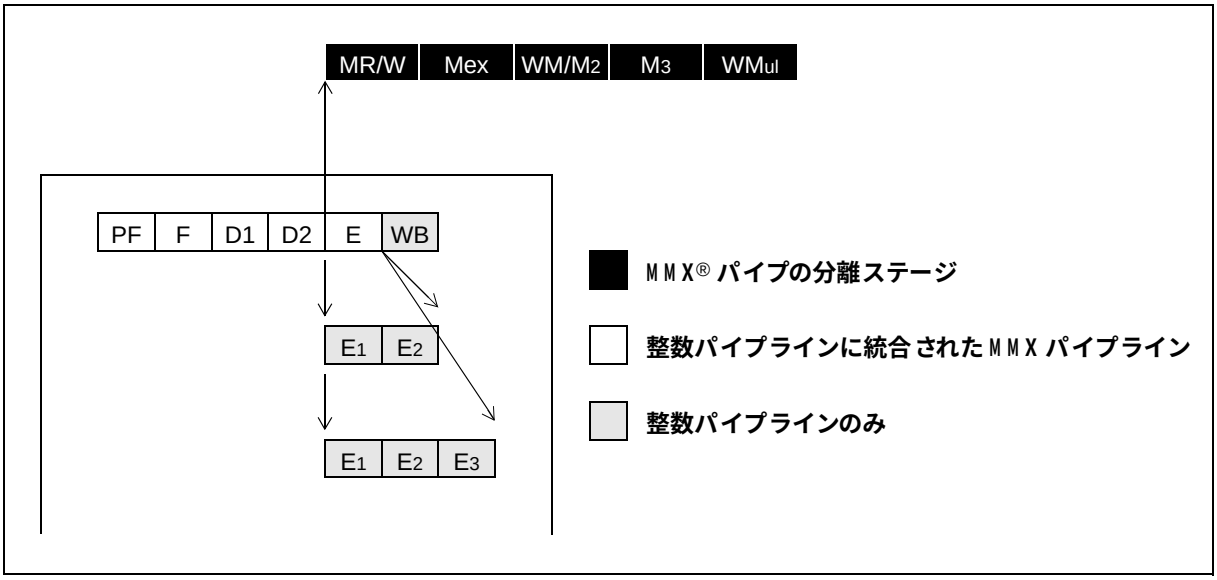


図 2-8. MMX® パイプライン構造

MMX テクノロジ Pentium プロセッサでは、整数パイプラインのステージの数が増えている。命令バイトがプリフェッチ (PF) ステージでコード・キャッシュからプリフェッチされ、フェッチ (F) ステージで命令に構文解析される。さらに、F ステージではすべてのプリフィックスがデコードされる。

命令の構文解析は、F ステージとデコード 1 (D1) ステージの間にあるファースト・イン、ファースト・アウト (FIFO) 命令バッファによる命令のデコードとは切り離されている。この FIFO には最高 4 命令分のスロットがある。この FIFO は透過である。つまり、それが空のときは追加レイテンシは発生しない。

この命令 FIFO には、1 クロック・サイクル当たり 2 つの命令をプッシュできる (コード・バイトの有無、およびプリフィックスなどその他の要因による)。FIFO からは命令ペアが抜き出され、D1 ステージに送られる。命令の平均実行レートは 1 クロック当たり 2 命令未満なので、FIFO は通常フル状態である。FIFO がフルであるかぎり、命令のフェッチおよび構文解析時に発生する可能性があるストールをバッファすることができる。つまり、そのようなストールが発生しても、そのストールがパイプの実行ステージでのストールにつながるのを阻止できる。FIFO が空の場合は、実行すべき命令がパイプラインからなくなり、実行ストールが発生する可能性がある。命令またはプリフィックスが長い場合は、FIFO の入口でストールが発生する可能性がある(3.7節および3.4.2項を参照)。

図 2-9 に、スーパースケーラ・プロセッサ上の MMX パイプラインと、そのパイプラインでストールが発生する可能性がある条件の詳細を示す。

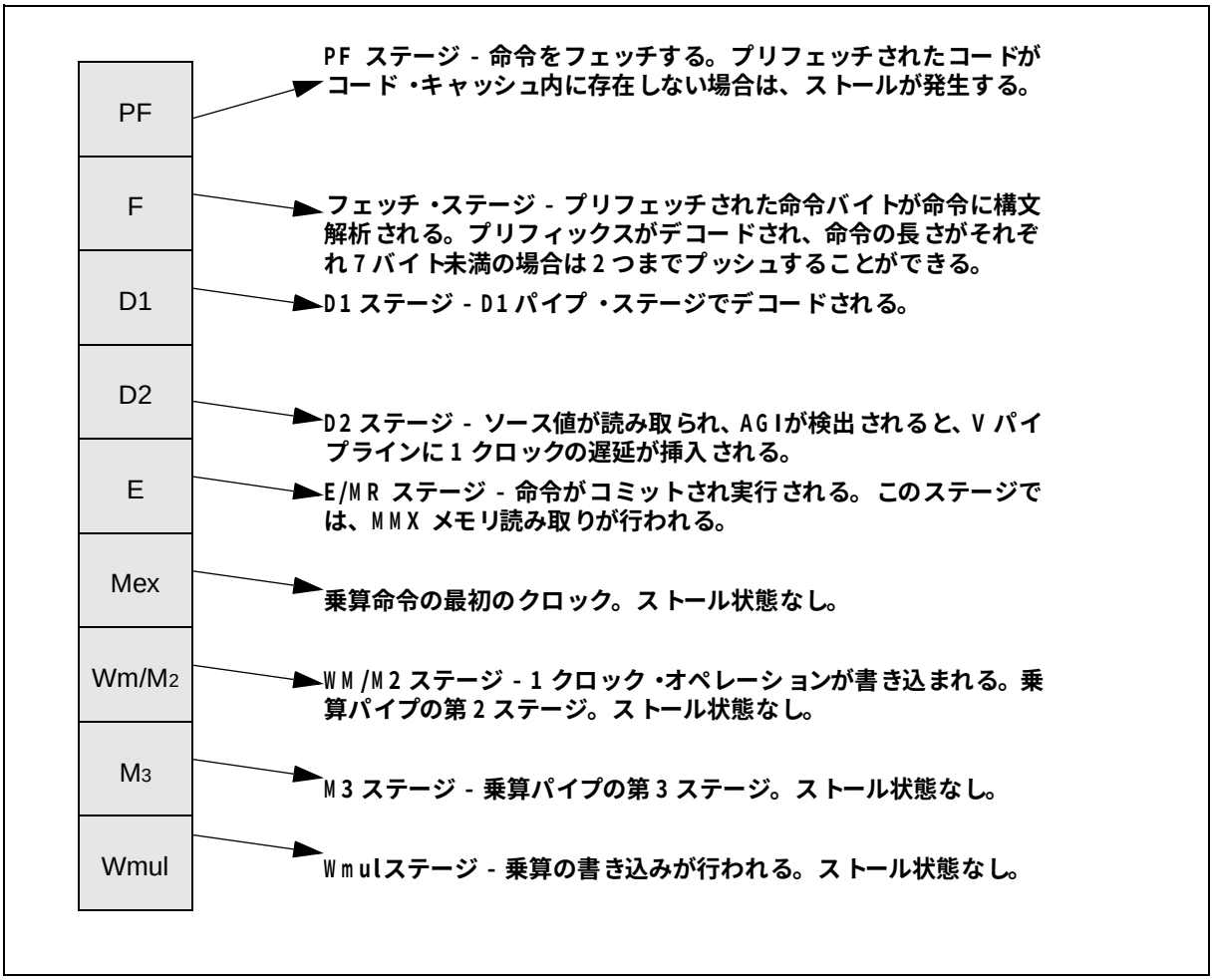


図 2-9. MMX® テクノロジ Pentium® プロセッサでの MMX® 命令フロー

表 2-2 に、各タイプの MMX 命令の機能ユニット、レイテンシ、スループット、および実行パイプの詳細を示す。

表 2-2. MMX® の命令と実行ユニット

操作	機能ユニット数	レイテンシ	スループット	実行パイプ
ALU	2	1	1	U および V
乗算	1	3	1	U または V
シフト / パック / アンパック	1	1	1	U または V
メモリ ・アクセス	1	1	1	U のみ
整数レジスタ ・アクセス	1	1	1	U のみ

- 算術論理演算ユニット(ALU)は、算術演算および論理演算 (すなわち、加算、減算、XOR、AND)を実行する。
- 乗算器ユニットは、すべての乗算を実行する。乗算には 3 サイクル必要であるが、パイプライン化が可能であり、1 クロック ・サイクルごとに乗算が 1 回行われる。乗算器ユニットはプロセッサに 1 つしかないため、乗算命令同士をペアリングすることはできない。ただし、乗算命令は他のタイプの命令とはペアリング可能である。乗算命令は、U、V いずれのパイプでも実行することができる。
- シフト ・ユニットは、すべてのシフト、パック、アンパック操作を実行する。シフタは 1 つしかなく、したがって、シフト、パック、アンパックの各命令は他のシフト ・ユニット命令とペアリングすることはできない。ただし、シフト ・ユニット命令は他のタイプの命令とはペアリング可能である。シフト ・ユニット命令は、U、V いずれのパイプでも実行することができる。
- メモリまたは整数レジスタにアクセスする MMX 命令は、U パイプでしか実行できず、MMX 命令以外のどの命令ともペアリングできない。
- MMX レジスタを更新後、そのレジスタをメモリか整数レジスタに移動するには、その前にもう 1 クロック ・サイクル待たなければならない。

ペアリングの必要条件については、3.3 節に記載してある。

命令フォーマットの詳細については、『インテル ・アーキテクチャ MMX® テクノロジ ・プログラマーズ ・リファレンス ・マニュアル』 資料番号 243007J）に記載されている。

2.3.2 Pentium® IIプロセッサ

Pentium IIプロセッサは、2.3 節で説明したものと同じパイプラインを使用する。機能上の大きな相違はMMX テクノロジが追加されていることである。表 2-3 に、Pentium IIプロセッサの実行ユニットを示す。

表 2-3. Pentium® II プロセッサの実行ユニット

ポート	実行ユニット	レイテンシ / スループット
0	整数 ALU ユニット: LEA 命令 シフト命令 整数乗算命令 浮動小数点ユニット: FADD 命令 FMUL 命令 FDIV ユニット MMX ALU ユニット MMX 乗算器ユニット	レイテンシ 1、スループット 1/ サイクル レイテンシ 1、スループット 1/ サイクル レイテンシ 1、スループット 1/ サイクル レイテンシ 4、スループット 1/ サイクル [2] レイテンシ 3、スループット 1/ サイクル レイテンシ 5、スループット 1/2 サイクル [1,2] レイテンシ - 単精度 17 サイクル、倍精度 36 サイクル、拡張精度 56 サイクル、スループット - 非パイプライン化 レイテンシ 1、スループット 1/ サイクル レイテンシ 3、スループット 1/ サイクル
1	整数 ALU ユニット MMX ALU ユニット MMX シフト・ユニット	レイテンシ 1、スループット 1/ サイクル レイテンシ 1、スループット 1/ サイクル レイテンシ 1、スループット 1/ サイクル
2	ロード・ユニット	キャッシュ・ヒット時レイテンシ 3、スループット 1/ サイクル [3]
3	ストア・アドレス・ユニット	レイテンシ 3 (該当しない)、スループット 1/ サイクル [3]
4	ストア・データ・ユニット	レイテンシ 1 (該当しない)、スループット 1/ サイクル

注記:

表 2-1 の下の注記を参照のこと。

2.3.3 キャッシュ

MMX テクノロジ Pentium プロセッサおよび Pentium IIプロセッサのオンチップ・キャッシュ・サブシステムは、32 バイト・キャッシュ・ライン長の 2 つの 16K バイト 4 ウェイ・セット・アソシエイティブ・キャッシュで構成されている。これらのキャッシュは、ライトバック・メカニズムおよび疑似 LRU 置換アルゴリズムを採用している。データ・キャッシュは、4 バイト境界でインターリーブしている 8 つのバンクからなっている。

MMX テクノロジ Pentium プロセッサでは、参照先のバンクが異なれば、データ・キャッシュには両方のパイプから同時にアクセスすることができる。Pentium Pro プロセッサ・ファミリでは、参照先のバンクが異なれば、データ・キャッシュにはロード命令およびストア命令で同時にアクセスすることができる。参照先のアドレスが同じ場合は、これらの命令はデータ・キャッシュをバイパスし、同じサイクルで実行される。MMX テクノロジ Pentium プロセッサでは、キャッシュ・ミスに対する遅延は 8 内部クロック・サイクルである。Pentium IIプロセッサでは、最小遅延は 10 内部クロック・サイクルである。

2.3.4 分岐ターゲット・バッファ

MMX テクノロジ Pentium プロセッサおよび Pentium II プロセッサの分岐予測は、1 つのマイナーな例外 (次項参照) を除けば、Pentium Pro プロセッサと同じである。その例外については、次項で説明する。

2.3.4.1 連続分岐

MMX テクノロジ Pentium プロセッサでは、下図に示すように、2 つの分岐命令の最後のバイトがアライメントが合った同じ 4 バイト・メモリ・セクションに現れたときは、分岐が誤って予測されることがある。

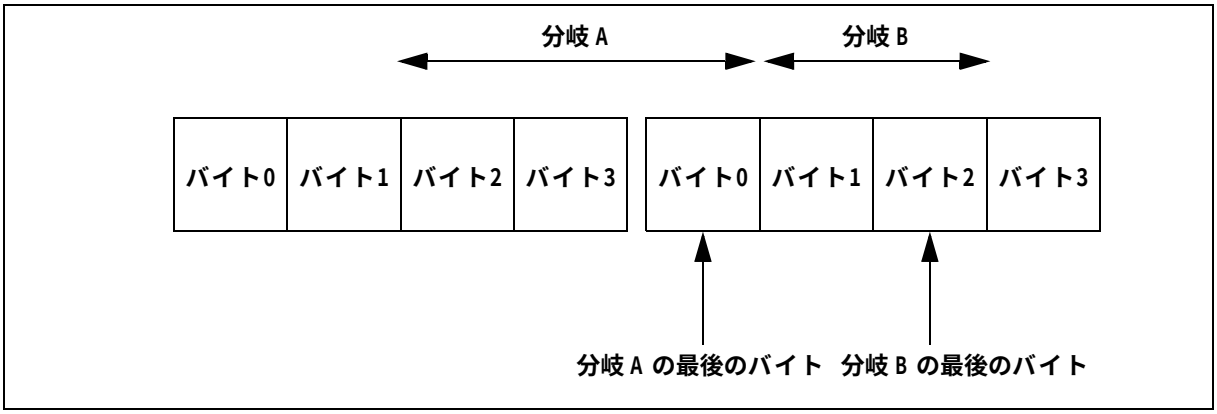


図 2-10. 連続分岐の例

この状況は、間に命令が介在しないで 2 つの分岐が連続し、かつ、2 番目の命令の長さが 2 バイトだけ (たとえば、± 128 相対ジャンプ) のときに発生することがある。

このような場合に予測ミスを避けるには、分岐命令に対して 8 ビットの相対ディスプレースメントではなく 16 ビットの相対ディスプレースメントを使用して、2 番目の分岐を長くする。

2.3.5 ライト・バッファ

MMX テクノロジ Pentium プロセッサには 4 つのライト・バッファがある (それに対して、インテル MMX テクノロジを搭載していない従来の Pentium プロセッサでは 2 つ)。さらに、これらのライト・バッファは U、V どちらのパイプからも使用できる (それに対して、従来の Pentium プロセッサでは各パイプに 1 つのバッファが対応する)。ライト・ヒットはライト・ミスをパスできない、すなわち見過ごすことができないので、メモリへの書き込みの順序をスケジュールすることにより、パフォーマンスを大きく左右するループの効率を上げることができる。ライト・ミスが現れることが予想されるときは、書き込み命令をグループ (4 つ以内) にまとめ、その次に書き込み以外の命令をはさんでから、その他の書き込み命令が来るようにスケジュールすればよい。

intel

3

整数ブレンデッド・コードの最適化手法

第 3 章
整数ブレンデッド・コードの最適化手法

本章では、インテル・アーキテクチャ全体を対象としてアプリケーションの性能を向上させる最適化手法について説明する。3.1 節では全般的ガイドラインを示し、3.2 節以降では各ガイドラインに関してさらに掘り下げて説明し、実際のコードの改善例を示す。

3.1 整数ブレンデッド・コーディングのガイドライン

以下のガイドラインは、インテル・アーキテクチャ上でコードを効率的に動作させるための最適化に有効である。

- 最適化されたアプリケーションを生成する最新世代のコンパイラを使用する。これは、最初からすぐれたコードを生成する上で効果的である。第 6 章を参照。
- 最適化を意識してコードを書き、コンパイルする。グローバル変数、ポインタ、複雑な制御フローの使用はできるだけ控える。'register' 修飾子を使用せず、'const' 修飾子を使用する。タイプ・システムを途中で変更することはせず、間接コールも行わない。
- 分岐予測のアルゴリズムに注意を払う (3.2 節を参照)。これが Pentium Pro および Pentium II プロセッサに対する最も重要な最適化である。分岐予測の精度を上げることにより、命令のフェッチに必要なサイクル数を減らすことができる。
- パーシャル・レジスタ・ストールを避ける。3.3 節を参照。
- すべてのデータのアライメントを合わせる。3.4 節を参照。
- 命令キャッシュ・ミスを最少にし、プリフェッチを最適化するようにコードを配置する。3.5 節を参照。
- Pentium プロセッサでペアリング数が最大になるようにコードの順序をスケジュールする。3.6 節を参照。
- 0F 以外のプリフィックス付きオペコードを避ける。3.7 節を参照。
- 同一のメモリ領域において、大きいストアの後に小さいロードが来ること、小さいストアの後に大きいロードが来ることを避ける。同一のメモリ領域におけるデータのロードとストアには、データ・サイズをそろえ、アドレス・アライメントを使用する。3.8 節を参照。
- ソフトウェアによるパイプライン化を図る。
- 常に CALL 命令と RET (return) 命令をペアリングする。
- 自己修正型のコードを避ける。
- データをコード・セグメント内に入れない。
- できるだけ早くストア・アドレスを計算する。
- μop を 4 つ以上含む命令、または 7 バイトより長い命令を避ける。できれば、μop を 1 つしか必要としない命令を使用する。
- 呼び出される側のセーブ・プロシージャを呼び出す前にパーシャル (部分) レジスタをゼロにクリアする。

3.2 分岐予測

分岐の最適化は、Pentium Pro および Pentium II プロセッサにとって最も重要な最適化である。これらの最適化は、Pentium プロセッサ・ファミリにも有益である。分岐フローを理解し、分岐予測の精度を上げることにより、コードの実行速度を大幅に向上させることができる。

3.2.1 動的分岐予測

動的分岐予測については、以下の 3 点を押さえておく必要がある。

1. 命令アドレスがBTB 内に存在しない場合は、分岐せずに次の命令を引き続き実行する(フォール・スルーする)と予測される。
2. 分岐すると予測された分岐は1クロックの遅延を生じる。
3. BTB は、Pentium Pro プロセッサ、Pentium Ⅱプロセッサ、およびＭＭＸ テクノロジPentium プロセッサでは、4ビットの分岐予測履歴を保持する。Pentium プロセッサでは、2ビットの分岐予測履歴を保持する。

命令プリフェッチ・プロセスでは、条件付き命令の命令アドレスが BTB のエントリと照合される。アドレスがBTB 内に存在しないときは、実行は次の命令にフォール・スルーすると予測される。これは、分岐の後には実行されるコードが必要であることを意味する。分岐の後のコードがフェッチされ、Pentium Pro および Pentium Ⅱプロセッサの場合は、フェッチされた命令が見込みで実行される。したがって、分岐命令の後にデータが来るようなことがあってはならない。

さらに、分岐命令の命令アドレスが BTB 内に存在し、かつその分岐が行われると予測された場合、その命令はPentium Pro および Pentium Ⅱプロセッサ上では1クロックの遅延を生じる。分岐する分岐に対するこの1クロックの遅延を避けるには、単に、分岐すると予想される分岐と分岐の間に他の作業を挿入するだけでよい。この遅延のために、ループの最小サイズは2クロック・サイクルに制限される。所要時間が2クロック・サイクル未満の小さいループについては、それをアンロールすることで分岐命令の1クロックのオーバーヘッドを除去できる。

Pentium Pro プロセッサ、Pentium Ⅱプロセッサ、およびＭＭＸ テクノロジPentium プロセッサの分岐予測メカニズムでは、規則的パターン(4クロック長まで)の分岐については正しく予測する。たとえば、奇数回目の繰り返しでは分岐し、偶数回目の繰り返しでは分岐しないループであれば、その中の分岐は正しく予測される。

3.2.2 Pentium® Pro および Pentium II プロセッサでの静的予測

Pentium Pro および Pentium II プロセッサでは、BTB 内に履歴が存在しない分岐は、以下のように静的予測アルゴリズムにより予測される。

- 無条件分岐は分岐すると予測する。
- 条件付き後方分岐は分岐すると予測する。この規則はループに適している。
- 条件付き前方分岐は分岐しない (フォール・スルーする) と予測する。

静的に予測される分岐では、プリフェッチに最高 6 サイクルが費やされる可能性がある。予測が外れた場合のペナルティは 12 クロックを超える。以下に、静的分岐予測アルゴリズムを示す。

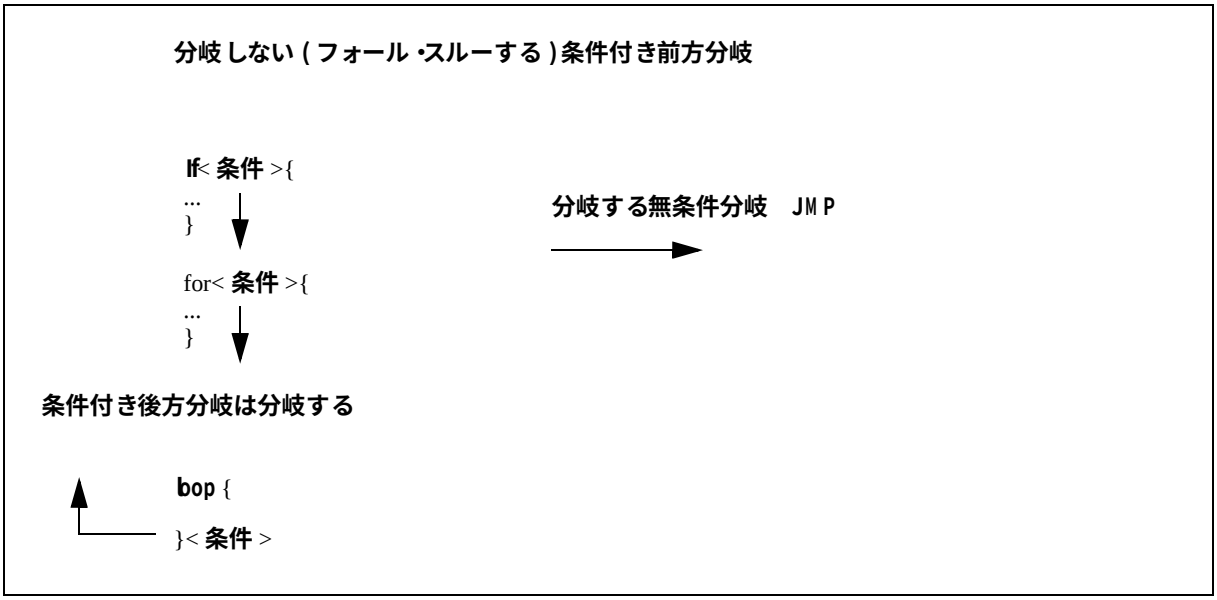


図 3-1. Pentium® Pro および Pentium II プロセッサの静的分岐予測アルゴリズム

以下の例は、静的予測アルゴリズムの基本的規則を示す。

```
Begin:   MOV  EAX, mem32
         AND  EAX, EBX
         IMUL EAX, EDX
         SHLD EAX, 7
         JC   Begin
```

この例では、最初のパスでは後方分岐 (JC Begin) の履歴が BTB 内に存在しないため、BTB による分岐予測は行われないが、静的予測により分岐すると予測される。したがって予測ミスはない。

```
         MOV  EAX, mem32
         AND  EAX, EBX
         IMUL EAX, EDX
         SHLD EAX, 7
         JC   Begin
         MOV  EAX, 0
Begin:   CALL Convert
```

このコード・セグメント内の最初の分岐命令 (JC Begin) は、条件付き前方分岐である。最初のパスではこの分岐のエントリは BTB に存在しないが、静的予測によってフォール・スルーすると予測される。

CALL Convert 命令は、最初のパスでは BTB による予測は行われないが、このコールは静的予測アルゴリズムによって分岐すると予測される。これは無条件分岐に対しては正しい。

これらの例では、条件付き分岐には分岐するかしないかの 2 つの道しかない。スイッチ・ステートメント、計算型 GOTO、コール・スルー・ポインタなどの間接分岐の場合は、ジャンプ先のアドレスは不定である。分岐先が特定のアドレスに大きく集中している (全分岐の 90% が同じアドレスに分岐する) 場合、BTB はほとんどの場合正確に予測する。しかし、分岐先が予測できない場合、パフォーマンスは一気に低下する可能性がある。そのような場合は、間接分岐を予測可能な条件付き分岐に変更することにより、パフォーマンスを向上させることができる。

3.2.3 分岐の除去と削減

分岐を除去すると、以下の理由によってパフォーマンスが向上する。

- 予測ミスの回避
- 必要な BTB エントリ数の削減

分岐は、setcc 命令を使用するか、または Pentium Pro プロセッサの条件付き移動 (CMOV または FCMOVE) 命令を使用することで除去することができる。

以下に、2 つの定数のいずれか 1 つに依存する条件付きの C コードの例を示す。

```
ebx = (A<B) ? C1 : C2;
```

このコードでは、条件によって 2 つの値 A と B を比較する。この条件が真であれば EBX は C1 に設定され、偽であれば C2 に設定される。これと等価なアセンブリ言語コードを以下に示す。

```
     cmp A, B                    ;condition
     jge   L30                   ;conditional branch
     mov   ebx, CONST1
     jmp   L31                   ;unconditional branch
L30:
     mov   ebx, CONST2
L31:
```

上の例の jge 命令を setcc 命令に置き換えても、EBX レジスタは C1 または C2 に設定される。このコードは、以下の例に示すように最適化すれば分岐を除去することができる。

```
xor  ebx, ebx                    ;clear ebx
cmp   A, B
setge bl                         ;when ebx = 0 or 1
                                 ;OR the complement condition
dec   ebx                        ;ebx=00...00 or 11...11
and   ebx, (CONST2-CONST1)            ;ebx=0 or(CONST2-CONST1)
add   ebx, min(CONST1,CONST2)    ;ebx=CONST1 or CONST2
```

最適化されたコードでは、EBX をゼロに設定し、次に A と B を比較する。A が B よりも大きいかまたは等しい場合は、EBX は 1 に設定される。EBX は次にデクリメントされ、両方の定数値の差との論理積 (AND) が取られる。その結果、EBX はゼロか両値の差に設定される。2 つの定数の小さい方を加算することにより、正しい値が EBX に書き込まれる。CONST1 または CONST2 の値がゼロのときは、正しい値がすでに EBX に書き込まれているので、最後の命令は削除することができる。

abs(CONST1-CONST2) が {2,3,5,9} のいずれか 1 つであるときは、以下の例があてはまる。

```
xor   ebx, ebx
cmp     A, B
setge bl       ; or the complement condition
lea   ebx, [ebx*D+ebx+CONST1-CONST2]
```

ここで、D は abs(CONST1-CONST2)-1 を表す。

Pentium Pro または Pentium II プロセッサでは、新しい CMOV 命令と FCMOV 命令を使用することでも分岐をなくすことができる。以下に、CMOV を使用してテストと分岐の命令シーケンスを変更し、分岐を除去する例を示す。テストが equal フラグをセットした場合は、EBX の値が EAX に移動される。この分岐はデータに依存する、予測不可能な分岐の典型である。

```
test ecx, ecx
jne  1h
mov  eax, ebx
1h:
```

このコードを変更するには、jne 命令と mov 命令を CMOVcc 命令に置き換え、それに equal フラグをチェックさせる。最適化されたコードを以下に示す。

```
test      ecx, ecx       ;test the flags
cmoveq    eax, ebx       ;if the equal flag is set, move ebx to eax
1h:
```

ラベル 1h: は、別の分岐命令の分岐先でないかぎり、必要ではなくなる。これらの命令は、一世代前のプロセッサで使用すると無効なオペコードを生成する。したがって、必ず CPUID 命令を使用して、アプリケーションが Pentium Pro または Pentium II プロセッサで動作していることを確認されたい。

分岐の除去に関する詳細については、VTune の 『Pentium Pro Processor Computer Based Training (CBT)』 で確認されたい。

分岐の除去の他に、次の方法でも分岐予測の精度を向上させることができる。

- コールとリターンが対になっていることを確認する。
- データと命令を混在させない。
- 非常に短いループはアンロールする。
- 静的予測アルゴリズムに従う。

3.2.4 分岐予測に関するパフォーマンス・チューニングの秘訣

3.2.4.1 Pentium® プロセッサ・ファミリ

MMX テクノロジ搭載の有無を問わず、Pentium プロセッサでパイプラインのフラッシュが生じる最も一般的な理由は、分岐すべき分岐に対する BTB の予測ミス (BTB ミス) である。パイプラインのフラッシュが頻繁に発生する場合は、アプリケーション内の分岐の振舞いを調べる必要がある。VTune を使用すれば、パフォーマンス・カウンタを以下の各イベントに設定してプログラムを評価することができる。

1. パイプライン・フラッシュによる合計オーバーヘッドをチェックする。

   BTB ミスによるパイプライン・フラッシュの合計オーバーヘッドは、以下の式で求められる。

   (分岐予測ミスによるパイプライン・フラッシュの回数) * 4/(WB ステージでの分岐予測ミスによるパイプライン・フラッシュの回数)

   注記

   パイプラインにステージが追加されたため、MMX テクノロジ Pentium プロセッサでの分岐予測ミスのペナルティは、Pentium プロセッサより 1 サイクル大きくなる。

2. BTB 予測レートをチェックする。

   BTB ヒット・レートは、以下の式で求められる。

   (BTB 予測回数)/(分岐した回数)

BTB ヒット・レートが低い場合は、アクティブな分岐の数は BTB エントリの数より多い。上記のイベントのモニタリングについては、第 7 章で詳細に説明する。

3.2.4.2 Pentium® Pro および Pentium II プロセッサ

予測ミスが発生すると、当の分岐命令までパイプライン全体がフラッシュされ、プロセッサは予測を誤った分岐がリタイアするのを待つ。

(分岐予測ミス比率) = BR_Miss_Pred_Ret/Br_Inst_Ret

分岐予測ミス比率が約5%未満であれば、分岐予測は正常に行われているものとみなしてよい。この比率が約5%を超える場合は、どの分岐によって重大な予測ミスが生じているのかを確認し、3.2.3項の手法に従って状況の解決を図る。

3.3 Pentium® Proおよび Pentium IIプロセッサでのパーシャル・レジスタ・ストール

Pentium Proおよび Pentium IIプロセッサでは、16または8ビット・レジスタ(AL、AH、AXなど)が書き込まれた直後に32ビット・レジスタ(EAXなど)が読み取られると、書き込みがリタイアする(最低7クロック・サイクルを要する)まで読み取りがストールされる。次に示す例では、最初の命令で値8をAXレジスタに転送し、次の命令でEAXレジスタにアクセスする。このコード・シーケンスでは、パーシャル・レジスタ・ストールを生じる。

```
MOV AX, 8
ADD  ECX, EAX
the EAX
```

レジスタのアクセス時にパーシャル・ストールが発生する

これは、以下に示す8および16ビット/32ビット・レジスタのすべての組み合わせにあてはまる。

小さいレジスタ:			大きいレジスタ:
AL	AH	AX	EAX
BL	BH	BX	EBX
CL	CH	CX	ECX
DL	DH	DX	EDX
		SP	ESP
		BP	EBP
		DI	EDI
		SI	ESI

Pentiumプロセッサでは、このペナルティは生じない。

Pentium Pro および Pentium Ⅱプロセッサでは、コードをアウトオブオーダで実行できるので、命令の直後に命令が続いてもストールが発生するとは限らない。以下の例でも、パーシャル・ストールが発生する。

```
MOV AL, 8
MOV EDX, 0x40
MOV EDI, new_value
ADD EDX, EAX
the EAX
```

レジスタ・アクセス時にパーシャル・ストールが発生する

μop でストールが発生すると、やはり後続の μop もすべてストールする。最初にストールが発生した μop がパイプ内の次のステージに進まなければ、後続の μop もストールから脱することができない。一般的に、ストールを避けるには、大きい(32 ビット)レジスタ(EAX)の一部である小さい(16 または 8 ビット)レジスタ(AL)の書き込みの後に大きいレジスタ(EAX)の読み取りが行われないようにすればよい。

さまざまな世代のプロセッサを対象とするブレンデッド・コードの生成を容易にするため、Pentium Pro および Pentium Ⅱプロセッサでは、小さいレジスタと大きいレジスタの組み合わせで読み書きをする特殊なケースがインプリメントされている。このような特殊なケースは、次の各例に示すように、EAX、EBX、ECX、EDX、EBP、ESP、EDI、ESIに対して XOR と SUB を実行する場合に備えてインプリメントされている。

```
xor   eax, eax
movb   al, mem8
add  eax, mem32
```

パーシャル・ストールなし

```
xor   eax, eax
movw   al, mem16
add  eax, mem32
```

パーシャル・ストールなし

```
sub    ax, ax
movb   al, mem8
add  eax, mem16
```

パーシャル・ストールなし

```
sub   eax, eax
movb   al, mem8
or   eax, mem16
```

パーシャル・ストールなし

```
xor    ah, ah
movb   al, mem8
sub   ax, mem16
```

パーシャル・ストールなし

一般的に、このシーケンスをインプリメントするときは、大きいレジスタをゼロにクリアしてからそのレジスタの下位半分に書き込む。

3.3.1 パーシャル・ストールに関するパフォーマンス・チューニングの秘訣

3.3.1.1 Pentium® プロセッサ

Pentium プロセッサでは、パーシャル・ストールは発生しない。

3.3.1.2 Pentium® Pro および Pentium II プロセッサ

パーシャル・ストールは、VTune の Renaming Stalls イベントで計測する。このイベントは、duration イベントまたは count イベントとしてプログラムすることができる。duration イベントは、プロセッサがストールしていた継続時間 (各イベントのサイクル数) の合計をカウントするのに対し、カウント・イベントはイベント発生回数の合計をカウントする。Renaming Stalls イベントの cmsk を count (カウント) か duration (デュレーション) に設定するのは、VTune の [Custom Events] ウィンドウで行う。デフォルトは duration である。duration を使用した場合、パーシャル・ストールによるストール時間の割合 (%) を次の式で求めることができる。

(リネーミング・ストールの継続時間)/(合計サイクル数)

継続時間が実行時間の約 3% を超えるストールがある場合は、コーディングし直してそのストールを除去する。

3.4 アライメントに関する規則とガイドライン

・ 以降の各項で、コードおよびデータのアライメントについて説明する。

Pentium プロセッサ・ファミリでは、アライメントが合わないアクセスを行うと 3 サイクルのコストを要する。Pentium Pro および Pentium II プロセッサでは、アライメントが合わない (キャッシュ・ライン境界にまたがる) アクセスを行うと、6 ～ 9 サイクルのコストを要する。32 バイト・ライン境界にまたがるメモリ・アクセスのことを、データ・キャッシュ・ユニット (DCU) スプリットという。アライメントが合わないアクセスで DCU スプリットが生じると、Pentium Pro および Pentium II プロセッサではストールが生じる。最高のパフォーマンスを得るためには、32 バイトより大きいデータ構造体および配列では、構造体または配列要素のアライメントを 32 バイト境界に合わせ、かつデータ構造体または配列要素へのアクセス・パターンがこのアライメント規則に違反しないようにする必要がある。

3.4.1 コード

Pentium、Pentium Pro、および Pentium II プロセッサのキャッシュ・ライン・サイズは 32 バイトである。プリフェッチ・バッファは 16 バイト境界でフェッチするので、コードのアライメントが合っているかどうかはプリフェッチ・バッファの効率に直接影響する。

インテル・アーキテクチャ・ファミリ全体を対象に、最適のパフォーマンスが得られるようにするには、以下のガイドラインを守るようお勧めする。

- 16 バイト境界からの隔たりが 16 バイト未満のときは、ループ・エントリ・ラベルはアライメントを 16 バイト境界に合わせる。
- 条件分岐の後のラベルはアライメントを合わせない。
- 16 バイト境界からの隔たりが 8 バイト未満のときは、無条件分岐または関数コールの後のラベルはアライメントを 16 バイト境界に合わせる。

MMX テクノロジ Pentium プロセッサ、Pentium Pro プロセッサ、および Pentium II プロセッサでは、実行が 2 サイクル未満のループを避ける。非常に高密度のループでは、シーケンス内のいずれかの命令が 16 バイト境界にまたがり、命令のデコードに余分なサイクルを要する確率が高い。このような状況に陥ると、Pentium プロセッサでは、1 回置きのループの繰り返しで余分なサイクルが 1 つずつ発生する。Pentium Pro および Pentium II プロセッサでは、実行可能な命令の数が制限され、したがって、サイクルごとにリタイアさせられる命令の数も制限される。パフォーマンスを大きく左右するループ・エントリは、キャッシュ・ライン境界に合わせて配置するようお勧めする。さらに、実行が 2 サイクル未満のループはアンロールするのが望ましい。Pentium Pro および Pentium II プロセッサでのデコードの詳細については、2.2 節を参照されたい。

3.4.2 データ

Pentium プロセッサでは、データ・キャッシュ内またはバス上でアライメントが合わないアクセスを行うと、最低 3 クロック・サイクルの余分なコストを要する。Pentium Pro および Pentium II プロセッサでは、データ・キャッシュ内でアライメントが合わない (キャッシュ・ライン境界にまたがる) アクセスを行うと、9 ～ 12 クロック・サイクルのコストが生じる。どのプロセッサ上でも最高の実行パフォーマンスが得られるよう、データのアライメントを以下のガイドラインに従って境界に合わせるようお勧めする。

- 8 ビット・データは任意の境界にアライメントを合わせる。
- 16 ビット・データはアライメントが合った 4 バイト・ワード内に収まるようにアライメントを合わせる。
- 32 ビット・データは 4 バイトの整数倍の任意の境界にアライメントを合わせる。
- 64 ビット・データは 8 バイトの整数倍の任意の境界にアライメントを合わせる。
- 80 ビット・データは 128 ビット境界 (すなわち、16 バイトの整数倍の任意の境界) にアライメントを合わせる。

3.4.2.1 32 バイト以上のデータ構造体と配列

32 バイト以上のデータ構造体または配列は、各構造体または配列要素の先頭を 32 バイト境界に合わせ、かつ、各構造体または配列要素が 32 バイト・キャッシュ・ライン境界をまたがないようにアライメントを合わせる。

3.4.3 データ・キャッシュ・ユニット (DCU) スプリット

以下の例で、DCU スプリットを生じる可能性のあるコードのタイプを示す。このコードは 2 つのダブルワード配列のアドレスをロードする。この例では、4 回ごとのループの繰り返しの最初の 2 つのダブルワードのロードで DCU スプリットが生じる。アドレス 029e70feh に宣言されているデータが 32 バイト境界に合っていないため、図 3-2 に示すように、このアドレスへの各ロードと、このアドレスから 32 バイト目ごと (4 回ごとの繰り返し) に行われるロードは、キャッシュ・ライン境界にまたがることになる。

```
                        mov     esi, 029e70feh
                        mov     edi, 05be5260h
        BlockMove:
                        mov     eax, DWORD PTR [esi]
                        mov     ebx, DWORD PTR [esi+4]
                        mov     DWORD PTR [edi], eax
                        mov     DWORD PTR [edi+4], ebx
                        add     esi, 8
                        add     edi, 8
                        dec     edx
                        jnz     BlockMove
```

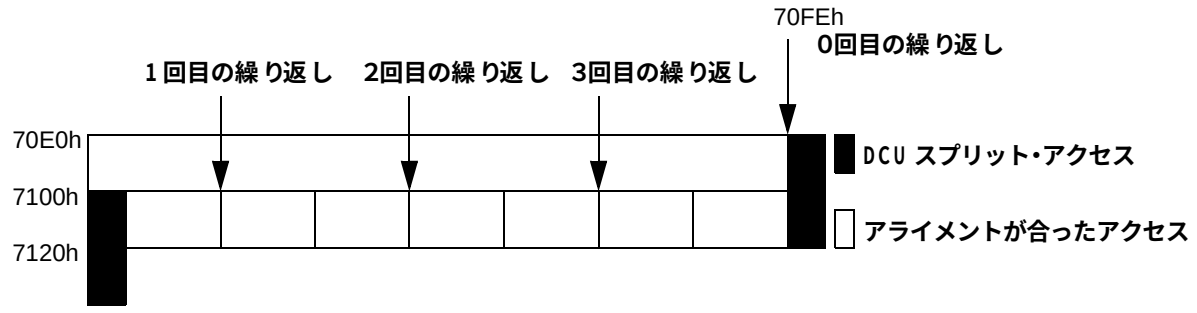


図 3-2. データ・キャッシュ内の DCU スプリット

3.4.4 アライメントが合わないアクセスに関するパフォーマンス・チューニングの秘訣

3.4.4.1 Pentium® プロセッサ

Pentium プロセッサでは、データのアライメントが合わないアクセスが行われると、3 サイクルのストールが生じる。VTune の動的実行機能を使用すると、アライメントが合わないアクセスの場所を正確につきとめることができる。

3.4.4.2 Pentium® Pro および Pentium II プロセッサ

Pentium Pro プロセッサでは、Misaligned Accesses イベント・カウンタを使用してアライメントが合っていないデータを検出することができる。アライメントが合っていないデータがキャッシュ・ライン境界にまたがると、6 ～ 12 サイクルのストールが生じる。

3.5 データの配置変更によるキャッシュ使用効率の向上

キャッシュの振舞いによって、アプリケーションのパフォーマンスは大きく変化する。キャッシュの動作の仕組みをよく理解し、それに見合ったコード / データ構造を選択することで、キャッシュの使用効率を大きく上げることができる。各プロセッサのキャッシュ構造については、第 2 章で説明している。

3.5.1 C 言語レベルの最適化

以降の各項で、C 言語レベルでどのようにデータ配置を改善できるかについて説明する。これらの最適化は、すべてのプロセッサに対して有益である。

3.5.1.1 キャッシュ使用効率向上のためのデータ宣言

変数の割り当ては一般にコンパイラによって制御されるため、最適化後に変数をメモリ内の任意の (開発者が意図する) アドレスに配置させることはできない。コンパイラは、特に構造体および配列の値については、言語規格に準拠する宣言順どおりにメモリ内に配置する。しかし、インライン・アセンブリを関数に挿入する場合は、多くのコンパイラでは最適化がオフになるので、その関数内でどのようにデータを宣言するかが重要になる。さらに、データをアセンブリ・レベルで宣言するときは、データ宣言の順序も重要である。DCU スプリットまたはデータのアライメントずれは、場合によっては、高水準コードまたはアセンブリ・コード内のデータのレイアウトを変更することにより回避できる。以下の例について検討されたい。

最適化されていないデータのレイアウト:

```
short a[15];        /* 2 bytes data */
int   b[15], c[15]; /* 4 bytes data */

for (i=0; i<15, i++) {
     a[i] = b[i] + c[i]
}
```

コンパイラによっては、メモリの割り当ては変数の宣言に伴って行われ、したがって、上の例の変数のキャッシュ・レイアウトは以下のようになる。

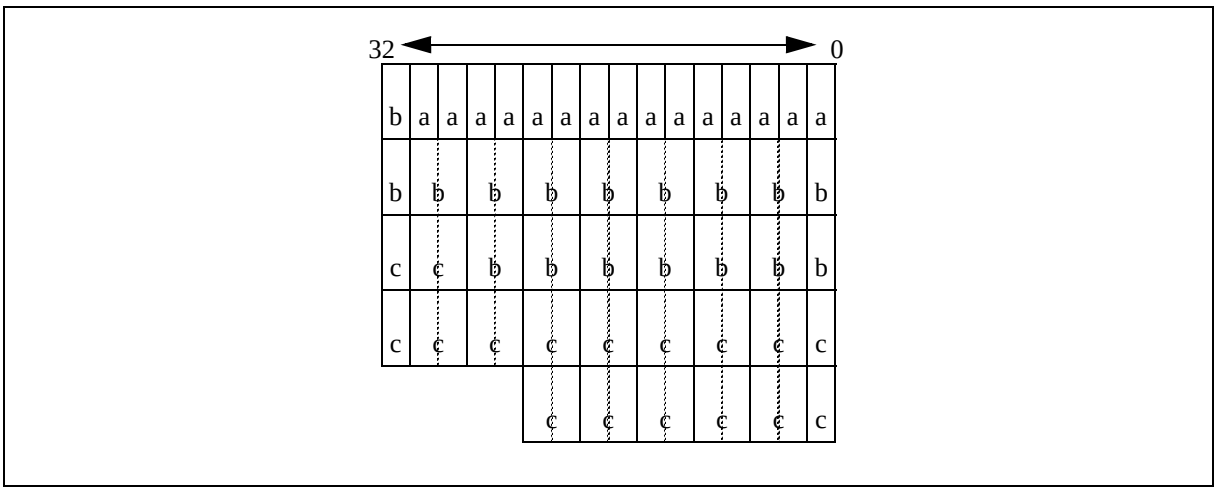


図 3-3. 構造 a、b、c のキャッシュ・レイアウト

この例では、a[0] はキャッシュ・ライン境界にアライメントが合っているものと想定している。図中、1 つのボックスは 2 バイトに相当する。Pentium Pro プロセッサでは、要素 b[0]、b[8]、c[1]、c[9] にアクセスしたとき、キャッシュ DCU スプリットが生じる。

この例では、大きい要素を先に宣言するようにすれば、アライメントずれを避けることができる。

最適化されたデータ・レイアウト:

```
int   b[15], c[15]; /* 4 bytes data */
short a[15];        /* 2 bytes data */

for (i=0; i<15, i++) {
    a[i] = b[i] + c[i]
}
```

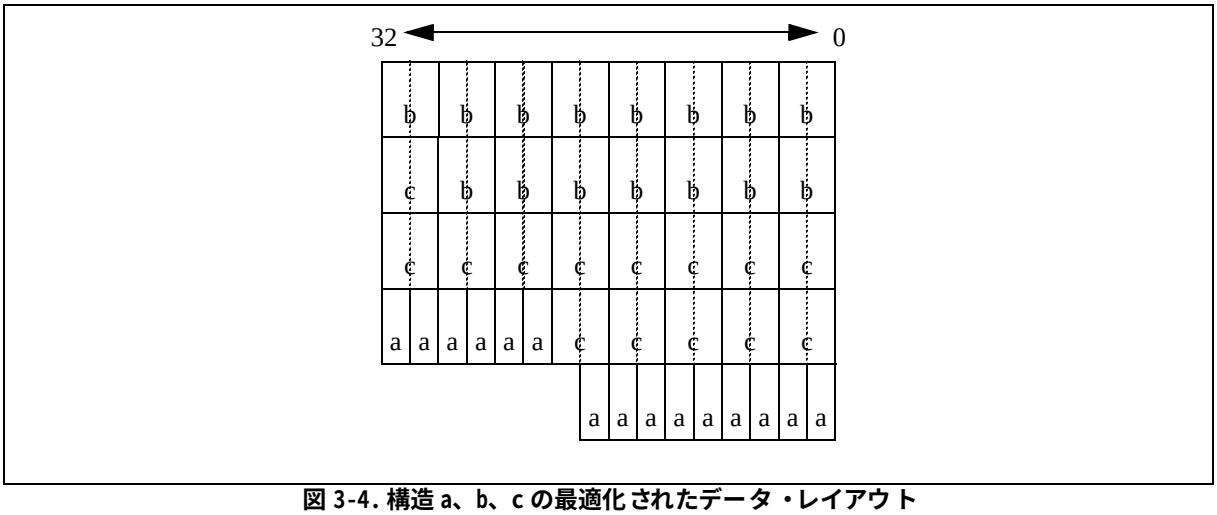


図 3-4. 構造 a、b、c の最適化されたデータ・レイアウト

Pentium Pro および Pentium II プロセッサでは、上記のデータがアクセスされても DCU スプリットは生じない。

3.5.1.2 データ構造体の宣言

データ構造体の宣言によって、構造体内データへのアクセス速度が大きく変わる場合がある。次項以降で、C コードでのアクセス速度を簡単に向上させることのできる方法について説明する。

最善の策は、データ構造体の占有スペースをできるだけ小さくすることである。これは、以下のガイドラインに従って配列および構造体を宣言することで実現できる。

- データ構造体の先頭を 32 バイト境界に合わせる。
- 構造体の個々の要素がキャッシュ・ライン境界をまたがないようにデータを配置する。
- 要素は大きいものから順に宣言する。
- 同時にアクセスされるデータ要素をまとめて配置する。
- アクセス頻度の高いデータ要素をまとめて配置する。

構造体内に大きい配列をどのように宣言するかは、コード内でそれらの配列にどのようにアクセスするかによる。そのような配列は、たとえば次のコード・セグメントに示すように、2 つの独立した配列の構造体として、または構造体の複合配列として宣言することになる。

独立配列：

```
struct    {
     int a[500];
     int b[500];
     } s;
```

複合配列：

```
struct    {
     int a;
     int b;
     } s[500];
```

独立配列を宣言した場合は、メモリ内には配列 a の要素が順次に配置され、その後に配列 b の要素が配置される。複合配列を宣言した場合は、各配列の要素が交互に、つまり、a[i] の後に b[i] という具合に配置される (図 3-5 参照)。

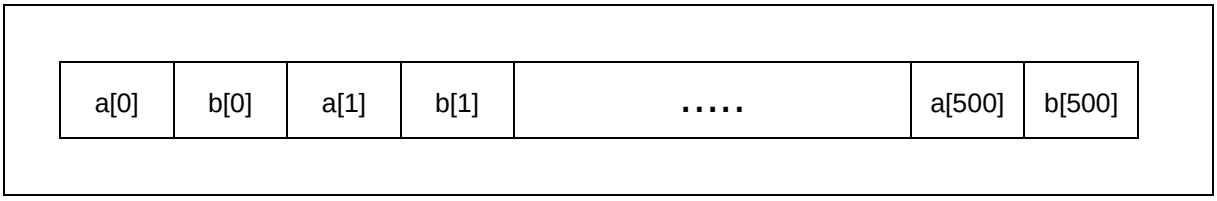


図 3-5. メモリ内における複合配列の格納状態

コードが配列 a および b に順次にアクセスする場合は、両方の配列を独立させて宣言する。このように宣言すると、キャッシュ・ライン・フィルにおいては、要素 a[i] がキャッシュされると同時にその配列の隣接要素もキャッシュされる。コードが配列 a および b に並行してアクセスする場合は、複合配列として宣言する。複合配列では、キャッシュ・ライン・フィルにおいては、要素 a[i] がキャッシュされると同時に要素 b[i] もキャッシュされる。

3.5.1.3 配列および構造体のパディングとアライメント合わせ

配列および構造体に対してパディングとアライメント合わせを実行すると、構造体または配列がランダムにアクセスされることによるキャッシュ・ミスを避けるのに効果的である。順次にアクセスする構造体または配列は、メモリ内にも順次に配置する必要がある。

以下のガイドラインに従うと、キャッシュ・ミスを減らすことができる。

- 各構造体に対してパディングを行い、そのサイズをキャッシュ・ライン・サイズの整数倍にする。
- キャッシュ・ラインの先頭 (Pentium および Pentium Pro プロセッサの場合、32 の整数倍) に合わせて各構造体をアライメントする。
- 配列の次元数を 2 のべき乗にする。

これらの手法の詳細および実例については、Pentium プロセッサのコンピュータ・ベース・トレーニングを参照されたい。

3.5.1.4 ループ変換によるメモリ使用効率の向上

メモリ内のデータ構造の選択の他に、コードのデータ・アクセス方法を改善することによってもキャッシュの使用効率を上げることができる。以下に、メモリ・アクセス・パターンを改善するための主な変換方法をいくつか示す。内側ループ内の参照を単位ストライドにし、キャッシュ内で 1 つでも多くの処理を実行させることができればよい。

「ループ・フュージョン」(Loop Fusion) とは、同じデータにアクセスする 2 つのループを融合させて、目的のデータがキャッシュ内にあるうちに、1 つでも多くの処理を実行できるようにすることである。

ループ・フュージョン前：

```
for (i = 1; i < n){
  ... A(i) ...
}
for (i = 1; i < n){
  ... A(i) ...
}
```

ループ・フュージョン後：

```
for (i = 1; i < n){
  ... A(i) ...
  ... A(i) ...
}
```

「ループ・フィッション」(Loop Fission) とは、キャッシュされたデータの処理が完了する前にキャッシュからフラッシュされないよう、ループを 2 つのループに分割することである。

ループ・フィッション前：

```
for (i = 1; i < n){
  ... A(i) ...
  ... B(i) ...
}
```

ループ・フィッション後：

```
for (i = 1; i < n){
  ... A(i) ...
}
for (i = 1; i < n){
  ... B(i) ...
}
```

ループを入れ換えると、データのアクセス方法が変わる。C コンパイラではマトリックスを行順にメモリに格納するのに対し、FORTRAN コンパイラではマトリックスを列順に格納する。メモリ内の格納順にデータにアクセスすることにより、多くのキャッシュ・ミスを避け、使用効率を向上させることができる。

ループ入れ換え前：

```
for (i = 1; i < n){for (j = 1; j < n){
     for (j = 1; j < n){
          ... A(i,j) ...
     }
}
```

ループ入れ換え後：

```
     for (i = 1; i < n){
          ... A(i,j) ...
     }
}
```

ブロック化とは、データ・アクセス時のキャッシュ・ミスをできるだけ少なくできるようにプログラムの構造を変更するプロセスである。ブロック化は、非常に大きなマトリックスの乗算に効果的である。

ブロック化前：

```
for (j = 1; j < n){
     for (i = 1; i < n){

       ... A(i,j) ...

     }
     for (i = ii; i < ii+k) {
}
          ... A(i,j) ...
```

ブロック化後：

```
     for (jj = 1; i < n jj+= k){
          for (ii = 1; ii < n; ii+=k){

               for (j = jj; i < jj+k){

                    }
               }
          }
     }
```

注記

これらの変換の一部は、プログラムによっては規格に合わない場合がある。また、アルゴリズムによっては、これらの変換を実施したとき異なる結果を生じる場合もある。これらの最適化手法の詳細については、『High Performance Computing』Kevin Dowd, O'Reilly and Associates, Inc., 1993、および『High Performance Compilers for Parallel Computing』Michel Wolfe, Addison Wesley Publishing Company, Inc., 1996, ISBN 0-8053-2730-4 を参照されたい。

3.5.1.5 メモリ内およびスタック上のデータのアライメント合わせ

Pentium プロセッサでは、8 バイト境界にアライメントが合っていない 64 ビット変数にアクセスすると、3 サイクルの余分なコストを要する。Pentium Pro および Pentium II プロセッサでは、そのような変数が 32 バイト・キャッシュ・ライン境界にまたがっていると、DCU スプリットを生じることがある。市場に出回っているコンパイラのなかには、倍精度データのアライメントを 8 バイト境界に合わせないものもある。Misaligned Accesses カウンタにより、データのアライメントが合っていないことが判明した場合は、以下の方法でデータのアライメントを合わせることができる。

- 静的変数を使用する。
- データのアライメントを明示的に合わせるアセンブリ・コードを使用する。
- C コードでは、malloc を使用して明示的に変数を割り当てる。

静的変数

スタック上に変数を割り当てると、変数のアライメントが合わない場合がある。コンパイラは静的変数をスタック上には割り当てず、メモリ内に割り当てる。コンパイラが静的変数を割り当てると、ほとんどの場合、静的変数のアライメントは合っている。

```
static float a;                    float b;
static float c;
```

アセンブリ言語によるアライメント合わせ

アセンブリ・コードを使用して、明示的に変数のアライメントを合わせる。以下の例では、スタックのアライメントを 64 ビット境界に合わせている。

プロシージャの開始部：

```
push   ebp
mov    esp, esp
and    ebp, -8
sub    esp, 12
```

プロシージャの終了部：

```
add    esp, 12
pop    ebp
ret
```

malloc による動的割り当て

動的割り当てを使用する場合は、コンパイラがダブルまたはクワッドワード値のアライメントを 8 バイト境界に合わせるかどうかを確認する。コンパイラがこのアライメント合わせを行わない場合は、以下の手法に従う。

- 目的の配列または構造体のサイズに 4 バイトを加えたサイズに等しいメモリを割り当てる。
- ビット単位の論理積を使用して、配列のアライメントが合っていることを確認する。

例：

```
double a[5];
double *p, *newp;

p = (double*)malloc ((sizeoff(double)*5)+4)
newp = (p+4) & (-7)
```

3.5.2 大きいメモリ・ブロックの移動

Pentium Pro および Pentium II プロセッサで大きいデータ・ブロックをコピーするときは、プロセッサの拡張機能をイネーブルにしてコピーの速度を向上させることができる。この特殊モードを使用するには、データのコピーに際して以下の基準を満たさなければならない。

- ソースおよびデスティネーションのアライメントが 8 バイト境界に合っていなければならない。
- コピーの方向が昇順でなければならない。
- データの長さに必要な繰り返し回数が 64 より大でなければならない。

これら 3 つの基準がすべて満たされているときは、ライブラリ関数の代わりに rep movs および rep stos 命令を使用して関数をプログラムすれば、文字列を高速コピーすることができる。さらに、コピーに長い時間を要するアプリケーションの場合は、これらの基準に従ってデータをセットアップすることにより、アプリケーション全体の速度を向上させることができる。

以下に、1 ページをコピーする例を示す。

```
MOV ECX, 4096        ; instruction sequence for copying a page
MOV EDI, destpageptr ; 8-byte aligned page pointer
MOV ESI, srcpageptr  ; 8-byte aligned page pointer
REP MOVSB
```

3.5.3 ライン・フィルの順序

キャッシュ可能なアドレスへのデータ・アクセスがデータ・キャッシュをミスしたときは、外部メモリからそのキャッシュ・ライン全体がキャッシュに入れられる。これをライン・フィルという。Pentium、Pentium Pro、および Pentium Ⅱプロセッサでは、このようなデータは4つの8バイトのセクションからなるバーストで、以下に示すバースト順で送られてくる。

最初のアドレス	2番目のアドレス	3番目のアドレス	4番目のアドレス
0h	8h	10h	18h
8h	0h	18h	10h
10h	18h	0h	8h
18h	10h	8h	0h

MMX テクノロジ Pentium プロセッサ、Pentium Pro および Pentium Ⅱプロセッサの場合は、データはメモリから送られてくる順序で使用可能になる。データの配列を連続して読み取る場合は、各データ項目がメモリから送られてくる順序で使用できるように、データに順次にアクセスすることが望ましい。Pentium プロセッサでは、最初の8バイト・セクションは即時に使用可能になるが、2番目以降のセクションは1キャッシュ・ライン全体がメモリから読み取られるまでは使用可能にならない。

サイズが32バイトの整数倍である配列の場合は、その配列の先頭をキャッシュ・ラインの先頭に合わせる。アライメントを32バイト境界に合わせることにより、ライン・フィルの順序付けを利用し、キャッシュ・ライン・サイズを揃えることができる。サイズが32バイトの整数倍でない配列の場合は、先頭を32または16バイト境界(それぞれキャッシュ・ラインの先頭または中央)に合わせる。アライメントを16または32バイト境界に合わせるには、データをパディングしなければならない場合がある。それが必要な場合は、パディングするスペース内でデータを(変数または定数がないか)探してみる。

3.5.4 メモリ・フィル・バンド幅の拡張

メモリのアクセスおよびフィルがどのように行われるかを理解することは有益である。メモリからメモリへのフィル(たとえば、メモリからビデオへのフィル)は、即座に(ビデオ・フレーム・バッファなどの)メモリにストア・バックされるメモリからの32バイト(キャッシュ・ライン)のロードとして定義されている。次項以降に、順次メモリ・フィル(ビデオ・フィル)のバンド幅を広げ、レイテンシを短縮するためのガイドラインを示す。これらのガイドラインの内容は、すべてのインテル・アーキテクチャ・プロセッサに関連し、ロードやストアが第2レベル・キャッシュでヒットしないケースにあてはまる。メモリのバンド幅の詳細については、第4章を参照されたい。

3.5.5 書き込み割り当ての効果

Pentium Pro および Pentium Ⅱプロセッサには、"所有権のための読み取りによる書き込みアロケート"型キャッシュがあり、Pentium プロセッサには"書き込みアロケートなし、書き込みミス時ライト・スルー"型キャッシュがある。

Pentium Pro および Pentium Ⅱプロセッサでは、書き込みが発生し、その書き込みがキャッシュをミスすると、32バイトのキャッシュ・ライン全体がフェッチされる。一方、Pentium プロセッサで同様の書き込みミスが発生すると、単にその書き込みがメモリに送られる。

順次ストアはバースト書き込みに併合され、データはキャッシュに保持されていて後でロードすることができるので、書き込みアロケートは何かと有利である。これが、P6 ファミリ・プロセッサでこの書き込みアロケートを採用している理由であり、また、Pentium プロセッサが書き込みミス時ライト・スルーを採用しているにもかかわらず、一部のPentium プロセッサ・システムの設計でやはりこの戦略がL2キャッシュに導入されている理由である。

書き込みアロケートは、以下のようなコードでは不利な場合がある。

- キャッシュ・ラインの1部分だけに書込む。
- キャッシュ・ライン全体を読み取らない。
- ストライドが32バイトのキャッシュ・ライン・サイズを超える。
- 書き込みを多数のアドレス(>8000)に行う。

下のプログラム例に示すように、同一のアプリケーション内で多数の書き込みが行われ、かつストライドが32バイト・キャッシュ・ラインより大きく、配列もまた大きいときは、Pentium Pro または Pentium Ⅱプロセッサでのすべてのストアで1キャッシュ・ライン全体がフェッチされる。さらに、このフェッチでは1つ(場合によっては2つ)のダーティ・キャッシュ・ラインが置き換えられる可能性が高い。

結果として、ストアのたびに余計なキャッシュ・ライン・フェッチが発生し、プログラムの実行速度が低下する。プログラム内で多数の書き込みが行われると、パフォーマンスが大きく低下することもある。下のエラストテネスのふるい法プログラムで、これらのキャッシュの影響の実例を示す。この例では、1ステップごとにストライドを順次に広げながら大きな配列全体に単一値の0が書き込まれていく。

注記

これは、キャッシュの影響を示すことだけを目的とした非常に単純化した例であり、このコードには他にもさまざまな最適化が可能である。

例:エラストテネスのふるい法

```
boolean array[max];
for(i=2;i<max;i++) {
     arry = 1;
    }

for(i=2;i<max;i++) {
     if( array[i] ) {
         for(j=2;j<max;j+=i) {
                    array[j] = o;  /* here we assign memory to 0 causing the */
                                   /* cache line fetch within the j loop     */
         }
      }
}
```

この例に対しては、2つの最適化が可能である。1つは、配列をビット単位にパックし、それにより配列のサイズを縮小し、さらにその結果としてキャッシュ・ラインのフェッチ回数を減少させるものである。もう1つは、書き込みの前に値をチェックし、それによりメモリへの書き込み回数(ダーティ・キャッシュ・ライン数)を減少させるものである。

3.5.5.1 最適化 1:boolean

上のプログラムの boolean は char 配列である。プログラムによっては、(キャッシュ・プロトコルはすべての読み取りを読み取り - 修正 - 書き込みにするので) 読み取り - 修正 - 書き込みが行われるように、boolean 配列をパックされたビットの配列にした方がずっと有利な場合もある。しかし、この例では、そのほとんどのストライドが 256 ビット (1 キャッシュ・ラインのビット数) を超えるので、パフォーマンスの向上はあまりない。

3.5.5.2 最適化 2:書き込み前チェック

もう 1 つの最適化は、書き込む前に値がすでにゼロであるかどうかをチェックするものである。

```
boolean array[max];
for(i=2;i<max;i++) {
     array = 1;
    }
    for(i=2;i<max;i++) {
     if( array[i] ) {
         for(j=2;j<max;j+=i) {
          if( array[j] != 0 ) { /* check to see if value is already 0 */
              array[j] = 0;
          }
         }
     }
    }
```

ふるい法プログラムでは、全実行時間を通じて、データはすでにゼロであることが圧倒的に多いため、外部バス活動は半減する。最初にチェックすることにより、読み取りには 1 つのバースト・バス・サイクルしか必要でなく、書き込みが抑止されるたびにバースト・バス・サイクルが節約される。修正されたラインの実際のライト・バックは必要でなくなり、したがって余分なサイクルが節約される。

注記

この操作は、Pentium Pro および Pentium II プロセッサにとっては有利であるが、Pentium プロセッサについてはパフォーマンス向上につながらない場合があり、すべてのプロセッサに対して最適化できるとは考えられない。順次ストアはバースト書き込みと併合され、データはキャッシュ内に保持されていて後でロードすることができるので、書き込みアロケートを行うとほとんどのシステムでパフォーマンスの向上を期待できる。これが、Pentium Pro および Pentium II プロセッサでこの戦略を採用している理由であり、また、一部の Pentium プロセッサ・ベースのシステムで、その L2 キャッシュの設計にこの戦略が導入されている理由である。

3.6 整数命令のスケジューリング

スケジューリングまたはパイプライン化は、全世代のプロセッサにわたってパフォーマンスを最適化するような方法で行われることが望まれる。以下に、Pentium、Pentium Pro、および Pentium II プロセッサ上で、コードの速度を向上させるペアリングおよびスケジューリングに関する規則の一覧を示す。場合によっては、特定のプロセッサでの最適化にはトレードオフが介在することもある。これらのトレードオフは、アプリケーション個々の特性によって異なる。

スーパースケーラ型 Pentium プロセッサでは、命令の順序が最適化の重要な鍵である。命令の順序を変更すると、2 つの命令を同時に処理できる可能性が大きくなる。データへの依存性がある命令が続く場合は、その間に他の命令を 1 つ (以上) はさむようにする。

3.6.1 ペアリング

本節では、整数命令のペアリングに関する規則を示す。MMX 命令および浮動小数点命令のペアリング規則については、それぞれ第 4 章と第 5 章に記載してある。ペアリングを可能にするためには、以下の規則を守らなければならない。

- 整数ペアリング規則 - 整数命令のペアリングのための規則。
- 一般的ペアリング規則 - 特定のオペコードにではなく、マシン・ステータスに依存する規則。これらの規則は整数および浮動小数点にも有効である。たとえば、命令のペアリングを可能にするためには、シングル・ステップはディスエーブルにする必要がある。
- MMX 命令同士のペアリング規則 - 2 つの MMX 命令のペアリングを可能にする規則。乗算器ユニットは 1 つしか存在しないので、通常では、プロセッサは2つのMMX命令を同時に発行することはできない。4.3節を参照。
- MMX 命令 / 整数命令ペアリング規則 - 1 つの整数命令と 1 つの MMX 命令のペアリングを可能にする規則。4.3 節を参照。
- 浮動小数点 / 整数ペアリング規則 - 5.3 節を参照。

注記

浮動小数点命令は、MMX 命令とペアリングすることはできない。

3.6.2 整数ペアリング規則

以下の条件が成立するときは、ペアリングを行うことはできない。

- 次の 2 つの命令がペアリング可能な命令でない (個々の命令のペアリング特性については付録 A を参照されたい)。一般的に、大部分の単純な ALU 命令はペアリング可能である。
- 次の 2 つの命令間にある種の (暗黙または明示の) レジスタ競合が生じる。ただし、レジスタの競合があっても例外的にペアリング可能なこともある。それらについては後で説明する。
- 次の 2 つの命令がどちらも命令キャッシュ内に存在しない。ただし、最初の命令が 1 バイト命令である場合は例外的にペアリングすることができる。

表 3-1. 整数命令のペアリング

U パイプでペアリング可能な整数命令			V パイプでペアリング可能な整数命令		
mov r,r	alu r,i	push r	mov r,r	alu r,i	push r
mov r,m	alu m,i	push i	mov r,m	alu m,i	push l
mov m,r	alu eax,i	pop r	mov m,r	alu eax,i	pop r
mov r,i	alu m,r	nop r	mov r,i	alu m,r	jmp near
mov m,i	alu r,m	shift/rot by 1	mov m,i	alu r,m	jcc near
mov eax,m	inc/dec r	shift by imm	mov eax,m	inc/dec r	OF jcc
mov m,eax	inc/dec m	test reg,r/m	mov m,eax	inc/dec m	call near
alu r,r	lea r,m	test acc,imm	alu r,r	lea r,m	nop
				test reg,r/m	test acc,imm

3.6.2.1 命令セットのペアリング性

ペアリング不可能な命令 (NP)

1. CL レジスタにシフト・カウントがあるシフトおよびローテート命令。
2. ロング算術演算命令。たとえば、MUL、DIV。
3. 拡張命令。たとえば、RET、ENTER、PUSHA、MOVS、STOS、LOOPNZ。
4. 一部の浮動小数点命令。たとえば、FSCALE、FLDCW、FST。
5. セグメント間命令。たとえば、PUSH sreg、CALL far。

U または V パイプに発行されるペアリング可能な命令 (UV)

1. 大部分の 8/32 ビット ALU 演算。たとえば、ADD、INC、XOR。
2. すべての 8/32 ビット比較命令。たとえば、CMP、TEST。
3. レジスタを使用するすべての 8/32 ビット・スタック操作。たとえば、PUSH reg、POP reg。

U パイプに発行されるペアリング可能な命令 (PU)

これらの命令は、U パイプに対して発行しなければならず、V パイプ内の適切な命令とペアリングすることができる。これらの命令は、決して V パイプ内では実行されない。

1. キャリーおよびボロー命令。たとえば、ADC、SBB。
2. プリフィックス付き命令 (3.7 節を参照)。
3. 即値によるシフト。
4. 一部の浮動小数点演算。たとえば、FADD、FMUL、FLD。

V パイプに発行されるペアリング可能な命令 (PV)

これらの命令は、U パイプでも V パイプでも実行できるが、V パイプにあるときだけにペアリングされる。これらの命令は、命令ポインタ (eip) を変更するので、次の命令が隣接していない場合があるため、U パイプではペアリングすることはできない。U パイプ内の分岐は、分岐しないと予測されたときでも、次の命令とペアリングしない。

1. 単純な制御転送命令。たとえば、call near、jmp near、jcc。これには、jcc ショートおよび jcc ニア (0f プリフィックス付き) 両バージョンの条件付きジャンプ命令が含まれる。
2. fxch 命令。

3.6.2.2 レジスタ依存性による非ペアリング性

命令のペアリングは、命令のオペランドによっても影響を受ける。以下に示す組み合わせは、レジスタの競合が生じるためにペアリングすることはできない。これらの規則の例外については、3.6.2.3 項に示す。

1. 最初の命令がレジスタに書き込み、2 番目の命令がそのレジスタから読み取る (フロー依存性)。

   例：
   ```
   mov eax, 8
   mov [ebp], eax
   ```

2. 両方の命令が同じレジスタに書き込む (出力依存性)。

   例：
   ```
   mov eax, 8
   mov eax, [ebp]
   ```

この制限は、EFLAGS レジスタに書き込む命令のペア (たとえば、条件コードを変更する 2 つの ALU 演算) には適用されない。ペアリングされた両方の命令が実行された後の条件コードは、V パイプ命令からの条件になる。

最初の命令がレジスタから読み取り、2 番目の命令がそのレジスタに書き込む (反依存性) 命令のペアは、ペアリングできることに注意されたい。

例：
```
mov eax, ebx
mov ebx, [ebp]
```

レジスタの競合を判断するために、バイト・レジスタまたはワード・レジスタへの参照は、それを含む 32 ビット・レジスタへの参照として扱われる。したがって、以下の例では、EAX レジスタの内容への出力依存性のためにペアリングしない。

例：
```
mov al, 1
mov ah, 0
```

3.6.2.3 特殊ペア

一部の命令は、上記の一般的規則にかかわらずペアリング可能である。これらの特殊ペアは、レジスタ依存性に左右されない。そして、ほとんどの場合、esp レジスタへの暗黙の読み取り / 書き込みまたは条件コードへの暗黙の書き込みを伴う。

スタック・ポインタ:

- `push reg/imm; push reg/imm`
- `push reg/imm; call`
- `pop reg ; pop reg`

条件コード:

- `cmp ; jcc`
- `add ; jne`

PUSH 命令と POP 命令という特殊なペアでは、即値オペランドまたはレジスタ・オペランドしか持つことができず、メモリ・オペランドは持つことができないことに注意されたい。

3.6.2.4 ペア実行に関する制限

命令のなかには、同時に発行できても、並行して実行されないものがある。

1. 両方の命令が同じデータ・キャッシュ・メモリ・バンクにアクセスする場合は、2 番目の要求 (V パイプ) は最初の要求が完了するのを待たなければならない。2 つの物理アドレスのビット 2 ～ 4 が同じときは、バンクの衝突が生じる。バンクの衝突は、V パイプの命令に 1 クロックのペナルティを生じる。
2. 実行におるパイプ間並行処理を通じて、メモリ・アクセスの順序付けは維持される。U パイプのマルチサイクル命令は、その最後のメモリ・アクセスまで単独で実行される。

```
add  eax, mem1
add  ebx, mem2              ; 1
(add)   (add)               ; 2 2-cycle
```

上記の命令は、どちらもレジスタの内容とメモリ位置の値を加算し、得られた値をそのレジスタに格納するというものである。メモリ・オペランドを使用する加算は、実行に 2 クロックを要する。最初のクロックではキャッシュから値をロードし、2 番目のクロックでは加算を実行する。U パイプの命令にはメモリ・アクセスが 1 つしかないので、V パイプ内の加算は同じサイクルで開始することができる。

```
add eax, mem1                   ; 1
(add)                           ; 2
(add)add mem2, ebx              ; 3
(add)                           ; 4
(add)                           ; 5
```

上記の命令は、レジスタの内容をメモリ位置に加算し、得られた値をそのメモリ位置にストアするというものである。メモリ内の結果 (値) を使用する加算は、実行に 3 クロックを要する。最初のクロックで値をロードし、2 番目のクロックで加算を実行し、3 番目のクロックで結果をストアする。ペアリングされていると、U パイプの命令の最後のサイクルは V パイプ命令の実行の最初のサイクルとオーバーラップする。

すでに実行中の命令が完了するまでは、その他の命令はいっさい実行を開始することはできない。

命令のスケジューリングおよびペアリングの機会を増大させるためには、一連の単純な命令を発行する方が、同じサイクル数を要する複雑な命令を 1 つ発行するよりも得策である。単純な命令のシーケンスでは、発行の回数が多く、それだけスケジューリングやペアリングの機会が増える。ロード / ストア形式のコードの生成では、必要なレジスタ数が多くなり、コード・サイズが大きくなる。これは、二次的影響にすぎないものの、Intel486 プロセッサのパフォーマンスには影響を与える。レジスタが余分に必要であるという不利を埋め合わせるため、並行実行性が高くなると認められない限りレジスタの使用数は増やさないよう、レジスタの割り当てと命令のスケジューリングには特別に配慮する必要がある。

3.6.3 ペアリングに関する一般規則

MMX テクノロジ Pentium プロセッサについては、下記のように、一部の一般規則が緩和されている。

- Pentium プロセッサでは、2 つの命令のどちらか一方が 7 バイトより長い場合は、それらの命令をペアリングしない。MMX テクノロジ Pentium プロセッサでは、最初の命令が 11 バイトより長いかまたは 2 番目の命令が 7 バイトより長い場合は、それらの命令をペアリングしない。プリフィックスはカウントしない。
- Pentium プロセッサでは、プリフィックス付きの命令は U パイプでのみペアリング可能である。MMX テクノロジ Pentium プロセッサでは、0Fh、66h、または 67h プリフィックス付きの命令は V パイプでもペアリング可能である。

上の 2 つのケースではともに、MMX テクノロジ Pentium プロセッサでは、FIFO への入口でのストールによってペアリングが妨げられる。

3.6.4 Pentium® Pro および Pentium II プロセッサでのスケジューリングの規則

Pentium Pro および Pentium II プロセッサには、2.2 節で説明したように、インテル・アーキテクチャ (IA) のマクロ命令をマイクロ・オペレーション (µop) に変換する 3 つのデコーダがある。これらのデコーダの制限は、以下のとおりである。

- 最初のデコーダ (0) は以下の命令をデコードすることができる。
  - µop が 4 つ以下の命令。
  - 長さが 7 バイト以下の命令。
- 他の 2 つのデコーダは 1 µop の命令をデコードする。

付録 C に、すべてのインテル・マクロ命令およびそれらがデコードされる µop 数の一覧表を記載してある。この表で、各命令をどのデコーダでデコードできるかがわかる。

マクロ命令は、順次に (インオーダで) パイプを通ってデコーダの処理行程に入れられる。したがって、マクロ命令は、次に使用可能になる (空く) デコーダに適合しない場合、次の 1 デコード・サイクルを待たなければならない。インオーダ・パイプライン内の命令のストールの確率を下げるように、デコーダに対して命令をスケジュールすることができる。

以下の例について検討されたい。

- マルチ µop 命令に対して次に使用可能になるデコーダがデコーダ 0 でない場合は、そのマルチ µop 命令は、そのクロックの間他のデコーダを空のまま放置して、デコーダ 0 が使用可能になる (通常は次のクロック) のを待つ。したがって、以下の 2 つの命令はデコードに 2 サイクルを要する。

```
add  eax, ecx                ; 1 uop instruction  (decoder 0)
add  edx, [ebx]              ; 2 uop instruction (stall 1 cycle wait
                             ;   till decoder 0 is available)
```

- デコード・サイクルの開始時に、連続する 2 つの命令がともに µop を 2 つ以上生成する場合は、デコーダ 0 が一方の命令をデコードし、次の命令は次のサイクルまでデコードされない。

```
add  eax, [ebx]              ; 2 uop instruction  (decoder 0)
mov  ecx, [eax]              ; 2 uop instruction (stall 1 cycle wait
                             ;   untill decoder 0 is available)
add  ebx, 8                  ; 1 uop instruction  (decoder 1)
```

`op reg, mem` 形式の命令には、2 つの µop、すなわち、メモリからのロードの µop と操作の µop を必要とする。デコーダ・テンプレート (4-1-1) 向けにスケジュールすることにより、アプリケーションのデコード・スループットを改善することができる。

一般に、`op reg, mem` 形式の命令は、メモリ拘束型でないコード内で、かつデータがキャッシュ内にあるときに、レジスタの負担を軽減するために使用される。Pentium Pro および Pentium II プロセッサでの速度を上げるには、単純な命令を多用すればよい。

MMX テクノロジ Pentium プロセッサで `op reg, mem` 命令を使用するときは、以下の規則に従う。

- Pentium プロセッサのパイプ内のストールを最小限に抑えるようにスケジュールする。できるだけ多数の単純な命令を使用する。一般に、Pentium プロセッサのパイプライン向けに最適化された 32 ビットのアセンブリ・コードは、Pentium Pro および Pentium II プロセッサでも効率よく実行される。
- Pentium プロセッサ向けにスケジュールするときは、Pentium Pro および Pentium II プロセッサでの主なストール条件とデコーダ・テンプレート (4-1-1) を念頭に置く (以下の例を参照)。

```
pmaddwd  mm6, [ebx]          ; 2 uop instruction  (decoder 0)
paddd     mm7, mm6           ; 1 uop instruction  (decoder 1)
add       ebx, 8             ; 1 uop instruction  (decoder 2)
```

3.7 プリフィックス付きオペコード

Pentium プロセッサでは、プリフィックス付きの命令は、その命令がプリフィックスなしの場合に両パイプ (UV) または U パイプ (PU) でペアリング可能であれば、U パイプでペアリング可能 (PU) である。プリフィックスは U パイプに発行され、各プリフィックスは 1 サイクルでデコードされる。次に命令が U パイプに発行され、そこでペアリングされることもある。

Pentium プロセッサ、Pentium Pro、および Pentium II プロセッサについて、避けるべきプリフィックスは以下のものである。

- ロックument is pro
- セグメント・オーバーライド ith Intel pro
- アドレス・サイズ e、express o
- オペランド・サイズ e, 6express o
- 2 バイト・オペコード・マップ (0f) プリフィックス

MMX テクノロジ Pentium プロセッサでは、命令にプリフィックスが付いていると、構文解析の遅延を生じ、命令のペアリングが禁止されることがある。

以下に、FIFO に対する命令プリフィックスの重要な影響を示す。

- 0F プリフィックス付き命令にはペナルティはない。
- 66h または 67h プリフィックス付き命令は、プリフィックスの検出に 1 クロックを要し、長さの計算にさらに 1 クロックを要し、FIFO への投入にさらにもう 1 クロックを要する (合計 3 クロック・サイクルを要する)。FIFO に投入されるためには、最初の命令でなければならず、2 番目の命令は、最初の命令とともにプッシュできる。
- その他 (0Fh、66h、67h 以外) のプリフィックス付き命令は、各プリフィックスの検出に 1 クロック・サイクルを要する。それらの命令は、最初の命令としてだけ FIFO にプッシュされる。プリフィックスが 2 つの命令は、FIFO にプッシュされるのに 3 クロック・サイクル (プリフィックスに 2 クロック・サイクル、および命令に 1 クロック・サイクル) を要する。2 番目の命令は、最初の命令とともに同じクロック・サイクルで FIFO にプッシュすることができる。

FIFO に最低 2 つのエントリが保持されていれば、パフォーマンスに対する影響はない。デコーダ (D1 ステージ) にデコードする命令が 2 つあるかぎり、ペナルティはない。2 つの命令が 1 クロック・サイクルで FIFO から引き出されると、FIFO はすぐに空になる。したがって、プリフィックス付き命令の直前の命令にパフォーマンス上の損失 (たとえば、ペアリング不成立、キャッシュ・ミスによるストール、アライメントずれなど) が生じても、プリフィックス付き命令のパフォーマンスに対するペナルティは表面には現れないことがある。

Pentium Pro および Pentium II プロセッサでは、7 バイトより長い命令の場合は、1 サイクルでデコードされる命令の数が制限される (2.2 節を参照)。プリフィックスがあると、命令が 1 または 2 バイト長くなり、デコーダが制限される可能性がある。

プリフィックス付き命令はできるだけ使用しないことが望ましい。やむを得ず使用する場合は、他のなんらかの理由でパイプをストールさせる他の命令の後にプリフィックス付き命令が続くようにスケジュールするようお勧めする。

3.8 アドレッシング・モード

Pentium プロセッサでは、レジスタがベース・コンポーネントとして使用されていて、そのレジスタが直前の命令のデスティネーションである場合 (すべての命令がすでにプリフェッチ・キューにある場合)、1 クロック余計に使用される。

例:

```
add  esi, eax              ; esi is destination register
mov  eax, [esi]            ; esi is base, 1 clock penalty
```

1. Pentium プロセッサには整数パイプラインが 2 つあるので、レジスタが (いずれかのパイプで) 実効アドレス計算のベース・コンポーネントまたはインデックス・コンポーネントとして使用されていて、そのレジスタが直前のクロック・サイクルのどちらかの命令のデスティネーションである場合、1 クロック・サイクル余計に必要になる。これを、アドレス生成インターロック (AGI) という。AGI を避けるには、そのような命令が連続する場合に、間に他の命令を入れることにより、それらの命令を最低 1 サイクル分離すればよい。MMX レジスタはベース、インデックスどちらのレジスタとしても使用できないので、AGI は MMX レジスタがデスティネーションである場合にはあてはまらない。
2. Pentium Pro および Pentium II プロセッサ上では、AGI 条件に対するペナルティは生じない。

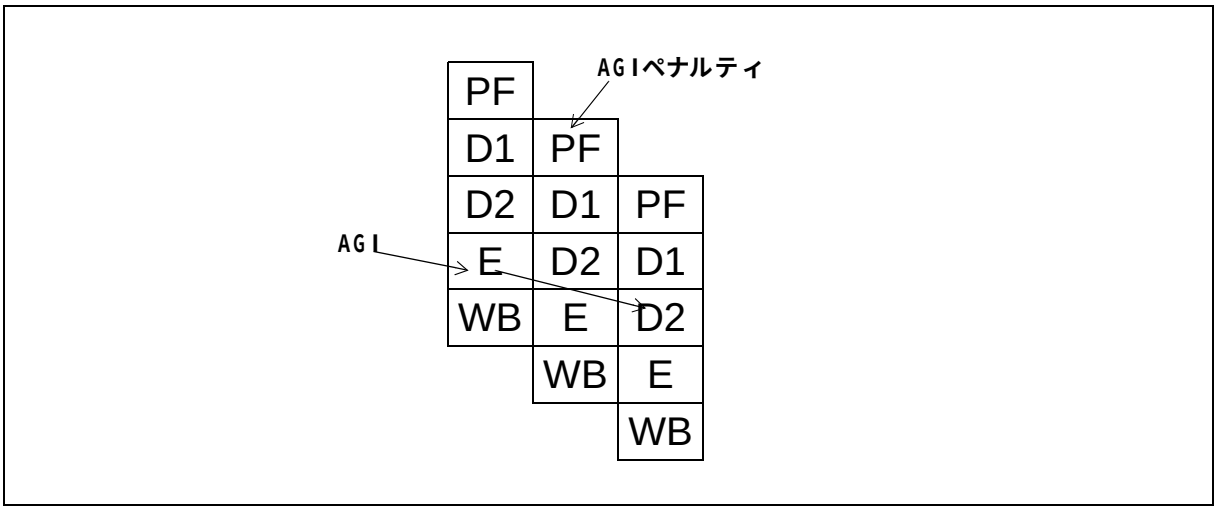


図 3-6. パイプラインにおける AGI ストールの例

一部の命令には、レジスタの暗黙の読み取り / 書き込みを伴うものがあることに注意されたい。ESP を介して暗黙にアドレスを生成する命令 (PUSH、POP、RET、CALL) にも、AGI ペナルティは生じる。

例:

```
sub  esp, 24
                           ; 1 clock cycle stall
push ebx
mov  esp, ebp
                           ; 1 clock cycle stall
pop  ebp
```

PUSH および POP も暗黙に ESP に書き込む。しかし、この場合、次の命令が ESP を介してアドレス指定しても AGI は発生しない。Pentium プロセッサでは、PUSH および POP 命令が使用する ESP を「リネーム」(名前変更)することで、AGI ペナルティを回避している。

例:

```
push edi        ; no stall
mov  ebx, [esp]
```

Pentium プロセッサでは、即値およびディスプレースメント両フィールドがある命令は U パイプでペアリング可能である。定数を使用する必要がある場合は、通常は、最初に定数をレジスタにロードするより即値データを使用する方が効率的である。ただし、同じ即値データを複数回使用する場合は、定数をレジスタにロードし、次にそのレジスタを複数回使用する方が高速である。

例:

```
mov result, 555              ; 555 is immediate, result is displacement
mov word ptr [esp+4], 1      ; 1 is immediate, 4 is displacement
```

MMX 命令には 2 バイトのオペコード (0x0F オペコード・マップ) があるので、ベースまたはインデックスによるアドレス指定でメモリにアクセスする MMX 命令 (ディスプレースメントは 4 バイト) は、すべて長さが 8 バイトになる。7 バイトより長い命令は、デコードが制限される可能性があるので、できるだけ使用しないようにする (3.6.4 項を参照)。即値をベース・レジスタまたはインデックス・レジスタの値に加算して即値フィールドを取り除くことにより、そのような命令のサイズを縮小できることが多い。

Pentium Pro および Pentium II プロセッサでは、パーシャル (8 または 16 ビット) レジスタに書き込んだ直後に (32 ビット) レジスタ全体を使用すると、ストールが生じる。これをパーシャル・ストールという。Pentium プロセッサでは、これに関してストールはない。

例 (Pentium プロセッサ):

```
mov  al, 0                   ; 1
mov  [ebp], eax              ; 2 - No delay on the pentium processor
```

例 (Pentium Pro および Pentium II プロセッサ):

```
mov  al, 0                   ; 1
mov  [ebp], eax              ; 3 PARTIAL REGISTER STALL
```

この読み取りは、パーシャル書き込みがリタイアするまでストールしている。このストールは最低 7 クロック・サイクルに及ぶものと推定される。

最高のパフォーマンスを得るには、大きいレジスタ (たとえば、EAX) の一部であるパーシャル・レジスタ (たとえば、AL、AH、AX) に書き込んだ直後には、大きいレジスタの使用を避ければよい。これにより、Pentium Pro および Pentium II プロセッサのパーシャル・ストール状態を防止することができる。これは、以下に示す小さなレジスタと大きなレジスタのすべての組み合わせにあてはまる。

```
AL  AH  AX  EAX
BL  BH  BX  EBX
CL  CH  CX  ECX
DL  DH  DX  EDX
        SP  ESP
        EP  EBP
        SI  ESI
        DI  EDI
```

パーシャル・レジスタ・ストールに関する詳細については、3.3 節を参照されたい。

3.8.1 AGIに関するパフォーマンス・チューニングの秘訣

3.8.1.1 Pentium® プロセッサ

イベント・パイプラインのアドレス生成インターロックによるストール状態の有無をモニタする。これは、アドレスが使用できないために生じるパイプ・ストールの数であり、アドレスの計算からそのアドレスが使用されるまでに最低1クロック間をおくことにより減少させることができる。

3.8.1.2 Pentium® Pro および Pentium II プロセッサ

これらのプロセッサでは、AGI状態に対してペナルティは生じない。

3.9 命令の長さ

Pentium プロセッサでは、長さが7バイトを超える命令はVパイプでは実行できない。さらに、MMX テクノロジ Pentium プロセッサでは、長さがともに7バイト以下でなければ、2つの命令を命令FIFOにプッシュすることはできない(2.3.1項を参照)。一方の命令がFIFOにプッシュされた場合は、すでにFIFOに命令が最低1つ入っていなければペアリングは行われない。ペアリングの効率がよいコード(MMX コードでよく起こる)では、または誤って予測された分岐の後では、FIFOが空になることがあり、結果として命令の長さが7バイトを超えると必ずペアリングの機会が失われることになる。

さらに、Pentium Pro および Pentium IIプロセッサは、命令が7バイトより長いときには、命令は一度に1つしかデコードすることができない。

したがって、すべてのインテル・プロセッサ上で最高のパフォーマンスを得るには、長さが8バイト未満の単純な命令を使用すればよい。

3.10 整数命令の選択

以下に、最適アセンブリ・コードを得る上で避けるべき命令シーケンスと活用すべき命令シーケンスをいくつか示す。これらは、Pentium プロセッサ、Pentium Pro、および Pentium IIプロセッサにあてはまる。

1. `lea` 命令は、3/4オペランドの加算命令として使用することもできる(たとえば、`lea ecx, [eax+ebx+4+a]`)。
2. 多くの場合、`lea` 命令、または `lea`、`add`、`sub`、および `shift` 命令のシーケンスを定数乗算命令に代えて使用することができる。Pentium Pro および Pentium IIプロセッサでは、定数乗算は他の命令と較べてPentium プロセッサの場合よりも高速であり、したがって2つのオプションの間のトレードオフが早く現れる。Pentium Pro および Pentium IIプロセッサ向けに設計されたコードでは、整数乗算命令を使用するようお勧めする。

3. lea ではそのオペランドが上書きされることがないので、加算の両方のオペランドが加算後も必要な場合でも、レジスタをコピーする必要がない。

lea 命令の欠点は、先行の命令との AGI ストールの可能性を増大させることである。Pentium プロセッサでは, lea は U、V いずれのパイプでも実行できるが、shift は U パイプでしか実行できないので、lea は 2、4、8 のシフトに便利である。Pentium Pro プロセッサでは、lea および shift 命令とも 1 サイクルで実行されるシングル µop 命令である。

複雑な命令

複雑な命令 (たとえば、enter、leave、loop) の使用を避け、単純な命令のシーケンスを使用する。

ショートのゼロ拡張

Pentium プロセッサでは、movzx 命令にプリフィックスがあり、その実行に 3 サイクル、合計で 4 サイクルを要する。これに代えて、以下のシーケンスを使用するようお勧めする。

```
xor eax, eax
mov al, mem
```

ループ内でこのシーケンスが現れた場合は、eax へのただ 1 つの代入が mov al, mem である場合にループから xor を抜き取れる可能性がある。このことは、movzx がペアリング可能でなく、そのために新しいシーケンスが隣接する命令とペアリングされてしまう可能性があるので、Pentium プロセッサにとってはより重要である。

Pentium Pro および Pentium II プロセッサでは、パーシャル・レジスタ・ストールを避けるために、このコード・シーケンスをストールなしに実行させることができる特別なハードウェアがインプリメントされている。にもかかわらず、Pentium Pro および Pentium II プロセッサでは、movzx 命令に優る代替シーケンスはない。パーシャル・ストールの詳細については、3.3 節を参照されたい。

push mem

push mem 命令は、Intel486 プロセッサでは 4 サイクルを要する。これに代えて、以下の命令シーケンスを使用すれば、同じ Intel486 プロセッサでも 2 サイクルしか必要としない。しかも、Pentium プロセッサではペアリングの機会が増える。このシーケンスを使用するようお勧めする。

```
mov reg, mem
push reg
```

ショート・オペコード

できるだけ多く 1 バイト命令を使用する。それにより、コード・サイズが縮小し、命令キャッシュ内の命令密度の増加が促進される。たとえば、add または sub で定数 1 を加算または減算する代わりに、それぞれ inc または dec を使用する。等価なシーケンスに代えて push および pop 命令を使用するのも効果的である。

8/16 ビット・オペランド

8 ビット・オペランドについては、符号拡張およびゼロ拡張されるバイトには 32 ビット演算を使用しないでバイト・オペコードを使用する。オペランド・サイズ・オーバーライドに対するプリフィックスは、16 ビットのオペランドに適用され、8 ビット・オペランドには適用されない。

符号拡張は、非常にコストが高くなりがちである。16 ビット・オペランドをゼロ拡張することにより意味を維持できることが多い。特に、以下の例の C コードは、符号拡張もオペランド・サイズ・オーバーライドのプリフィックスも必要としない。

```
static short int a, b;
if (a==b) {
   . . .
}
```

これらの 16 ビット・オペランドの比較用コードは、次のようなものになる。

```
U パイプ:                          V パイプ:
xor  eax, eax                      xor  ebx, ebx; 1
movw ax, [a]                        ; 2
(prefix) + 1
movw bx, [b]                        ; 4
(prefix) + 1
                                                cmp  eax, ebx; 6
```

もちろん、これが可能なのは特定の状況下においてだけであるが、そのような状況には共通の傾向がある。比較が、より大きい、より小さい、より大きいかまたは等しいなどである場合、あるいは eax または ebx の値を符号拡張を必要とする別の演算に使用しなければならない場合、これはうまくいかない。

Pentium Pro および Pentium Ⅱプロセッサでは、同一レジスタ同士の排他的論理和を取り、レジスタのクリアがそのレジスタの以前の値に依存しないことを認識する機能を備えている。さらに、上記のコード・シーケンスでのパーシャル・ストールを回避する機能も備えている。詳細については 3.9 節を参照されたい。

以下に示すような方法では、Pentium プロセッサ上では処理速度が落ちる可能性がある。

```
movsw    eax, a; 1 prefix + 3
movsw    ebx, b; 5
cmp      ebx, eax; 9
```

Pentium Pro および Pentium Ⅱプロセッサでは、パーシャル・ストールの発生を抑えるために、movzx 命令のパフォーマンス向上が図られている。Pentium Pro および Pentium Ⅱプロセッサ向けのコーディングでは、movzx 命令を使用するとよい。

比較

レジスタの値をゼロと比較するときは、test を使用する。test は、基本的にデスティネーション・レジスタに書き込まないでオペランド間の論理積を取る。同一の値間の論理積演算が行われ、結果がゼロ条件フラグをセットした場合、その値はゼロであったことになる。and は結果レジスタに書き込み、その結果として Pentium Pro または Pentium Ⅱプロセッサでは AGI または人為的な出力依存性が生じるので、and よりも test を使用する方が望ましい。命令サイズが小さい点で、test は cmp ..,0 よりも優れている。

レジスタがEAXの場合、ブール論理積の結果を即値定数と比較して等しいか等しくないかを知りたいときは、test を使用する(if (avar & 8) { })。

Pentium プロセッサでは、形式が eax, imm または reg, reg のときは、test は1サイクルのペアリング可能な命令である。test のその他の形式は、2サイクルを要し、ペアリングすることはできない。

アドレス計算

アドレス計算は、ロード命令およびストア命令のオペランドで行う。内部的には、メモリ参照命令は、4つのオペランド、すなわち再配置可能なロード時定数、即値定数、ベース・レジスタ、およびスケール用のインデックス・レジスタを持つことができる。(セグメント化されたモデルでは、セグメント・レジスタによってリニア・アドレス計算のオペランドが増える場合がある。)多くの場合、メモリ参照用オペランドを十分に使いこなすことにより、使う整数命令の数を減らすことができる。

レジスタのクリア

レジスタにゼロを移動する、すなわちゼロにクリアする望ましいシーケンスは、xor reg, reg である。このシーケンスでは、コード・スペースは節減されるが、条件コードが設定される。条件コードを保持する必要がある場合は、mov reg, 0 を使用する。

整数除算

一般的には、整数除算の前に cdq 命令が実行される(除算命令はEDX:EAX を被除数として使用し、cdq はEDX をセットアップする)。しかし、それよりもEAX をEDX にコピーしてからEDX を31ビット右シフトして符号拡張する方が望ましい。Pentium プロセッサでは、コピー/シフト・シーケンスに要するクロック数は cdq の場合と同じであるが、コピー/シフト方式では他の2つの命令を同時に実行させることができる。値が正であることがわかっている場合は、xor edx, edx を使用する。

Pentium Pro および Pentium Ⅱプロセッサでは、コピー/シフト・シーケンスが2つの命令であるのに対して cdq は1μop 命令なので、cdq の方が高速である。

プロローグ・シーケンス

プロシージャおよび関数のプロローグ・シーケンスでは、esp レジスタによるAGIに注意しなければならない。push は他の push 命令とペアリングすることができるので、関数のエントリでは、呼び出された側がセーブするレジスタのセーブには、これらの命令を使用するようにする。できれば、パラメータをロードしてからESP をデクリメントするとよい。

他のルーチンを呼び出さないルーチン(リーフ・ルーチン)内では、ESP をベース・レジスタとして使用してEBP を解放する。32ビットのフラット・モデルを使用していない場合は、EBP はスタック・セグメントを参照するので、汎用ベース・レジスタとしては使用できないということを忘れてはならない。

即値ゼロとの比較の回避

値をゼロと比較したときには、多くの場合、その値を生成した演算によって条件コードが設定され、それらの条件コードは jcc 命令で直接テストできる。ただし、mov と lea だけは例外で、これらの場合は test を使用する。

エピローグ・シーケンス

現在の関数のスタック・フレームに 4 バイトしか割り当てられていない場合は、スタック・ポインタを 4 インクリメントする代わりに pop 命令を使用する。これにより、AGI を避けることができる。Pentium プロセッサの場合は、pop を 2 つ使用して 8 バイトのスタック・フレームに対応する。

intel

4

MMX® テクノロジ ・コード開発のガイドライン

第 4 章
MMX® テクノロジ ・コード開発のガイドライン

第 3 章のガイドラインに加えて、本章に示すガイドラインも重要である。本章と第 3 章のガイドラインに従えば、インテル MMX テクノロジ搭載のすべてのプロセッサに適応する、高速で効率的な MMX テクノロジ ・コードを開発することができるはずである。

4.1 規則および提言

以降に、規則と提言の一覧を示す。

4.1.1 規則

- MMX 命令と浮動小数点命令を混用してはならない。4.2.3 項を参照。
- 同一メモリ領域での大きいストアの後の小さいロードと、小さいストアの後の大きいロードを避ける。同一メモリ領域でのロードとストアに対しては、データのサイズを揃え、アドレスのアライメントを合わせる。4.5.1 項を参照。
- `OP reg, mem` フォーマットを使用して、命令数の低減、またはレジスタにかかる負荷の軽減を図る。ただし、ロードを多用しすぎてパフォーマンスを損なわないよう注意する。4.4.1 項を参照。
- 浮動小数点コードに遷移することがわかっているすべての MMX 命令セクションの終わりに EMMS を使用する。4.2.4 項を参照。
- MMX テクノロジ ・レジスタへのキャッシュ ・データ ・バンド幅を最適化する。4.5.2 節を参照。

4.2 計画に際しての留意点

既存のアプリケーションを改造するにしても、新しいアプリケーションを作成するにしても、MMX 命令を使って最高のパフォーマンスを得るには、留意すべき問題がいくつかある。一般には、実行時間の長いコード ・セグメント、整数命令によるインプリメントが可能なコード ・セグメント、キャッシュ ・アーキテクチャの効率的利用が可能なコード ・セグメントを探すことが大切である。インテルのパフォーマンス ・ツール ・セットには、この評価およびチューニングに役立つツールが用意されている。

インプリメンテーションを開始する前に、下記の点を明確にしておく必要がある。

- MMX テクノロジの恩恵を受けることができるのは、コードのどの部分か。
- 現在のアルゴリズムが MMX テクノロジにとって最適か。
- このコードは整数または浮動小数点のどちらか。

- データをどのように配置したらよいか。
- データは 8、16、32 ビットのいずれか。
- アプリケーションはインテル MMX テクノロジ搭載のプロセッサおよび従来のプロセッサの両方で実行させる必要があるか。CPUID を使用して、スケーラブルなインプリメントを実現することができるか。

4.2.1 MMX® テクノロジの恩恵を受けることができるのは、コードのどの部分か

どのコードを変換するかを決定する。

多くのアプリケーションには、非常に実行時間の長いコード・セクションがある。その代表格が、音声圧縮アルゴリズムおよびフィルタ、動画表示ルーチン、およびレンダリング・ルーチンである。これらのルーチンは、一般的に 8 または 16 ビット整数を操作する小さい繰り返しループであり、アプリケーション実行時間のかなりの部分を占める。これらのルーチンを MMX テクノロジ最適化コードに置き換えると、パフォーマンス上最大の効果が得られる。これらのループを MMX テクノロジ最適化ライブラリに取り込むことにより、インテル MMX テクノロジ搭載の有無に関わらず、どのプラットフォームにも柔軟に対応することができる。

インテルの VTune ビジュアル・チューニング・ツールなどのパフォーマンス最適化ツールを使用すると、実行時間の長いコード・セクションを確認することができる。それらのコード・セグメントを確認したら、現在のアルゴリズムと修正したアルゴリズムのどちらがより高いパフォーマンスを発揮するかを判断する必要がある。場合によっては、アルゴリズムの操作タイプを変更することによりパフォーマンスを向上させることができる。アルゴリズムを MMX 命令の能力に適合させることが、最高のパフォーマンスを引き出す秘訣である。

4.2.2 浮動小数点か整数か？

アルゴリズムに含まれるデータが浮動小数点データか整数データかを確認する。

整数データでインプリメントされているアルゴリズムでは、マイクロプロセッサ・クロック・サイクル数を最も多く使用するアルゴリズムの部分を確認すればよい。それを確認できたら、それらのコード・セクションを MMX 命令に置き換えてインプリメントする。

浮動小数点データでインプリメントされているアルゴリズムでは、浮動小数点が使用されている理由を考えてみる。浮動小数点演算を使用する理由として挙げられるのが、パフォーマンス、数値の範囲、精度である。そして、アルゴリズムを浮動小数点でインプリメントしている理由がパフォーマンスであった場合、そのアルゴリズムは、MMX 整数コードへの置き換でパフォーマンスを向上させることのできる有力候補である。

範囲または精度上の理由で浮動小数点を使用しているアルゴリズムについては、さらなる調査が必要である。それは「データ値を整数に変換しても、必要な範囲と精度を維持できるか」ということである。維持できない場合、そのコードは浮動小数点コードのままにしておくのが最良である。

4.2.3 浮動小数点および MMX® テクノロジ両コード混用のアプリケーション

MMX コードを生成する際には、8 つの MMX レジスタが浮動小数点レジスタで別名化されていることを念頭に置く必要がある。MMX 命令から浮動小数点命令への切り換えには最高で 50 クロック・サイクルかかることもあるので、このような命令タイプの切り替えは極力抑えた方がよい。命令レベルでは、MMX コードと浮動小数点コードを混用してはならない。浮動小数点命令と MMX 命令を頻繁に切り替えざるを得ないアプリケーションでは、その切り替えによるペナルティを最小限に抑えるため、アプリケーションが MMX 命令ストリームまたは浮動小数点ストリームにとどまっている時間を極力長くするように留意する。

浮動小数点および MMX 命令の両タイプの命令を使用するアプリケーションを作成するときは、下記のガイドラインに従って両タイプの命令実行を切り離すとよい。

- MMX 命令ストリームと浮動小数点命令ストリームを、それぞれ一方のタイプだけの命令からなる別々の命令ストリームに区分する。
- 切り替えの過程でレジスタの内容に依存しない。
- 浮動小数点コードへの切り替えがあることが確実なときは、EMMS 命令を使用して、浮動小数点タグ・ワードが空の MMX コード・セクションをそのままにしておく。
- スタックが空の浮動小数点コード・セクションはそのままにしておく。
- 以下の例を参照。

```
FP_code:
     ...
     ...                /* leave the floating-point stack empty */
MMX_code:
     ...
     EMMS               /* empty the MMX registers */
FP_code1:
     ...
     ...                /* leave the floating-point stack empty */
```

この浮動小数点プログラミング・モデルの詳細については、『Pentium® ファミリー・デベロッパーズ・マニュアル、下巻：アーキテクチャとプログラミング』（資料番号 241430J）を参照されたい。

4.2.4 EMMS に関するガイドライン

浮動小数点コードへの切り替えがあることが確実なときは、MMX コードの終わりで EMMS 命令を呼び出す。

MMX レジスタは浮動小数点レジスタで別名化されているので、浮動小数点命令を発行する前に MMX レジスタをクリアすることが非常に重要である。EMMS 命令で MMX レジスタをクリアし、浮動小数点タグ・ワードの値を空（つまり、オール・ワンズ）に設定する。浮動小数点命令の実行時に浮動小数点スタックのオーバーフロー例外を避けるために、EMMS 命令はすべての MMX コード・セグメントの終わりに挿入する必要がある。

4.2.5 MMX® テクノロジ搭載の有無を検知するCPUID の使用法

アプリケーションがMMX テクノロジを使用しているかどうかを判定する。

MMX テクノロジは、2 つの方法でアプリケーションに取り込むことができる。その 1 つは、インストール時にMMX テクノロジの有無をアプリケーションでチェックするという方法である。インテルMMX テクノロジが使用されている場合は、それに該当するDLL がインストールされる。もう1つは、プログラム実行中にチェックし、必要なDLL を動的にインストールするという方法である。これは、さまざまなプラットフォーム上で動作することが要求されるプログラムに対して効果的である。

プロセッサがインテル MMX テクノロジを搭載しているかどうかを知るには、アプリケーション側でインテル・アーキテクチャ機能フラグを調べる。CPUID 命令は機能フラグの値をEDX レジスタに返す。その値に基づいて、プログラムはシステムに対してどのバージョンのコードが適切であるかを判断する。

MMX テクノロジのサポートの有無は、機能フラグのビット23 の値で判断できる。このビットが 1 にセットされている場合は、プロセッサはMMX テクノロジを搭載している。以下に示すコード・セグメントは機能フラグの値をEDX にロードし、MMX テクノロジの有無を調べるというものである。CPUID 命令の使用法の詳細については、アプリケーション・ノートAP-485 『Intel Processor Identification with CPUID Instruction』(資料番号 241618) を参照されたい。

```
…                   identify exstence of CPUID instruction
…                   ;
…                   ; identify Intel Processor
…                   ;
mov EAX, 1          ; request for feature flags
CPUID               ; 0Fh, 0A2h, CPUID Instruction
test EDX, 00800000h ; is MMX technology bit(bit 23)in feature
                    ; flag equal to 1
jnz  Found
```

4.2.6 データのアライメント

データのアライメントを合わせる。

多くのコンパイラでは、コントロールを使用して変数のアライメントを指定することができる。これによって、希望する境界に変数のアライメントを合わせることができる。しかし、ときには、変数のアライメントが指定の境界に合っていないこともある。このようなアライメントのずれが見つかった場合は、以下に示すC アルゴリズムにより、変数のアライメントを合わせる。このアルゴリズムでは、64 ビットの変数のアライメントを64 ビット境界に合わせる。アライメントを合わせると、データがキャッシュ・ライン境界にまたがっていたときに較べて、この変数へのアクセスのたびに、Pentium プロセッサでは 3 クロック・サイクルが節減され、Pentium Pro およびPentium IIプロセッサでは6～9 クロック・サイクルが節減される。

```
double a[5];
double *p, *newp;

p = (double*)malloc ((sizeof(double)*5)+4)
newp = (p+4) & (-7)
```

データ・アライメントを改善するもう1つの方法は、64 ビットのアライメント境界に合わせてデータをコピーするというものである。頻繁にアクセスされるデータについては、これにより、大きなパフォーマンスの向上がもたらされる可能性がある。

4.2.6.1 スタックのアライメント

規則上の問題として、コンパイラは静的でないものは何でもスタック上に割り当てるので、スタックにストアされているデータを 64 ビット単位で扱うと便利な場合がある。これが必要な場合は、スタックのアライメントを合わせることが大切である。以下に示す関数のプロローグとエピローグ内のコードは、スタックのアライメントが合っているかどうかを確認するためのものである。

```
Prplogue:
     push      ebp                ; save old frame ptr
     mov       ebp, eps           ; make new frame ptr
     sub       ebp, 4             ; make room of stack ptr
     and       ebp, 0FFFFFFF8     ; align to 64 bits
     mov       [ebp],esp          ; save old stack ptr
     mov       esp, ebp           ; copy aligned ptr
     sub       esp, FRAMESIZE     ; allocate space
     … callee saves, etc

epilogue:
     … callee restores, etc
     mov       esp, [ebp]
     pop       ebp
     ret
```

アクセス頻度の高いデータのアライメントずれが避けられない場合は、そのデータをアライメントが合った一時記憶域にコピーすると効果的な場合がある。

4.2.7 データの配置

MMX テクノロジでは、SIMD 方式により多くのマルチメディア・アルゴリズムに本来備わっている並行性を活用している。MMX コードから最高のパフォーマンスを引き出すには、以下のガイドラインに従ってメモリ内のデータをフォーマットする必要がある。

ここで、配列のすべての 16 ビット要素に 16 ビットのバイアスを加算するという、単純な例を基に考えてみる。正規のスカラー・コードでは、ループの初めでバイアスをレジスタにロードし、別のレジスタ内の配列要素にアクセスし、一度に 1 つずつ加算を行う。

MMX では、MOVQ 命令により配列要素を一度に 4 つずつ処理でき、PADDW 命令により一度に 4 回ずつ加算を行えるので、このルーチンを MMX コードに変換すると、4 倍のスピードアップを期待できる。ただし、実際にそのスピードアップを達成するには、加算を行う際に MMX レジスタにバイアスを 4 回続けてコピーする必要がある。

オリジナルのスカラー・コードでは、メモリ内にはバイアスのコピーは 1 つしかない。MMX 命令を使用するには、各種の操作により MMX レジスタにバイアスの 4 つのコピーを設定すればよい。あるいは、バイアスの 4 つの連続コピーを保持するようにあらかじめメモリをフォーマットしておくのもよい。そうすれば、ループの前に 1 つの MOVQ 命令を使用してこれらのコピーをロードしておくだけで、4 倍のスピードアップが達成される。このタイプのデータ配置について、もう 1 つの興味深い例を 4.6 節に紹介する。

さらに、SIMD 操作で SIMD データにアクセスするときは、データへのアクセスは単に宣言の変更によって改善することができる。たとえば、スペース内の 1 点を表す構造体の宣言について考えてみる。構造体は、3 つの 16 ビット値とパディング用の 1 つの 16 ビット値からなっている。以下に宣言例を示す。

```
typedef struct { short x,y,z; short junk; } Point;
Point pt[N];
```

以下のコードでは、第 2 次元の y にスケーリング値をかける必要がある。ここでは、for ループで配列 pt 内の各 y 次元にアクセスする。

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

アクセスされるのは連続データではない。それが連続データであれば、キャッシュ・ミスが何回も発生し、アプリケーションのパフォーマンスを低下させることになりかねない。

しかし、データを以下のように宣言した場合は、スケーリング操作を配列化することができる。

```
short ptx[N], pty[N], ptz[N];
for (i=0; i<N; i++) pty *= scale;
```

MMX テクノロジの出現によって、データ編成の選択がより重要になり、この選択はそのデータに対して行われる操作に基づいて注意深く行う必要がある。アプリケーションによっては、従来方式のデータ配置では最高のパフォーマンスが得られない場合がある。

MMX テクノロジの新しい 64 ビット・パックド・データ型では、アライメントに合わないデータ・アクセスの可能性がより高くなる。MMX 命令と他のパックド・データ型を使用すれば、どのアルゴリズムでもアライメントが合わないデータ・アクセスが発生するのは当然である。これを示す単純な例は、FIR フィルタである。FIR フィルタは、実際上は係数タップ数の長さでのベクトル・ドット積である。データ要素 i のフィルタ操作がデータ要素 j から始まるベクトル・ドット積 (`data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1]`) であるとすれば、データ要素 `i+1` のフィルタ操作はデータ要素 `j+1` から始まる。

メモリ内の 64 ビット・データのアライメントについては、4.2.6 項を参照されたい。64 ビット境界にアライメントが合ったデータ配列と 64 ビット境界にアライメントが合った係数の配列があると想定すると、最初のデータ要素に対するフィルタ操作は完全にアライメントが合う。しかし、2 番目のデータ要素に対するフィルタ操作については、データ配列への各アクセスのアライメントがずれてくる。データ構造体の複製とパディングにより、本来アライメントがずれるアルゴリズムでのデータ・アクセスの問題を避けることができる。FIR フィルタでのアライメントずれの問題を避ける方法については、アプリケーション・ノート AP-559 『MMX Instructions to Compute a 16-Bit Real FIR Filter』(資料番号 243044) の例を参照されたい。

注記

複製およびパディングでは、データ・サイズの増大という代償を払ってアライメントずれの問題を克服し、アライメントが合わないデータ・アクセスの大きなペナルティを避けることができる。コードを開発するときは、このトレードオフに留意し、最高のパフォーマンスが得られるオプションを使用することが望ましい。

4.2.8 開発最終段階におけるアプリケーションのチューニング

アプリケーションが正しく動作した段階で、すなわち開発の最終段階でアプリケーションをチューニングする最善の方法は、アプリケーションをシステム上で動作させながらプロファイラを使用して計測することである。インテルのビジュアル・チューニング・ツール VTune はまさにそのためのツールであり、パフォーマンス向上のためにアプリケーションのどこを変更したらよいかを的確に知ることができる。さらに、インテルのプロセッサには、パフォーマンス・カウンタが組み込まれている。これらのカウンタの説明および使用法については、7.1 節を参照されたい。

4.3 スケジューリング

以下の項では、命令のスケジューリングについて説明する。

4.3.1 MMX® 命令ペアリングに関するガイドライン

本項では、MMX 命令同士のペアリングおよび MMX 命令と整数命令のペアリングに関するガイドラインを示す。命令のペアリングは、Pentium プロセッサのパフォーマンスを大幅に向上させ、Pentium Pro および Pentium II プロセッサにも無害であるだけでなく、場合によってはそれら両プロセッサのパフォーマンス向上にも寄与する。

4.3.1.1 2 つの MMX® 命令のペアリング

以下に、2 つの MMX 命令のペアリングに関する規則を示す。

- ともに MMX シフタ・ユニットを使用する 2 つの MMX 命令 (パック、アンパック、シフト命令) は、MMX シフタ・ユニットが 1 つしかないのでペアリングはできない。シフト操作は、U、V どちらのパイプにも発行できるが、2 つのシフト操作を同一クロック・サイクルで発行することはできない。
- ともに MMX 乗算器ユニットを使用する 2 つの MMX 命令 (pmull、pmulh、pmadd タイプの命令) は、MMX 乗算器ユニットが 1 つしかないのでペアリングはできない。乗算は、U、V どちらのパイプにも発行できるが、2 つの乗算を同一クロック・サイクルで発行することはできない。
- メモリ、整数レジスタ・ファイルのどちらかにアクセスする MMX 命令は、U パイプだけに発行することができる。これらの命令は、待機してから、次の命令ペアとして (U パイプに) 発行されるので、V パイプ向けにスケジュールしてはならない。
- U パイプ命令の MMX デスティネーション・レジスタは、V パイプ命令のソース、デスティネーションのどちらのレジスタとも同じであってはならない (依存性チェック)。
- EMMS 命令は、ペアリングすることはできない。
- CR0.TS ビットまたは CR0.EM ビットがセットされている場合、MMX 命令は V パイプに進入することはできない。

4.3.1.2 U パイプ内の整数命令と V パイプ内の MMX® 命令のペアリング

以下に、U パイプ内の整数命令と V パイプ内の MMX 命令のペアリングに関する規則を示す。

- MMX 命令は、浮動小数点命令の後の最初の MMX 命令であってはならない。
- V パイプ内の MMX 命令は、メモリにも整数レジスタ・ファイルにもアクセスしてはならない。
- U パイプ内の整数命令は、ペアリング可能な U パイプ整数命令でなければならない (表 3-1 を参照)。

4.3.1.3 U パイプ内の MMX® 命令と V パイプ内の整数命令のペアリング

以下に、U パイプ内の MMX 命令と V パイプ内の整数命令のペアリングに関する規則を示す。

- V パイプ内の整数命令は、ペアリング可能な V パイプ整数命令でなければならない (表 3-1 を参照)。
- U パイプ内の MMX 命令は、メモリにも整数レジスタ・ファイルにもアクセスしてはならない。

4.3.1.4 スケジューリングに関する規則

MMX 命令は、Pentium、Pentium Pro、および Pentium II プロセッサで、乗算命令を含めてすべてパイプライン化が可能である。MMX 乗算命令を除いて、すべての命令は 1 クロックで実行できる。MMX 乗算命令は 3 クロックを要する。

乗算命令は実行に 3 クロックを要するので、乗算命令の結果は 3 クロック後に発行された他の命令によってだけ使用することができる。このため、乗算の後の 2 つの命令ペアに乗算結果に依存する命令が関わることのないようにスケジューリングしなければならない。

2.3.1 項で説明したように、レジスタに書き込んだ後のそのレジスタのストアには、レジスタの更新に 2 クロック待たなければならない。したがって、レジスタ更新の 2 クロック・サイクル後にストアをスケジュールすれば、パイプラインのストールを避けることができる。

4.4 命令の選択

以下の項では、命令選択の最適化について説明する。

4.4.1 メモリにアクセスする命令の使用

MMX 命令は、2 つのレジスタ・オペランド (`OP reg, reg`)、または 1 つのレジスタ・オペランドと 1 つのメモリ・オペランド (`OP reg, mem`) を取ることができる。ここで、`OP` は命令のオペランドを表し、`reg` はレジスタ、`mem` はメモリをそれぞれ表す。`OP reg, mem` という形式の命令は、特定のケースではレジスタにかかる負荷を軽減し、1 サイクル当りの操作数を増やし、さらにコード・サイズを縮小するのに効果的である。

以降の説明では、メモリ・オペランドはデータ・キャッシュ内に存在するものと想定している。データ・キャッシュ内に存在しない場合、結果として生じるペナルティは、この項で説明するスケジューリングの効果が無駄になるほど大きなものになる。

Pentium プロセッサでは、OP reg, mem MMX 命令のレイテンシは (キャッシュ・ヒットを想定して) OP reg, reg 命令のそれよりも長くない。ただし、OP reg, reg 命令の方がペアリングの機会が少なくなる (4.3.1 項を参照)。Pentium Pro および Pentium II プロセッサでは、OP reg, mem MMX 命令は 2 µop に変換される。それに対して、OP reg, reg 命令は 1 µop に変換される。したがって、OP reg,mem MMX 命令は OP reg, reg 命令よりもデコード・バンド幅を狭め (2.2 節を参照)、多くのリソースを占有する傾向がある。

"OP reg, mem" 命令をどのようなときに使用すべきかは、MMX コードがメモリ拘束型であるか (つまり、実行速度がメモリ・アクセスに依存するか) どうかによって異なる。経験則からいうと、MMX コード・セクションは、以下の不等式が成立する場合にメモリ拘束型であると考えられる。

(命令数)/2 < (メモリ・アクセス) + (非 MMX 命令数)/2

メモリ拘束型の MMX コードについては、同一のメモリ・アドレスが 2 回以上使用される場合は常に複数のロードを併合するようお勧めする。こうすることにより、メモリ・アクセス回数が減少する。

例:

```
OP            MM0, [address A]
OP            MM1, [address A]
```

これは以下のコードになる。

```
MOVQ     MM0, [address A]
OP            MM0, MM2
OP            MM1, MM2
```

メモリ拘束型でないコードについては、同一のメモリ・アドレスが 3 回以上使用される場合だけロードの併合をお勧めする。ロードの併合が不可能な場合は、"OP reg, mem" 命令を使用して命令数を減らし、コード・サイズを縮小するようお勧めする。

例:

```
MOVQ     MM0, [address A]
OP            MM1, MM0
```

これは以下のコードになる。

```
OP            MM1, [address A]
```

多くの場合、MOVQ reg, reg に代えて MOVQ reg, mem を、OP reg, mem に代えて OP reg, reg を使用することができる。これにより、Pentium Pro および Pentium II プロセッサでは 1 µop を節減するので、可能なかぎり使用されたい。

例 (ただしOPは対称型操作):

```
MOVQ     MM1, MM0                  (1 micro-op)
OP            MM1, [address A]     (2 micro-ops)
```

これは以下のコードになる。

```
MOVQ     MM1, [address A]          (1 micro-op)
OP            MM1, MM0             (1 micro-op)
```

45 メモリの最適化

本節では、メモリ・アクセスの向上について説明する。

451 パーシャル・メモリ・アクセス

MMX レジスタでは、プロセッサをストールさせることなく、より大量のデータを移動させることができる。8、16、または 32 ビット長の単一の配列値をロードする代わりに、単一のクワッドワードの値をロードし、次に構造体または配列のポインタをインクリメントすればよい。

MMX 命令で操作されるデータはすべて、以下のどちらかを使用してロードする。

- 64 ビット・オペランドをロードするMMX 命令 (MOVQ MM0, m64 など)。
- クワッドワード・メモリ・オペランドを操作対象とするレジスタ - メモリ形式の任意のMMX 命令 (PMADDW MM0, m64 など)。

SIMD データはすべて、64 ビット・オペランドをストアするMMX 命令 (MOVQ m64, MM0 など) でストアする。

これらの提言の目的は 2 つある。第一は、より大きいクワッド・データ・ブロック・サイズを使用することにより、SIMD データのロードとストアの効率化を図ることである。第二は、8、16、または 32 ビットのロードおよびストア操作と、同じ SIMD データに対する 64 ビットの MMX ロードおよびストア操作との混用を回避することである。これにより、同一メモリ領域で大きいストアの後に小さいロードが行われたり、小さいストアの後に大きいロードが行われたりするような状況を防止することができる。Pentium Pro および Pentium II プロセッサは、このような状況ではストールする。

以下の例について検討されたい。第 1 のケースでは、(メモリ・アドレス mem から始まる) 同一メモリ領域で一連の小さいストアの後に大きいロードが行われる。この場合、大きいロードはストールする。

```
MOV         mem, eax          ; store dword to address “mem”
MOV         mem + 4, ebx      ; store dword to address “mem +4”
        :
        :
MOVQ     mm0, mem             ; load qword at address “mem”, stalls
```

MOVQ は、一連のストアのメモリ書き込みが終わるのを待ってから、必要なすべてのデータにアクセスしなければならない。このストールは、他のデータ型でも (たとえば、バイトまたはワードがストアされ、次に同一メモリ領域からワードまたはダブルワードが読み取られるときにも) 発生する可能性がある。上記のコード・シーケンスを以下に示すように変更すると、プロセッサは遅延なしにデータにアクセスすることができる。

```
MOVD      mm1, ebx          ; build data into a qword first before
                            ; storing it to memory
MOVD      mm2, eax
PSLLQmm1, 32
POR       mm1, mm2
MOVQ      mem, mm1          ; store SIMD variable to “mem” as a qword
       :
       :
MOVQ      mm0, mem          ; load qword SIMD variable “mem”, no stall
```

第 2 のケースでは、(メモリ・アドレス mem から始まる)同一メモリ領域で大きいストアの後に一連の小さいロードが行われる。この場合、小さいロードはストールする。

```
MOVQ      mem, mm0          ; store qword to address “mem”
       :
       :
MOV       bx, mem + 2       ; load word at address “mem + 2” stalls
MOV       cx, mem + 4       ; load word at address “mem + 4” stalls
```

一連のワード・ロードは、クワッドワード・ストアのメモリ書き込みが終わるのを待ってから、必要なデータにアクセスしなければならない。このストールは、他のデータ型でも (たとえば、ダブルワードまたはワードがストアされ、次に同一メモリ領域からワードまたはバイトが読み取られるときにも) 発生する可能性がある。上記のコード・シーケンスを以下に示すように変更すると、プロセッサは遅延なしにデータにアクセスすることができる。

```
MOVQ      mem, mm0          ; store qword to address “mem”
       :
       :
MOVQ      mm1, mem          ; load qword at address “mem”
MOVD      eax, mm1          ; transfer “mem +2” to ax from
                            ; MMX register not memory
PSRLQmm1, 32
SHR       eax, 16
MOVD      ebx, mm1          ; transfer “mem + 4 to bx from
                            ; MMX register, not memory
AND  ebx, 0ffffh
```

一般的に、これらの変換では、求められる操作の実行に必要な命令の数が増大する。Pentium Pro および Pentium II プロセッサでは、命令数の増大によるパフォーマンス上のペナルティが利点を上回ることはない。しかし、Pentium プロセッサの場合は、上記のコードの変換が有利に働かないので、命令数の増大はパフォーマンスにとって逆効果になる場合がある。このため、これらの変換は注意深く、効率的にコーディングして、Pentium プロセッサのパフォーマンスへの負の影響を最小限に抑える必要がある。

452 メモリ・フィルおよびビデオ・フィルのバンド幅の増大

メモリがどのようにアクセスされ、フィルされるかを理解することは有益である。メモリからメモリへのフィル(メモリからビデオへのフィルなど)は、メモリからの32バイト(キャッシュ・ライン)ロードと、その直後の(ビデオ・フレーム・バッファなどの)メモリへのストア・バックとして定義されている。以降の各項に、バンド幅を広げ、順次メモリ・フィル(ビデオ・フィル)のレイテンシを小さくするためのガイドラインを示す。これらのガイドラインは、インテルMMXテクノロジを搭載しているすべてのインテル・アーキテクチャ・プロセッサを対象とし、ロードとストアが第2レベル(L2)のキャッシュでヒットしない場合に該当する。

4521 MOVQ命令によるメモリ・バンド幅の拡大

値をロードすると、1キャッシュ・ライン全体がオンチップ・キャッシュにロードされる。しかし、32ビット・ストア(MOVDなど)を使用せず、MOVQを使用して再びメモリにデータをストアすると、1回のメモリ・フィル当たりのストア回数が半分に減る。結果として、メモリ・フィル・サイクルのバンド幅は大幅に拡大する。一部のPentiumプロセッサ・ベースのシステムでは、32ビット・ストアに代えて64ビット・ストアを使用したところ、30%のバンド幅の拡大が得られたという計測結果も報告されている。一方、Pentium ProおよびPentium IIプロセッサでは、ロードとストアの両方をMOVQ命令で実行すると、バンド幅の拡大によってパーシャル(部分的)メモリ・アクセスが回避される。

読み取りと書き込みを混用すると、一連の読み取りを行ってからデータを書き出すよりも速度が低下する。たとえば、メモリ内のデータを移動する場合は、1つの読み取り命令と1つの書き込み命令を発行するよりも、メモリから数ラインをキャッシュに読み込んでから、それらのラインを再び新しいメモリ位置に書き出す方が高速になる。

4522 同一DRAMページのロードおよびストアによるメモリ・バンド幅の拡大

DRAMはページ単位に分割されているが、この分割はオペレーティング・システム(OS)のページと同じではない。DRAMのページ・サイズは、DRAMの合計サイズとDRAMの編成によって変わる。数Kバイトのページ・サイズが一般的である。OSページと同様に、DRAMページは順次アドレス形式で編成されている。同一のDRAMページへの順次アクセスの方が、異なるDRAMページへの順次アクセスよりもレイテンシは短い。多くのシステムでは、メモリ・ページ・ミス(すなわち、直前にアクセスされたページでなく、他のページへのアクセス)のレイテンシは、メモリ・ページ・ヒット(直前のアクセスと同じページへのアクセス)のレイテンシの2倍大きくなる可能性がある。したがって、メモリ・フィル・サイクルのロードとストアが同じDRAMページを使用すると、メモリ・フィル・サイクルのバンド幅は大幅に拡大する。

4523 アライメントに合ったストアによるメモリ・フィル・バンド幅の拡大

アライメントに合わないストアでは、メモリのストア回数が倍増する。クワッドワード・ストアはアライメントを8バイト境界に合わせるよう強くお勧めする。1キャッシュ・ラインをメモリに書き込むには、アライメントに合ったクワッドワード・ストアが4回必要である。クワッドワード・ストアのアライメントが8バイト境界に合っていない場合は、各MOVQストア命令で32ビットの書き込みが2回行われる。一部のシステムでは、アライメントが合ったストアに代えて64ビットのアライメントが合わないストアを使用したところ、バンド幅が20%縮小したという計測結果も報告されている。

4.5.2.4 64 ビット・ストアによるビデオへのバンド幅の拡大

プロセッサとフレーム・バッファを結ぶ PCI バスは 32 ビット幅であるが、大部分の Pentium プロセッサ・ベースのシステムでは、MOVQ を使用してビデオにストアする方が 32 ビット・ストアを 2 回使用するよりも高速である。その理由は、クワッドワード・ストアを使用したときは、(プロセッサと PCI バスの間にある) PCI ライト・バッファへのバンド幅が拡大するためである。

4.5.2.5 アライメントに合ったストアによるビデオへのバンド幅の拡大

アライメントに合わないストアが現れると、ビデオへのバンド幅が一気に狭くなる。アライメントずれがあると、ストア回数が倍増し、(フレーム・バッファへの) PCI バス上のストアのレイテンシがずっと長くなる。PCI バス上では、アライメントが合っていない一連のストアをバースト処理することはできない。Pentium プロセッサ搭載のシステムでは、アライメントが合ったストアの場合と比較すると、アライメントが合わないストアではビデオ・フィル・バンド幅は平均で 80% 狭くなる。

4.6 コーディング手法

本節では、アプリケーションのコーディングに着手する際に役立つ単純な例をいくつか示す。目的は、よく使用される単純な、低水準の操作を提示することにある。各例では、Pentium、Pentium Pro、および Pentium II プロセッサで最高のパフォーマンスを得るために必要な最少数の命令を使用している。

各例の構成は、以下のようになっている。

- 簡潔な説明
- サンプル・コード
- 注記 (必要に応じて)

これらの例は、これらよりも長いコード・シーケンスの一部として取り入れられることを想定しているため、スケジューリングについては言及していない。

4.6.1 符号なしアンパック

インテルＭＭＸ テクノロジには、ＭＭＸ レジスタ内のデータをパックおよびアンパックするための命令がいくつか用意されている。アンパック命令は、符号なし数値をゼロ拡張する目的に使用することができる。以下の例では、ソースはパックド・ワード(16 ビット)データ型であると想定している。

入力： ＭＭ0: ソース値

ＭＭ7:0 (必要に応じて、レジスタＭＭ7 の代わりにローカル変数を使用することができる。)

出力： ＭＭ0:2 つの LOW エンド・ワードからの 2 つのゼロ拡張された 32 ビット・ダブルワード

ＭＭ1:2 つのＨＩＧＨ エンド・ワードからの 2 つのゼロ拡張された 32 ビット・ダブルワード

```
MOVQ      MM1, MM0  ; copy source
PUNPCKLWD MM0, MM7  ; unpack the 2 low end words
                    ; into two 32pbit doubleword
PUNPCKHWD MM1, MM7  ; unpack the 2 high end words into two
                    ; 32-bit doubleword
```

4.6.2 符号付きアンパック

符号付き数値は、値がアンパックされると符号拡張される。これは、上に示したゼロ拡張とは別の形で行われる。以下の例では、ソースはパックド・ワード(16 ビット)データ型であると想定している。

入力： ＭＭ0: ソース値

出力： ＭＭ0:2 つの LOW エンド・ワードからの 2 つの符号拡張された 32 ビット・ダブルワード

ＭＭ1:2 つのＨＩＧＨ エンド・ワードからの 2 つの符号拡張された 32 ビット・ダブルワード

```
PUNPCKHWD MM1, MM0   ; unpack the 2 high end words of the
                    ; source into the second and fourth
                    ; words of the destination
PUNPCKLWD MM0, MM0   ; unpack the 2 low end words of the
                    ; source into the second and fourth
                    ; words of the destination
PSRAD     MM0, 16    ; Sign-extend the 2 low end words of
                    ; the source into two 32-bit signed
                    ; double words
PSRAD     MM1, 16    ; Sign-extend the 2 high end words of
                    ; the source into two 32-bit signed
                    ; doublewords
```

4.6.3 飽和ありインターリーブ型パック

パック命令は、2 つの値をあらかじめ決められた順序でデスティネーション・レジスタにパックする。特に、PACKSSDW 命令は、以下の図に示すように、ソース・オペランドからの 2 つの符号付きダブルワードとデスティネーション・オペランドからの 2 つの符号付きダブルワードを、4 つの符号付きワードとしてデスティネーション・レジスタにパックする。

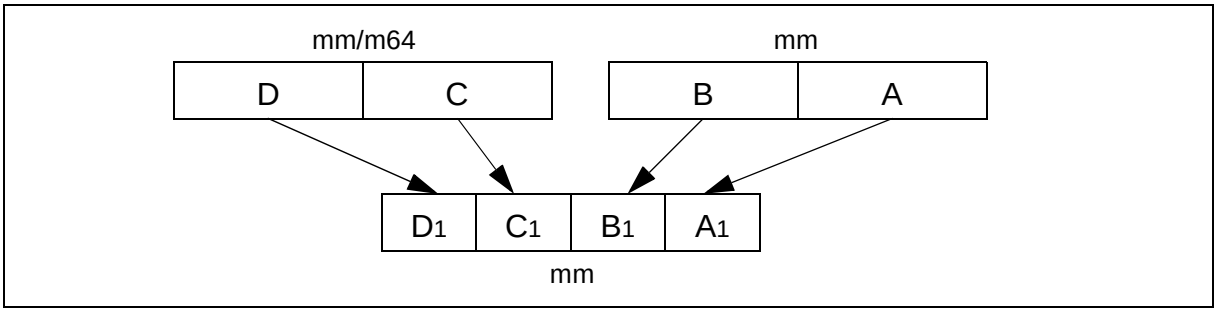


図 4-1. PACKSSDW mm,mm/mm64 命令の例

以下の例では、図に示すように、2 つの値をデスティネーション・レジスタ内でインターリーブする。

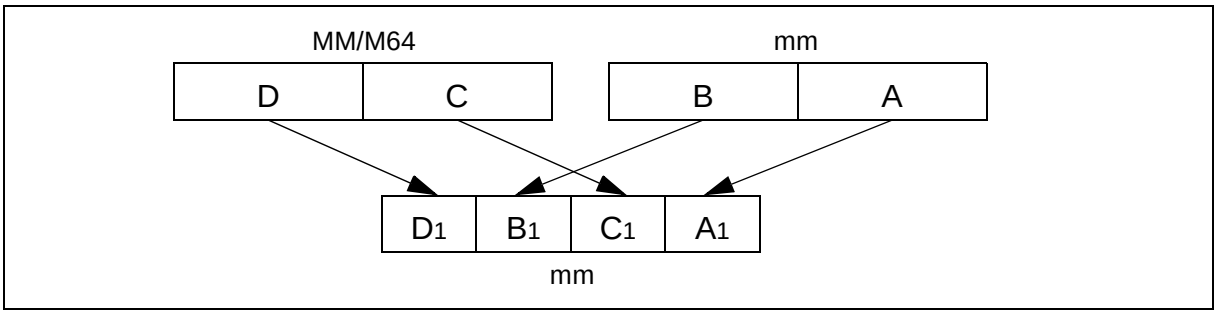


図 4-2. 飽和ありインターリーブ型パックの例

この例では、ソース・オペランドとして符号付きダブルワードを使用し、符号付きワードがインターリーブされた結果を生じる。パック命令は、必要に応じて飽和ありでもなしでも実行することができる。

入力： MM0: 符号付きソース 1 値

MM1: 符号付きソース 2 値

出力： MM0: 最初と 3 番目のワードに MM0 の飽和した符号付きダブルワードがロードされる。

MM0: 2 番目と 4 番目のワードに MM1 の飽和した符号付きダブルワードがロードされる。

```
PACKSSDW  MM0, MM0   ; pack and sign saturate
PACKSSDW  MM1, MM1   ; pack and sign saturate
PUNPKLWD  MM0, MM1   ; interleave the low end 16-bit values of the
                     ; operands
```

パック命令は、常にソース ・オペランドが符号付き数値であると想定している。デスティネーション ・レジスタの結果は、常に操作を実行するパック命令によって定義される。たとえば、PACKSSDW 命令は、2 つのソースの 2 つの符号付き 32 ビット値をそれぞれ 4 つの飽和した符号付き 16 ビット値としてデスティネーション ・レジスタにパックする。それに対して、PACKUSWB 命令は、2 つのソースの 4 つの符号付き 16 ビット値をそれぞれ 4 つの飽和した符号なし 8 ビット値としてデスティネーション ・レジスタにパックする。MMX 命令セット全体の仕様については、『インテル ・アーキテクチャ MMX® テクノロジ ・プログラマーズ ・リファレンス ・マニュアル』 資料番号 243007J）を参照されたい。

4.6.4 飽和なしインターリーブ型パック

以下の例は、結果の各ワードが飽和していない点を除いて 4.6.3 項の例と同じである。その他に、オーバーフローさせないために、この操作では各ダブルワードの下位 16 ビットだけが使用されている。

入力： MM0: 符号付きソース値

MM1: 符号付きソース値

出力： MM0: 最初と 3 番目のワードに MM0 のダブルワードの下位 16 ビットがロードされる。

MM0: 2 番目と 4 番目のワードに MM1 のダブルワードの下位 16 ビットがロードされる。

```
PSLLD       MM1, 16     ; shift the 16 LSB from each of the doubleword
                        ; values to the 16 MSB position
PAND        MM0, {0,ffff,0,ffff}
                        ; mask to zero the 16 MSB of each
                        ; doubleword value
POR         MM0, MM1    ; merge the two operands
```

4.6.5 非インターリーブ型アンパック

アンパック命令は、デスティネーションおよびソースのオペランドのデータ要素をデスティネーション ・レジスタにインター リーブして併合する。以下の例では、2 つのオペランドをインター リーブしないでデスティネーション ・レジスタに併合する。たとえば、ソース 1 のパックド ・ワード ・データ型の 2 つの要素を取り出して結果の下位 32 ビットに入れ、次に、ソース 2 の 2 つのパックド ・ワード ・データ型の 2 つの要素を取り出して結果の上位 32 ビットに入れる。一方のデスティネーション ・レジスタの内容は、図 4-3 に示す組み合わせになる。

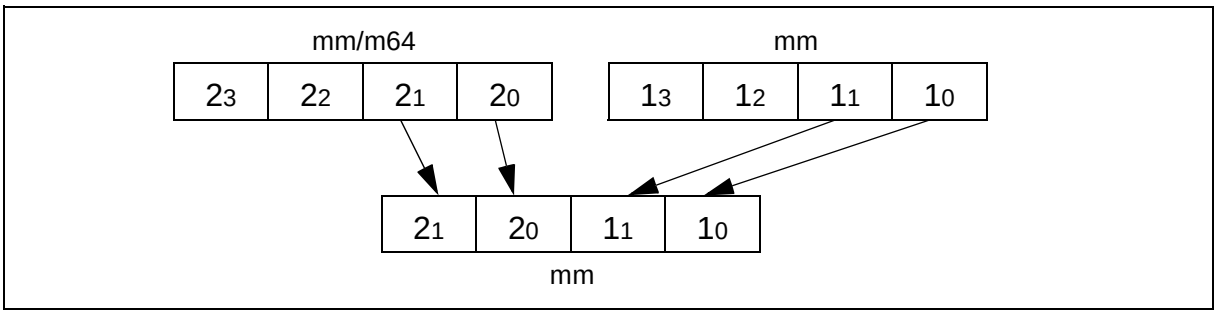


図 4-3. MM0 の非インターリーブ型アンパック結果

他方のデスティネーション・レジスタには、図 4-4 に示すように逆の組み合わせがロードされる。

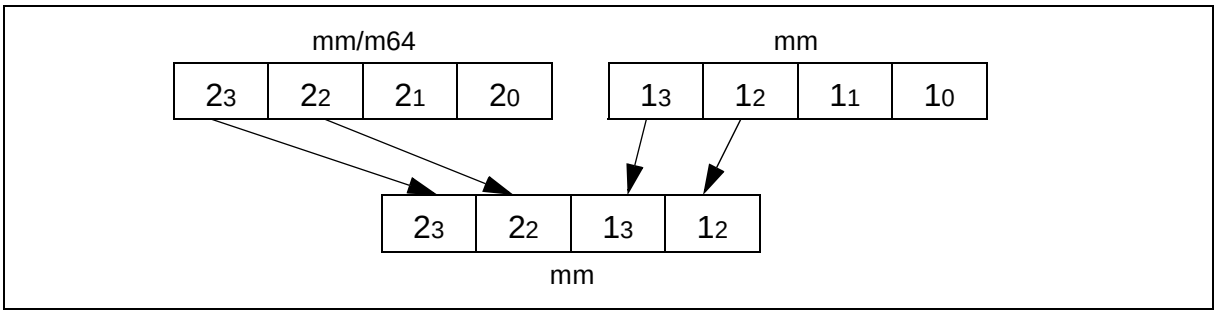


図 4-4. MM1 の非インターリーブ型アンパック結果

以下の例では、2 つのパックド・ワード・ソースを非インターリーブ方式でアンパックする。そのために、ワードをダブルワードにアンパックする命令を使用せず、ダブルワードをクワッドワードにアンパックする命令を使用している。

入力： MM0:パックド・ワード・ソース値

MM1:パックド・ワード・ソース値

出力： MM0:元のソースの 2 つの LOW エンド・ワードがインターリーブされないでロードされる。

MM2:元のソースの 2 つの HIGH エンド・ワードがインターリーブされないでロードされる。

```
MOVQ            MM2, MM0    ; copy source1
PUNPCKLDQ MM0, MM1          ; replace the two high end words of MM0
                            ; with the two low end words of MM1; leave
                            ; the two low end words of MM0 in place
PUNPCKHDQ MM2, MM1          ; move the two high end words of MM2 to the
                            ; two low end words of MM2; place the two
                            ; high end words of MM1 in the two high end
                            ; words of MM2
```

4.6.6 定数との複素乗算

複素乗算は、4つの乗算と2つの加算を必要とする演算である。PMADDWD命令はまさにそのとおりの演算を行う。この命令を使用するためにしなければならないのは、データを4つの16ビット値にフォーマットすることだけである。実数成分および虚数成分は、それぞれ16ビットでなければならない。

入力データをDrおよびDiとする。

ここで、

Dr= データの実数成分

Di= データの虚数成分

メモリ内の定数の複素係数を4つの16ビット値 [Cr-CiCiCr] としてフォーマットする。MOVQ命令でこれらの値をMMXレジスタにロードすることを忘れてはならない。

入力： MM0: 複素数Dr、Di

MM1: [Cr-CiCiCr] の形式の定数の複素係数

出力： MM0: [PrPi] を内容とする2つの32ビット・ダブルワード

複素積の実数成分はPr= Dr*Cr - Di*Ciであり、虚数成分はPi= Dr*Ci+ Di*Crである。

```
PUNPCKLDQ       MM0, MM0   ; This makes [Dr Di Dr Di]
PMADDWD         MM0, MM1   ; and you're done, the result is
                           ; [(Dr*Cr-Di*Ci)(Dr*Ci+Di*Cr)]
```

出力はパックド・ワードであることに注目されたい。必要に応じて、パック命令を使用して、結果を16ビットに変換する(したがって、入力フォーマットに一致させる)ことができる。

4.6.7 符号なし数値間の絶対差

以下の例では、2つの符号なし数値間の絶対差を計算する。この例では、符号なしパックド・バイト・データ型を想定している。ここでは、符号なし飽和ありの減算命令を利用している。この命令は、2つのUNSIGNEDオペランドを受け取り、その間でUNSIGNED(符号なし)飽和あり減算を行う。このUNSIGNED飽和あり減算は、パックド・バイトおよびパックド・ワードのみをサポートしている。パックド・ダブルワードはサポートしていない。

入力： MM0: ソース・オペランド

MM1: ソース・オペランド

出力： MM0: 符号なしオペランド間の絶対差

```
MOVQ            MM2, MM0      ; make a copy of MM0
PSUBUSB         MM0, MM1      ; compute difference one way
PSUBUSB         MM1, MM2      ; compute difference the other way
POR             MM0, MM1      ; OR them together
```

この例は、両オペランドが符号付きの場合は所定の機能を果たさない。次の符号付き絶対差の例を参照されたい。

4.6.8 符号付き数値間の絶対差

この例では、2 つの符号付き数値間の絶対差を計算する。SIGNED オペランドを受け取り、それらの値を UNSIGNED (符号なし) 飽和ありで減算する MMX 減算命令はない。ここで使用している手法は、最初に対応する両方の入力オペランドの要素をパックド・ワードの最大値とパックド・ワードの最小値の順に並べ替え、次に最大値から最小値を引いて要求される絶対差を得るというものである。鍵になるのは、B = XOR(A,XOR(A,B)) および A = XOR(A,0) であるという事実を利用する高速の並べ替え手法である。これにより、パックド・データ型では、一部の要素を XOR(A,B) とし、他の一部の要素を 0 とすれば、そのようなオペランドと A との排他的論理和を取り、一部の位置には値 A を、また他の一部の位置には値 B を受け取れることになる。以下の例では、パックド・ワード・データ型を想定し、各要素は符号付きの値であるとしている。

入力： MM0: 符号付きソース・オペランド

MM1: 符号付きソース・オペランド

出力： MM0: 符号付きオペランド間の絶対差

```
MOVQ            MM2, MM0      ; make a copy of source1 (A)
PCMPGTW         MM0, MM1      ; create mask of source1>source2 (A>B)
MOVQ            MM4, MM2      ; make another copy of A
PXOR            MM2, MM1      ; Create the intermediate value of the swap
                              ; operation - XOR(A,B)
PAND            MM2,MM0       ; create a mask of 0s ans XOR(A,B)
                              ; elements. Where A>B there will be a value
                              ; XOR(A,B) and where A<+=B there will be 0.
MOVQ            MM3, MM2      ; make a copy of swap mask
PXOR            MM4, MM2      ; This is the minima - XOR(A, swap mask)
PXOR            MM1, MM3      ; This is the maxima - XOR(B, swap mask)
PSUBW           MM1, MM4      ; absolute difference = maxima-minima
```

4.6.9 絶対値

以下の例は、| x | の計算に使用する。ここで、x は符号付きである。この例では、符号付きワードをオペランドと想定している。

入力： MM0: 符号付きソース・オペランド

出力： MM1: ABS(MM0)

```
MOVQ        MM1, MM0     ; make copy of x
PSRAW       MM0,15       ; replicate sign bit (use 31 if doing DWORDS)
PXOR        MM0, MM1     ; take 1's complement of just the
                         ; negative fields
PSUBS       MM1, MM0     ; add 1 to just the negative fields
```

絶対値が最大である負の値の絶対値 (すなわち、16 ビットの場合 8000hex) はデスティネーション・オペランドに収まらないが、このコードはこの場合に対して穏当な処置を行っている。つまり、1 をオフにした 7fff を結果としている。

4.6.10 符号付き数値の任意の符号付き範囲 [HIGH,LOW] へのクリップ操作

この例では、符号付きの値を符号付き範囲 [HIGH,LOW] にクリップする方法を示す。値が LOW より小さいか、または HIGH より大きい場合は、それぞれ LOW または HIGH にクリップする。この手法では、符号なし飽和ありのパックド加算およびパックド減算命令を使用している。 符号なし飽和あり」ということは、この手法がパックド・バイト・データ型およびパックド・ワード・データ型に対してしか使用できないことを意味している。

下の 2 つの例では、定数 packed_max および packed_min を使用している。これらの例は、ワード値に対する操作を示している。簡素化のため、以下の各定数を使用している (バイト値を操作する場合は、それぞれ対応する定数が使用される)。

- PACKED_MAX - 0x7FFF7FFF7FFF7FFF
- PACKED_MIN - 0x8000800080008000
- PACKED_LOW - パックド・ワード・データ型の全 4 ワードの LOW 値
- PACKED_HIGH - パックド・ワード・データ型の全 4 ワードの HIGH 値
- PACKED_USMAX - オール・ワンズ
- HIGH_US - HIGH 値を PACKED_MIN の全データ要素 (4 ワード) に加算した結果
- LOW_US - LOW 値を PACKED_MIN の全データ要素 (4 ワード) に加算した結果

入力： MM0: 符号付きソース・オペランド

出力： MM0: 符号なし範囲 [HIGH,LOW] にクリップされた符号付きオペランド

```
PADD      MM0, PACKED_MIN                          ; add with no saturation
                                                   ; 0x8000 to convert to
                                                   ; unsigned
PADDUSW   MM0, (PACKED_USMAX - HIGH_US)            ; in effect this clips
                                                   ; to HIGH
PSUBUSW   MM0, (PACKED_USMAX - HIGH_US + LOW_US)   ; in effect this clips
                                                   ; to LOW
PADDW     MM0, PACEKED_LOW                         ; undo the previous
                                                   ; two offsets
```

上のコードでは、最初に値を符号なし数値に変換し、次にそれらの値を符号なし範囲にクリップする。最後の命令は、データを変換して符号付きデータに戻し、符号付き範囲内に入れる。符号なしデータへの変換は、量 (HIGH - LOW) が 0x8000 より小さいときに正しい結果を得るために必要である。

(HIGH - LOW) >= 0x8000 の場合は、以下のようにアルゴリズムを簡素化することができる。

入力： MM0:符号付きソース・オペランド

出力： MM0:符号なし範囲 [HIGH,LOW] にクリップされた符号付きオペランド

```
PADDSSW   MM0, (PACKED_MAX - PACKED_HIGH)                ; in effect this clips
                                                         ; to HIGH
PSUBSSW   MM0, (PACKED_USMAX - PACKED_HIGH + PACKED_LOW) ; clips to LOW
PADDW     MM0, LOW                                       ; undo the previous
                                                         ; two offsets
```

このアルゴリズムでは、(HIGH - LOW) >= 0x8000 であることが既知のときは、1 サイクルを節約できる。(HIGH - LOW) < 0x8000 のとき、上の 3 命令アルゴリズムが所定の機能を果たさない理由は、[0xffff - 0x8000 未満の任意の数値] からは、負の数値である 0x8000 より絶対値が大きい数値が生じることにある。以下の命令 (上の 3 ステップ・アルゴリズムの 2 番目の命令) が実行されると、負の値からの減算が行われ、当然のケースとして MM0 の値が減少するのではなく増大し、誤った答が得られる。

```
PSUBSSW   MM0, (0xFFFF-HIGH+LOW)
```

4.6.11 符号なし数値の任意の符号なし範囲 [HIGH,LOW] へのクリップ

この例では、符号なしの値を符号なし範囲 [HIGH,LOW] にクリップする。値が LOW より小さい場合、あるいは HIGH より大きい場合は、それぞれ LOW または HIGH にクリップする。この手法では、符号なし飽和ありのパックド加算およびパックド減算命令を使用している。したがって、この手法は、パックド・バイト・データ型およびパックド・ワード・データ型に対してしか使用できない。

この例では、ワード値に対する操作を示す。

入力： MM0:符号なしソース・オペランド

出力： MM0:符号なし範囲 [HIGH,LOW] にクリップされた符号なしオペランド

```
PADDUSW   MM0, 0xFFFF - HIGH           ; in effect this clips to HIGH
PSUBUSW   MM0, (0xFFFF - HIGH + LOW)   ; in effect this clips to LOW
PADDW     MM0, LOW                     ; undo the previous two offsets
```

4.6.12 定数の生成

MMX 命令セットには、即値定数を MMX レジスタにロードする命令がない。以下に示すコード・セグメントは、使用頻度の高い定数を MMX レジスタに生成する。もちろん、定数をローカル変数としてメモリに入れることもできる。ただし、そうしたときは、必ずそれらの値をメモリ内に複製し、MOVQ 命令でそれらの値をロードする。

ＭＭ０にゼロ・レジスタを生成する。

```
PXOR      MM0, MM0
```

ＭＭ１にオール・ワンズ、つまり各パックド・データ・タイプ型フィールドから１を引いた値を生成する。

```
PCMPEQ    MM1, MM1
```

すべてのパックド・バイト[またはパックド・ワード](またはパックド・ダブルワード) フィールドに定数１を生成する。

```
PXOR      MM0, MM0
PCMPEQ    MM1, MM1
PSUBB     MM0, MM1    [PSUBW    MM0, MM1]      (PSUBD    MM0, MM1)
```

すべてのパックド・ワード (またはパックド・ダブルワード) フィールドに符号付き定数 2^{n-1} を生成する。

```
PCMPEQ    MM1, MM1
PSRLW     MM1, 16-n                            (PSRLD    MM1, 32-n)
```

すべてのパックドワード (またはパックド・ダブルワード) フィールドに符号付き整数 -2^{n} を生成する。

```
PCMPEQ    MM1, MM1
PSLLW     MM1, n                               (PSLLD    MM1, n)
```

ＭＭＸ命令セットでは、バイトのシフト命令をサポートしていないので、2^{n-1} および -2^{n} はパックド・ワードおよびパックド・ダブルワード以外には該当しない。

intel

5

浮動小数点アプリケーションの最適化

第 5 章
浮動小数点アプリケーションの最適化

浮動小数点アプリケーションの最適化

本章では、浮動小数点アプリケーションの最適化に関する詳細を示す。本章の説明内容は以下のとおりである。

- 浮動小数点コードの最適化に関する一般規則
- 最適化の実例

5.1 浮動小数点アプリケーションのパフォーマンス向上

浮動小数点アプリケーションのプログラミングに関しては、C または FORTRAN 言語レベルから最適化を意識してコードを書くのが望ましい。多くのコンパイラは、状況に応じて浮動小数点命令のスケジューリングと最適化を行う。ただし、コンパイラにすべて任せ切りにしたのでは、最適なコードを得ることはできない。

5.1.1 浮動小数点コードの最適化に関するガイドライン

浮動小数点アプリケーションの速度向上を図るには、以下の規則に従う。

- コンパイラによる浮動小数点コードの取り扱いを理解する。アセンブリ・ダンプを調べ、プログラムに対してすでにどのような変換が行われているかを確認する。アプリケーションの実行時間の大半を占めるループ・ネストについて検討する。コンパイラが最高速のコードを作成しない理由を調べる。
- 解決可能な依存関係がないかどうかを調べる。
  - メモリ・バンド幅が狭い
  - キャッシュ・ヒット率が低い
  - 浮動小数点算術演算のレイテンシが大きい
- 不必要に高い精度は使わない。演算内容によっては、単精度 (32 ビット) の方が高速である。しかも、単精度の場合に必要なメモリ・スペースは、倍精度 (64 ビット) または拡張倍精度 (80 ビット) の場合の半分になる。
- 高速の浮動小数点から整数への変換 (フロート・ツー・インテジャ) ルーチンを使用していることを確認する。多くのライブラリは必要以上の処理を実行する。浮動小数点から整数への変換には、必ず高速ルーチンを使用する。5.4 節を参照。
- アプリケーションが所定の範囲内であることを確認する。範囲を超えると、オーバーヘッドが非常に大きくなる。
- FXCH を使用して、コードをアセンブリ言語でスケジュールする。ループをアンロールし、コードをパイプライン化する。5.1.2 項を参照。
- コードの変換により、メモリ・アクセス・パターンを改善する。ループ・フュージョンまたは圧縮を使用して、できるだけ多くの計算をキャッシュ内で行えるようにする。5.5 節を参照。
- 依存性チェーンを断つ。

5.1.2 並行性の改善

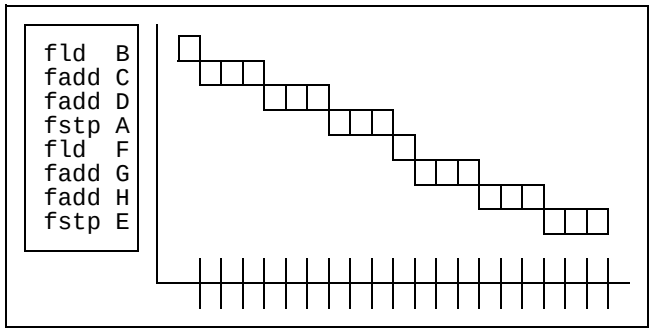
Pentium、Pentium Pro、および Pentium Ⅱプロセッサには、パイプライン化された浮動小数点ユニットがある。浮動小数点命令をスケジュールすることにより、Pentium プロセッサの浮動小数点ユニットから最高のスループットを得ることができる。さらに、浮動小数点ユニットのパイプライン化の改善があれば、これらの最適化は Pentium Pro および Pentium Ⅱプロセッサにも有効である。以下の図 5-1 の例について検討されたい。

ソース・コード:

```
A = B + C + D;
E = F + G + E;
```

アセンブリ・コード:

```
fld  B
fadd C
fadd D
fstp A
fld  F
fadd G
fadd H
fstp E
```



合計 20 サイクル

図 1. 浮動小数点例

Pentium、Pentium Pro、および Pentium Ⅱプロセッサの並行処理性を活用するには、どの命令が並行に実行できるかを知る必要がある。上記の例の 2 つの高水準コード・ステートメントは独立しており、したがって、それらのアセンブリ命令は並行に実行されるようにスケジュールすることができ、その結果、実行速度を向上させることができる。

ソース・コード:

```
A = B + C + D;
E = F + G + E;
```

```
fld  B       fld  F
fadd C       fadd G
fadd D       fadd H
fstp A       fstp E
```

大部分の浮動小数点演算では、1 つのオペランドと結果がスタックのトップを使用する必要がある。このため、各命令がそれぞれ前の命令に依存することになり、命令をオーバラップさせることができない。

これを克服するための 1 つの方法は、スタックの代わりに、フラットな浮動小数点レジスタ・ファイルを使用した場合のことを想像することである。その場合、コードは次のようになる。

```
fld   B           →F1
fadd  F1,C        →F1
fld   F           →F2
fadd  F2,G        →F2
fadd  F1,D        →F1
fadd  F2,H        →F2
fstp  F1          →A
fstp  F2          →E
```

これらの想像上のレジスタの機能をインプリメントするには、fxch 命令を使用してスタックのトップの値を変更する必要がある。これにより、スタックのトップへの依存性を解くことができる。fxch 命令は通常の浮動小数点演算とのペアリングが可能であり、したがって、Pentium プロセッサではペナルティを生じない。さらに、Pentium Pro および Pentium Ⅱプロセッサでは、fxch は余分な実行サイクルを使用しない。

```
                                      ST0       ST1
fld   B           →F1  fld B          B
fadd  F1,C        →F1  fadd C         B+C
fld   F           →F2  fld F          F         B+C
fadd  F2,G        →F2  fadd G         F+G       B+C
                       fxch ST(1)     B+C       F+G
fadd  F1,D        →F1  fadd D         B+C+D     F+G
                       fxch ST(1)     F+G       B+C+D
fadd  F2,H        →F2  fadd H         F+G+H     B+C+D
                       fxch ST(1)     B+C+D     F+G+H
fstp  F1          →A   fstp A         F+G+H
fstp  F2          →E   fstp E
```

Pentium プロセッサでは、fxch 命令は先行の fadd 命令とペアリングし、それらの命令と並行に実行される。fxch 命令は、次の浮動小数点命令が使用できる位置にオペランドを移動する。結果として、図 5-2 に示すように、Pentium プロセッサでは実行速度が向上する。

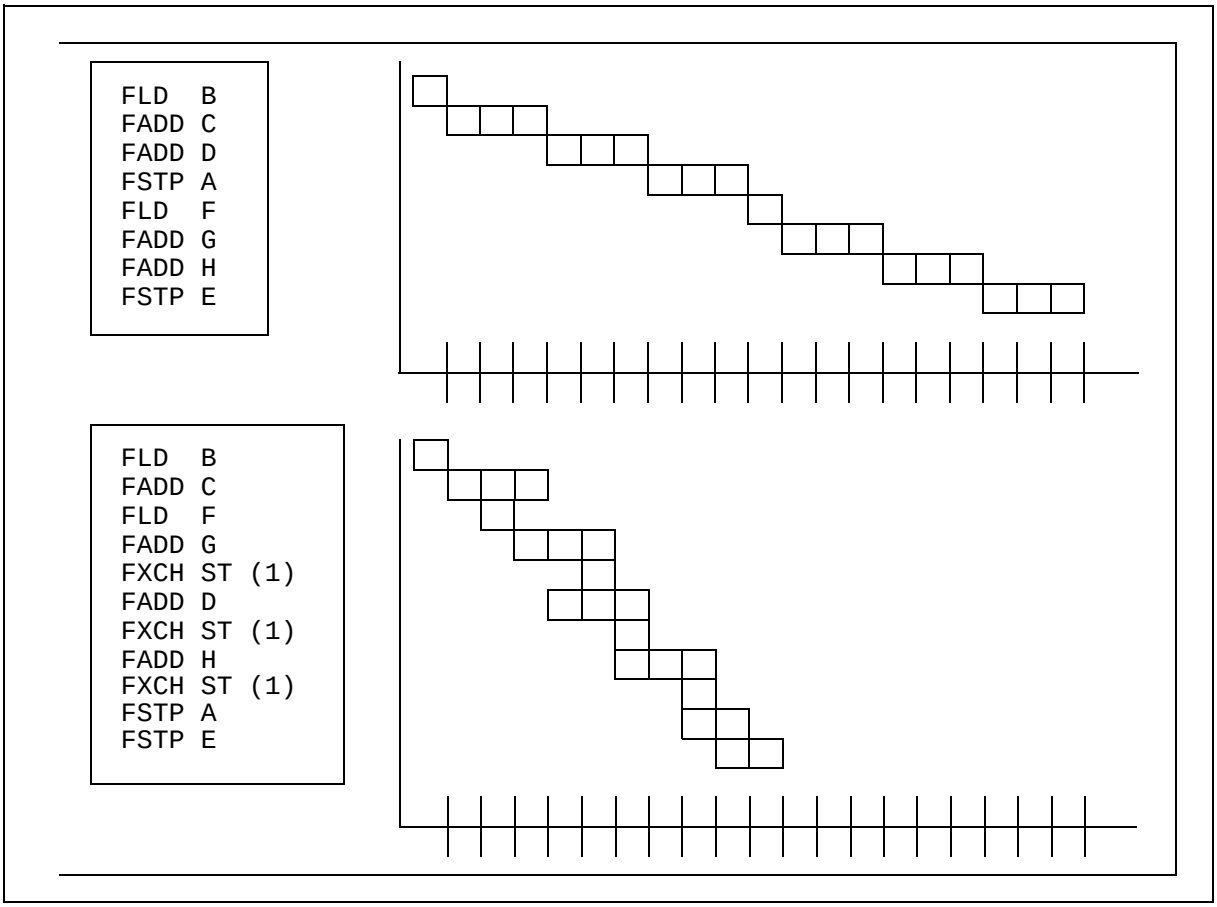


図 2. 浮動小数点コード最適化の前後

5.1.2.1 fxch に関する規則と調整

Pentium プロセッサでは、以下の条件がすべて成立するときは、fxch 命令は V パイプで他の浮動小数点命令とともに実行されるので、この命令には余分なサイクルのコストはかからない。

- fxch 命令の直後が FP 命令である。
- fxch 命令の直前が、fadd、fsub、fmul、fld、fcom、fucom、fchs、ftst、fabs、fdiv のうちのいずれかの FP 命令である。
- fxch 命令がすでに 1 回実行されている。これは、fxch 命令が初めて実行されるときにキャッシュ内の命令境界がマークされ、したがって、fxch 命令が 2 回目にキャッシュから実行されるとき初めてペアリングが行われるためである。

上記の各条件が成立するときは、fxch 命令はほとんど「フリー」である。これを使用すると、要素をストアしてから再びロードするのではなく、FP スタックのレベルより深いレベルでそれらの要素にアクセスすることができる。

5.2 メモリ・オペランド

Pentium プロセッサでは、メモリ・オペランドがキャッシュにある場合、スタック・レジスタの代わりにメモリ・オペランドに対して浮動小数点演算を実行すると、サイクル・コストがかからない。Pentium Pro および Pentium II プロセッサでは、メモリ・オペランドを使用する命令からは 2 つの µop が生成され、デコードを制限することがある。さらに、メモリ・オペランドはデータ・キャッシュ・ミスの原因になり、したがってペナルティを生じることもある。浮動小数点オペランドは 64 ビットであり、8 バイト境界にアライメントを合わせる必要がある。デコードの詳細については、3.6.4 項を参照されたい。

5.3 メモリ・アクセス・ストール

浮動小数点レジスタには、64 ビット値を倍精度値としてロードすることができる。8、16、または 32 ビットの長さの単一の配列値をロードする代わりに、単一のクワッドワード値をロードし、次にそれに従って構造体または配列のポインタをインクリメントしていくことを考えてみる。

第一に、クワッドワード・データのロードとストアは、大きいクワッドワード・データ・ブロックを使用した方が効率的である。第二に、こうすることは、メモリ・アドレスに対する 8、16、または 32 ビットのロードおよびストア操作と 64 ビットのロードおよびストア操作との混用を避ける上でも効果的である。これにより、Pentium Pro および Pentium II プロセッサではメモリ・アクセス・ストールの可能性がなくなる。メモリ・アクセス・ストールは、以下の場合に発生する。

- 同一のメモリ領域で大きいストアの後に小さいロードが行われる。
- 同一のメモリ領域で小さいストアの後に大きいロードが行われる。このような状況では、Pentium Pro および Pentium II プロセッサはストールする。

以下の例について検討されたい。第 1 のケースでは、(メモリ・アドレス mem から始まる) 同一のメモリ領域で一連の小さいストアの後に大きいロードが行われる。この場合、大きいロードはストールする。

```
mov         mem, eax          ; store dword to address “mem”
mov         mem + 4, ebx      ; store dword to address “mem + 4”
            :
            :
fld         mem               ; load qword at address “mem”, stalls
```

fld は、一連のストアのメモリ書き込みが終わるのを待ってから、必要なすべてのデータにアクセスしなければならない。このストールは、他のデータ型でも (たとえば、バイトまたはワードがストアされ、次に同一のメモリ領域からワードまたはダブルワードが読み取られるときにも) 発生する可能性がある。

第 2 のケースでは、(メモリ・アドレス mem から始まる) 同一のメモリ領域で大きいストアの後に一連の小さいロードが行われる。この場合、小さいロードはストールする。

```
fstp         mem              ; load qword to address “mem”
            :
            :
mov         bx, mem + 2       ; load word at address “mem + 2”, stalls
mov         cx, mem + 4       ; load word at address “mem + 4”, stalls
```

一連のワード・ロードは、クワッドワード・ストアのメモリ書き込みが終わるのを待ってから、必要なデータにアクセスしなければならない。このストールは、他のデータ型でも (たとえば、ダブルワードまたはワードがストアされ、次に同一のメモリ領域からワードまたはバイトが読み取られるときにも) 発生する可能性がある。これは、ストア命令をできるだけロード命令から離すことによって避けることができる。一般に、ストール状態を避けるには、ロード命令とストア命令は 10 命令以上離す必要がある。

5.4 浮動小数点から整数への変換

多くのライブラリは、浮動小数点値を整数に変換する「フロート・ツー・インテジャ」ライブラリ・ルーチンを備えている。このようなライブラリの多くは、丸めモードを切り捨てとして規定しているANSIのC コーディング規格に準拠している。FIST 命令のデフォルトは最も近い整数値への丸めであり、したがって、多くのコンパイラでは、C および FORTRAN の基準に準拠するために、プロセッサの丸めモードを変更する。この変更をインプリメントするには、fldcw 命令を使用してプロセッサの制御ワードを変更する必要がある。この命令は同期型の命令であり、Pentium、Pentium Pro、および Pentium II プロセッサではアプリケーションのパフォーマンスが大きく低下する。

アプリケーションのインプリメントに当たっては、結果に対して丸めモードを適用するのが重要であるかどうかを検討する。重要でなければ、以下に示す関数を使用して、同期とfldcw 命令のオーバーヘッドを避ける。

丸めモードの変更を避けるには、以下に示すアルゴリズムを使用するとよい。

```
_fto132proc
     lea      ecx,[esp-8]
     sub      esp,16          ; allocate frame
     and      ecx,-8          ; align pointer on boundary of 8
     fld      st(0)           ; duplicate FPU stack top
     fistp    qword ptr[ecx]
     fild     qword ptr[ecx]
     mov      edx,[ecx+4]     ; high dword of integer
     mov      eax,[ecx]       ; low dword of integer
     test     eax,eax
     je       integer_QNaN_or_zero

arg_is_not_integer_QNaN:
     fsubp    st(1),st        ; TOS=d-round(d),
                              { st(1)=st(1)-st & pop st }
     test     edx,edx         ; what's sign of integer
     jns      positive        ; number is negative
                              ; dead cycle
                              ; dead cycle
     fstp     dword ptr[ecx] ; result of subtraction
     mov      ecx,[ecx]       ; dword of difference(single precision)
     add      esp,16
     xor      ecx,80000000h
     add      ecx,7fffffffh  ; if difference>0 then increment integer

     adc      eax,0           ; inc eax (add CARRY flag)
     ret
positive:
     fstp     dword ptr[ecx] ; 17-18 ; result of subtraction
     mov      ecx,[ecx]       ; dword of difference (single precision)

     add      esp,16
     add      ecx,7fffffffh  ; if difference<0 then decrement integer
     sbb      eax,0           ; dec eax (subtract CARRY flag)
     ret
integer_QNaN_or_zero
     test     edx,7fffffffh
     jnz      arg_is_not_integer_QNaN
     add      esp,16
     ret
```

5.5 ループのアンロール

ループのアンロールには多くの利点があるが、これらの利点は命令キャッシュの制約条件およびその他のマシン・リソースとのバランスを計る必要がある。利点は、以下のとおりである。

- アンロールすると分岐オーバーヘッドが解消される。Pentium、Pentium Pro、Pentium IIプロセッサでは、BTBでループを予測することができ、ループ・インデックスのインクリメント用およびジャンプ用の命令のコストは低い。
- アンロールすることにより、レイテンシを隠蔽するようにループを意図的にスケジュール（またはパイプライン化）することができる。これは、問題のパスを明らかにするために依存性チェーンを展開しても、空きレジスタが十分にあって変数を「活きた」状態に維持できる場合に効果的である。
- 命令のフェッチおよびデコードに関する制約が緩和されるよう、ループを意図的にスケジュールすることができる。
- Pentium Proおよび Pentium IIプロセッサでは、(分岐すると予測された)後方分岐のペナルティは1クロックだけであり、したがって、非常に細かいループ本体をアンロールして解くことができる。
- 以下の例に示すように、アンロールすると、その他の最適化の可能性が見えてくることがある。

下のループは100回実行され、すべての偶数番目の要素にxを、またすべての奇数番目の要素にyをそれぞれ代入する。

```
do i=1,100
  if (i mod 2 == 0) then a(i) = x
  else a(i) = y
enddo
```

このループをアンロールすると、各繰り返しでxとyの代入を行い、ループ内の1つの分岐を除去することができる。

```
do i=1,100,2
  a(i) = y
  a(i+1) = x
enddo
```

5.6 浮動小数点ストール

浮動小数点命令の多くには1サイクルより大きいレイテンシがあり、したがってPentium プロセッサ・ファミリでは、最初の命令の実行が完了するまで、次の浮動小数点命令はその結果にアクセスすることができない。このレイテンシを隠蔽するためには、パイプ・ストールを生じる命令ペアの間に他の命令をいくつか挿入する必要がある。挿入する命令は、整数命令またはそれら自身が新しいストールを生じない浮動小数点命令にする。挿入する命令の数は、レイテンシの長さによって異なる。Pentium Proおよび Pentium IIプロセッサでは、アウトオブオーダ実行が可能であるため、ストールは必ずしも命令または µop 単位で生じるとはかぎらない。しかし、FDIV などのように命令のレイテンシが非常に長い場合は、命令をスケジュールすることによりアプリケーション全体としてのスループットを向上させることができる。以降の各項に、Pentium プロセッサ・ファミリでの浮動小数点パイプライン化に対する留意点をまとめて示す。

5.6.1 整数命令による浮動小数点命令レイテンシの隠蔽

浮動小数点命令が直前の命令の結果に依存し、かつその命令も浮動小数点命令であるときは、それら2つのFP命令の間に整数命令を入れると、たとえそれらの整数命令がループ制御を行うものであっても有利である。以下の例では、そのようにしてループの構造を変更している。

```
for (i=0; i<Size; i++)
     array1 [i] += array2 [i];
; assume eax=Size-1, esi=array1, edi=array2
                                Pentium Processor
                                     CLOCKS
     LoopEntryPoint:
     fld  real4 ptr [esi+eax*4]          ; 2 - AGI
     fadd real4 ptr [edi+eax*4]          ; 1
     fstp real4 ptr [esi+eax*4]          ; 5 - waits for fadd
     dec  eax                            ; 1
     jnz  LoopEntryPoint

; assume eax=Size-1, esi=array1, edi=array2
     jmp  LoopEntryPoint
     Align     16
TopOfLoop:
     fstp real4 ptr [esi+eax*4+4]        ; 4 - waits for fadd + AGI
LoopEntryPoint:
     fld  real4 ptr [esi+eax*4]          ; 1
     fadd real4 ptr [edi+eax*4]          ; 1
     dec  eax                            ; 1
     jnz  TopOfLoop
     ;
fstp real4 ptr [esi+eax*4+4]
```

fadd とfstp の間に整数命令を移動することにより、整数命令は、浮動小数点ユニット内での fadd の実行中に、fstp の実行が開始する前に実行することができる。この新しいループ構造は、ループが fld から開始される必要があるので、最初の繰り返しには別のエントリ・ポイントを必要とする。さらに、ループの最後の繰り返しを完了するために、条件付きジャンプの後に fstp を1つ追加する必要がある。

5.6.2 浮動小数点ストアの 1 クロック・レイテンシの隠蔽

浮動小数点ストアでは、その浮動小数点オペランドのために 1 サイクル余分に待たなければならない。fld の後では、fst は 1 クロック待たなければならない。一般的な算術演算 fmul および fadd には通常 3 サイクルのレイテンシがあるが、それらの後では fst は余分に 1 サイクル待つので、合計レイテンシは 4 サイクルになる。[1]

```
fld      mem1               ; 1 fld takes 1 clock
                            ; 2 fst waits, schedule something here
fst      mem2               ; 3,4 fst takes 2 clocks

fadd     mem1               ; 1 add takes 3 clocks
                            ; 2 add, schedule something here
                            ; 3 add, schedule something here
                            ; 4 fst waits, schedule something here
fst      mem2               ; 5,2 fst takes 2 clocks
```

以下の例では、ストアはその前のロードに依存しない。

```
fld      mem1               ; 1
fld      mem2               ; 2
fxch     st(1)              ; 2
fst      mem3               ; 3 stores values loaded from mem1
```

レジスタは、(fld で) ロードされた直後に使用することができる。

```
fld      mem1               ; 1
fadd     mem2               ; 2,3,4
```

fadd、fsub、または fmul によって書き込まれたレジスタが、その直後に別の浮動小数点演算に使用されるときは、2 サイクルの遅延が生じる。これら 2 つの命令の間に他の命令を挿入すれば、レイテンシと潜在的なストールを隠蔽することができる。

さらに、浮動小数点ユニット・パイプ内で実行されるマルチサイクル浮動小数点命令 (fdiv および fsqrt) もある。これらの命令を浮動小数点ユニット・パイプで実行している間に、整数命令を実行することができる。マルチサイクル命令の後で整数命令をいくつも発行すると、整数実行ユニットはビジー状態が続く (正確な命令数は浮動小数点命令のサイクル数によって異なる)。

整数命令は、最後の浮動小数点演算が fxch であったとき以外は、浮動小数点演算とオーバーラップする。最後の命令が fxch であった場合は、1 サイクルの遅延が生じる。

```
   U パイプ :            V パイプ :
     fadd                fxch       ; 1
                                    ; 2 fxch delay
     mov eax, 1          inc edx    ;
```

[1] このセットには、さらに faddp、fsubrp その他の命令も含まれる。

5.6.3 整数と浮動小数点の乗算

整数乗算 mulおよび imul は、浮動小数点ユニット内で実行されるので、これらの命令を浮動小数点命令と並行して実行させることはできない。

直前のサイクルで fmul または fmul/fxch ペアが実行された場合は、浮動小数点乗算命令 (fmul) は 1 サイクル遅延する。乗算器は、1 サイクル置きにしか新しいオペランド・ペアを受け付けられない。

5.6.4 整数オペランドでの浮動小数点演算

整数オペランドを取る浮動小数点演算 (fiadd または fisub など) は使用しないようにする。これらの命令は、fild と浮動小数点演算という2つの命令に分割することが望ましい。以下の例に示すように、fiadd の次の命令を発行できるまでのサイクル数 (スループット) は 4 サイクルであるのに対し、fild および単純な浮動小数点演算のそれは 1 サイクルである。

複雑な命令：

```
fiadd  [ebp]   ; 4
```

オーバーラップの機会が多くなる：

```
fild   [ebp]   ; 1
faddp  st(1)   ; 2
```

fild - faddp 命令を使用すると、2 つの空きサイクルができ、そこで他の命令を実行することができる。

5.6.5 FSTSW 命令

fstsw 命令は、通常浮動小数点比較命令 (fcom、fcomp、fcompp) の後に使用され、3 サイクル遅延する。そこで、比較命令の後に他の命令を挿入すれば、その遅延を隠蔽することができる。Pentium Pro および Pentium II プロセッサでは、代わりに fcmov 命令を使用することができる。

5.6.6 超越関数

超越演算は U パイプで実行されるが、これらの命令は他のどの命令ともオーバーラップさせることができず、したがって、そのような命令の後の整数命令は前者が完了するまで待つことになる。

Pentium Pro および Pentium II プロセッサでは、超越演算の実行はずっと高速である。ライブラリ・コールに介在するコールおよびプロローグ / エピローグのオーバーヘッドがもはや無視できないという事実から、これらの算術演算ライブラリ・コールの一部をインライン展開した方がよい場合もある。ソフトウェアによるこれらの演算のエミュレーションは、精度を犠牲にしないかぎり、ハードウェアより高速にはなることはない。

5.6.7 バック・ツー・バック浮動小数点命令

オペランドの指数が [-1FFF,-0FFF] および [1000,1FFE] の範囲のとき、つまり、非常に大きい拡張精度数値および非常に小さい拡張精度数値に対して、バック・ツー・バック浮動小数点乗算または除算の命令を実行すると、MMX テクノロジ Pentium プロセッサはストールする。Pentium プロセッサでは、このストールはない。以下の例を参照されたい。

```
FMUL ST0,ST1
FLD fld1
```

指数同士の加算から生じた値が上記の範囲内の場合、FLD 操作は、FMUL 演算がオーバーフローの結果として例外を発生するかどうかの確認を待つ (オーバーフローは、ST0*ST1 > MAX の場合に発生する)。

intel

6

コンパイラ選択のガイドライン

第 6 章
コンパイラ選択のガイドライン

今日では、多くのコンパイラが市場に出回っている。その中から、最も最適化されたコードを生成することのできるコンパイラを選ぶのは難題である。本章では、コンパイラ選びのポイントとして何に注目すべきかを説明する。さらに、最適化のためのコンパイル・スイッチの概要と、インテル・アーキテクチャの世代間の相違を示す。最後に、6.2.4 項でコード最適化のためのブレンデッド (全世代対応) 戦略を推奨する。

6.1 コンパイラについて注目すべき機能

以下に、アプリケーション開発用のコンパイラに必要な機能を示す。これらは主としてパフォーマンス指向の機能である。リスト順は優先順位とは無関係であり、いずれも同じ比重である。

- 「ブレンデッド・コード」生成用のスイッチの他に、特定のプロセッサを対象とするスイッチ (6.2 節で説明) があること。
- すべてのデータ・サイズのアライメントを適切な境界に合わせること。さらに、ターゲット分岐のアライメントを 16 バイト境界に合わせられること。
- プロシージャ間 (プログラム全体) の分析と最適化を実行できること。
- プロフィールに基づいて最適化を実行できること。
- 行番号およびその他のテキスト付きアセンブリ・コードのリストを表示できること。
- 適切なインライン・アセンブリ・サポートがあること。インライン・アセンブリがある上に、コンパイラが高水準言語のコードを最適化できれば、利点がさらに増大する。
- 3.5.1.5 項で説明したループ変換などのメモリ階層を対象とする「高度な最適化」を実行できること。
- 最適化コードのデバッグ機能を備えていること。VTune のチューニング環境に関しては、デバッグ情報の生成が非常に重要である。
- MMX® テクノロジをサポートしていること。少なくとも、インライン・アセンブリおよび 64 ビット・データ型をサポートしていること。組み込み関数をサポートしていればベストである。
- 信頼性が高いこと。つまり、あらゆる水準の最適化で正しいコードを生成すること。

コンパイラの購入に際しては、使い勝手など、他にも考慮すべき重要な問題が多数ある。最低でも、購入しようと考えているコンパイラの評価版を注文し、実際のアプリケーションでコンパイラのベンチマーク・テストを行うこと。ベンチマーク・テストで得られる情報は、コンパイラ選びの決め手である。

6.2 コンパイラ・スイッチ

次項以降では、インテル・アーキテクチャのコンパイラで指定すべきコンパイラ・スイッチを示す。デフォルトは、本来プロセッサ・ファミリ全体を対象とする最適化を狙ったブレンデッド・スイッチである。特定のプロセッサに固有のスイッチも必要である。

6.2.1 デフォルト(ブレンデッド・コード)

ブレンデッド・コードを生成する。このスイッチを指定してコンパイルされたコードはすべてのインテル・アーキテクチャ・プロセッサ(Intel386、Intel486、Pentium、Pentium Pro、および Pentium II)で実行することができる。このスイッチは、複数の世代のプロセッサで実行される可能性のあるコード向けである。このスイッチを指定した場合は、コード・ジェネレータによってパーシャル・レジスタ・ストールが発生することはないはずである。

6.2.2 プロセッサ固有のスイッチ

6.2.2.1 ターゲット・プロセッサ - Pentium® プロセッサ

最適の Pentium プロセッサ・コードを生成する。生成されたコードは、すべての 32 ビット・インテル・アーキテクチャ・プロセッサで動作する。このスイッチは、Pentium プロセッサだけで動作するコード向けである。

6.2.2.2 ターゲット・プロセッサ - Pentium® Pro プロセッサ

最適の Pentium Pro プロセッサ・コードを生成する。生成されたコードは、すべての 32 ビット・インテル・アーキテクチャ・プロセッサで動作する。このスイッチは、Pentium Pro および Pentium II プロセッサだけで動作するコード向けである。パーシャル・ストールの発生はないはずである。

6.2.3 その他のスイッチ

6.2.3.1 Pentium® Pro プロセッサ用の新しい命令

このスイッチは、Pentium Pro プロセッサ固有の新しい命令(`cmov`、`fcmov`、および `fcomi`)を使用する。これは、Pentium Pro プロセッサ固有のスイッチとは独立している。ターゲット・プロセッサ・スイッチも指定した場合は、Pentium Pro プロセッサ形式のコスト分析に依存して'`if to cmov`'最適化が行われる。

6.2.3.2 コード・サイズ削減のための最適化

このスイッチを指定すると、生成されるコードのサイズを小さくすることができる。これによって、実行速度が犠牲になることもある(たとえば、ストアではなくプッシュを使用する場合など)。このスイッチは、命令キャッシュ・ミス率の高いプログラム向けである。なお、このスイッチを指定すると、ターゲット・プロセッサに関係なく、コード・アライメントはオフになる。

6.2.4 プロセッサ世代間の相違

以下の表に、Pentium、Pentium Pro、Pentium Ⅱプロセッサ間のマイクロ・アーキテクチャ上の相違を示す。コード生成に関する留意点も示してある。

表 6-1. インテル・マイクロプロセッサ・アーキテクチャの相違

	Pentium® プロセッサ	Pentium® Pro プロセッサ	MMX® テクノロジ Pentium プロセッサ	Pentium Ⅱ プロセッサ
キャッシュ	8K コード、8K データ	8K コード、8K データ	16K コード、16K データ	16K コード、16K データ
プリフェッチ	4x32 ビット・プライベート・バス、キャッシュ接続	4x32 ビット・プライベート・バス、キャッシュ接続	4x32 ビット・プライベート・バス、キャッシュ接続	4x32 ビット・プライベート・バス、キャッシュ接続
デコーダ	2 デコーダ	3 デコーダ	2 デコーダ	3 デコーダ
コア	5 ステージのパイプラインおよびスーパースケーラ	12 ステージのパイプラインおよび動的実行	6 ステージのパイプラインおよびスーパースケーラ	12 ステージのパイプラインおよび動的実行
算術演算	オンチップおよびパイプライン化	オンチップおよびパイプライン化	オンチップおよびパイプライン化	オンチップおよびパイプライン化

以下に、インテル・アーキテクチャ・ファミリ全体を対象とするブレンデッド・コードのガイドラインを示す。

- 予測ミスされたラベルまたは割り込み関数などの重要なコード・エントリ・ポイントは、16 バイト境界にアライメントを合わせる。
- パーシャル・ストールを避ける。
- アドレス生成インターロック (AGI) およびその他のパイプライン・ストールを除去するように命令をスケジュールする。
- 単純な命令を使用する。
- 分岐予測アルゴリズムに従う。

浮動小数点コードをスケジュールして、スループットを向上されたい。

intel

7

パフォーマンス監視機能の拡張

第 7 章
パフォーマンス監視機能の拡張

アプリケーション・コードの最も効率的な改善方法は、コード内からパフォーマンスのボトルネックを見つけ出し、ストール条件を除去することである。ストール条件を識別するため、インテル・アーキテクチャ・プロセッサでは、アプリケーションのパフォーマンスに関する情報を収集する 2 つのカウンタを組み込んでいる。これらのカウンタは、コードの実行中に発生するイベントを監視するもので、これらのカウンタが収集した情報は、プログラム実行中に読み取ることができる。これにより、そのアプリケーションでストールが発生しているかどうか、発生しているとすればどの部分に問題があるのかを容易に調べることができる。この 2 つのカウンタには、インテルの VTune を使用するか、またはアプリケーション・コード内でパフォーマンス・カウンタ用の命令を使用してアクセスすることができる。

本章では、Pentium、Pentium Pro、および Pentium II プロセッサのパフォーマンス監視機能について説明する。

RDPMC 命令については、7.3 節で説明する。

7.1 スーパースケーラ (Pentium® プロセッサ・ファミリ) のパフォーマンス監視イベント

すべての Pentium プロセッサには、2 つのパフォーマンス・カウンタが組み込まれ、インテル MMX テクノロジをサポートするための新しいイベントがいくつか追加されている。これらの新しいイベントは、「D1 スターベーション」や「FIFO エンプティ」などの「双子イベント」を除いて、2 つのイベント・カウンタ (CTR0、CTR1) のどちらか 1 つに割り当てる。双子イベントは、それらを同時に並行してカウントできるように別のカウンタに割り当てる。各イベントはそれぞれに指定されているカウンタに割り当てなければならない。表 7-1 に、パフォーマンス監視用のイベントの一覧を示す。新しいイベントには網かけしてある。

表 7 - 1. パフォーマンス監視イベント

シリアル番号	エンコーディング	カウンタ0	カウンタ1	パフォーマンス監視イベント	発生回数と持続時間の区別
0	000000	○	○	データ読み取り	発生
1	000001	○	○	データ書き込み	発生
2	000010	○	○	データTLB ミス	発生
3	000011	○	○	データ読み取りミス	発生
4	000100	○	○	データ書き込みミス	発生
5	000101	○	○	M またはE ステート・ラインへの書き込み(ヒット)	発生
6	000110	○	○	データ・キャッシュ・ライン・ライトバックあり	発生
7	000111	○	○	外部データ・キャッシュ・スヌープ	発生
8	001000	○	○	外部データ・キャッシュ・スヌープ・ヒット	発生
9	001001	○	○	両パイプ内メモリ・アクセス	発生
10	001010	○	○	バンク衝突	発生
11	001011	○	○	アライメントずれデータ・メモリまたは I/O 参照	発生
12	001100	○	○	コード読み取り	発生
13	001101	○	○	コードTLB ミス	発生
14	001110	○	○	コード・キャッシュ・ミス	発生
15	001111	○	○	セグメント・レジスタへのロード	発生
16	010000	○	○	予約済み	
17	010001	○	○	予約済み	
18	010010	○	○	分岐	発生
19	010011	○	○	BTB 予測数	発生
20	010100	○	○	分岐した分岐またはBTB ヒット	発生
21	010101	○	○	パイプライン・フラッシュ数	発生
22	010110	○	○	実行命令数	発生

表 7 - 1. パフォーマンス監視イベント　続き）

シリアル番号	エンコーディング	カウンタ 0	カウンタ 1	パフォーマンス監視イベント	発生回数と持続時間の区別
23	010111	○	○	V パイプ実行命令数、たとえば、並行 / ペア実行	発生
24	011000	○	○	バス ・サイクル進行中クロック数 (バス利用率)	持続
25	011001	○	○	フル ・ライト ・バッファによるストール ・クロック数	持続
26	011010	○	○	データ ・メモリ読み取り待ちによるパイプライン ・ストール	持続
27	011011	○	○	E または M ステート ・ラインへの書き込み時ストール	持続
29	011101	○	○	I/O 読み取りまたは書き込みサイクル	発生
30	011110	○	○	キャッシュ不可能メモリ読み取り数	発生
31	011111	○	○	アドレス生成インターロックによるパイプライン ・ストール	持続
32	100000	○	○	予約済み	
33	100001	○	○	予約済み	
34	100010	○	○	FLOP 数	発生
35	100011	○	○	DR0 レジスタのブレークポイント ・マッチ	発生
36	100100	○	○	DR1 レジスタのブレークポイント ・マッチ	発生
37	100101	○	○	DR2 レジスタのブレークポイント ・マッチ	発生
38	100110	○	○	DR3 レジスタのブレークポイント ・マッチ	発生
39	100111	○	○	ハードウェア割り込み数	発生
40	101000	○	○	データ読み取り数またはデータ書き込み数	発生
41	101001	○	○	データ読み取りミス数またはデータ書き込みミス数	発生

表 7 - 1. パフォーマンス監視イベント　続き）

シリアル番号	エンコーディング	カウンタ 0	カウンタ 1	パフォーマンス監視イベント	発生回数と持続時間の区別
43	101011	○	×	U パイプ実行 MMX® 命令数	発生
43	101011	×	○	V パイプ実行 MMX 命令数	発生
45	101101	○	×	実行 EMMS 命令数	発生
45	101101	×	○	MMX 命令 /FP 命令間遷移	発生
46	101110	×	○	キャッシュ不可能メモリ書き込み数	発生
47	101111	○	×	実行飽和あり MMX 命令数	発生
47	101111	×	○	実行飽和 MMX 命令数	発生
48	110000	○	×	HLT 状態外サイクル数	持続
49	110001	○	×	MMX 命令データ読み取り数	発生
50	110010	○	×	浮動小数点ストール数	持続
50	110010	×	○	分岐した分岐数	発生
51	110011	×	○	D1 欠乏および FIFO 内 1 命令	発生
52	110100	○	×	MMX 命令データ書き込み数	発生
52	110100	×	○	MMX 命令データ書き込みミス数	発生
53	110101	○	×	誤り分岐予測によるパイプライン・フラッシュ数	発生
53	110101	×	○	WB ステージで解決された誤り分岐予測によるパイプライン・フラッシュ数	発生
54	110110	○	×	MMX 命令でのアライメントずれデータ・メモリ参照	発生

表 7 - 1. パフォーマンス監視イベント　続き）

シリアル番号	エンコーディング	カウンタ0	カウンタ1	パフォーマンス監視イベント	発生回数と持続時間の区別
54	110110	×	○	MMX 命令データ・メモリ読み取り待ちによるパイプライン・ストール	持続
55	110111	○	×	誤って予測された復帰数	発生
55	110111	×	○	(正誤)予測復帰数	発生
56	111000	○	×	MMX 命令乗算ユニット・インターロック	持続
56	111000	×	○	先行操作によるMOVD/MOVQ ストア・ストール	持続
57	111001	○	×	復帰数	発生
57	111001	×	○	RSB オーバーフロー数	発生
58	111010	○	×	BTB 偽エントリ数	発生
58	111010	×	○	分岐しない分岐での BTB ミス予測回数	発生
59	111011	○	×	MMX 命令実行中フル・ライト・バッファによるストール・クロック数	持続
59	111011	×	○	E または M ステート・ラインへの MMX 命令書き込み時ストール	持続

7.1.1 MMX® 命令イベントの解説

以下では、イベント・コード / カウンタを括弧で囲んで示す。

- **U パイプ実行 MMX 命令数 (101011/0)**
  U パイプで実行された MMX 命令の合計数。
- **V パイプ実行 MMX 命令数 (101011/1)**
  V パイプで実行された MMX 命令の合計数。
- **実行 EMMS 命令数 (101101/0)**
  実行された EMMS 命令数をカウントする。

- **ＭＭＸ 命令 /FP 命令間遷移 (101101/1)**

  ＭＭＸ 命令の後の最初の浮動小数点命令、または浮動小数点命令の後の最初のＭＭＸ 命令をカウントする。このカウントに基づいて、FP 状態とＭＭＸ 状態の間の遷移でのペナルティを推定することができる。このカウント値が偶数の場合は、プロセッサがＭＭＸ ステートであることを示している。奇数の場合は、FP 状態であることを示している。
- **キャッシュ不可能メモリ書き込み数 (101110/1)**

  キャッシュ不可能なメモリへの書き込みアクセス数をカウントする。このカウント値には、TLB ミスによる書き込みサイクル数および I/O 書き込みサイクル数が含まれる。BOFF# によって再開されたサイクル数は再度カウントされない。
- **実行飽和ありＭＭＸ 命令数 (101111/0)**

  実際に飽和したかどうかの別を問わず、実行された飽和ありＭＭＸ 命令数をカウントする。飽和ありＭＭＸ 命令は、加算、減算、またはパック操作を実行する。
- **実行飽和ＭＭＸ 命令数 (101111/1)**

  飽和あり算術演算を使用し、そこで少なくとも結果の１つが実際に飽和したＭＭＸ 命令数をカウントする (つまり、４つのダブルワードに対する演算を行ったＭＭＸ 命令が４つの結果のなかの３つで飽和しても、カウンタは１つしかインクリメントされない)。
- **HALT(HLT)状態外サイクル数 (110000/0)**

  プロセッサがＨＡＬＴ(HLT)命令によるアイドル状態でない間のサイクル数をカウントする。このイベントは、「ネットＣＰＩ」の計算に使用する。プロセッサがＨＬＴ 命令を実行している間は、タイム・スタンプ・カウンタ (TSC) はディスエーブルにされていないことに注意されたい。このイベントはカウンタ・コントロールＣＣ0、ＣＣ1 によって制御されるので、このイベントを使用して TSC からは得られないＣＰＬ=3 におけるＣＰＩを計算することができる。
- **ＭＭＸ 命令データ読み取り数 (110001/0)**

  「データ読み取り数」に類似している。ＭＭＸ 命令によるアクセス数だけをカウントする。
- **ＭＭＸ 命令データ読み取りミス数 (110001/1)**

  「データ読み取りミス数」に類似している。ＭＭＸ 命令によるアクセス数だけをカウントする。
- **浮動小数点ストール数 (110010/0)**

  浮動小数点凍結のためにパイプがストールしている間のクロック数をカウントする。
- **分岐した分岐数 (110010/1)**

  分岐した分岐数をカウントする。
- **D1 欠乏および FIFO 空 (110011/0)、D1 欠乏および FIFO 内 1 命令のみ (110011/1)**

  FIFO バッファにある命令が使用可能な場合は、D1 ステージは１クロック当たり命令を１つも発行しないか、あるいは命令を１つまたは２つ発行することができる。最初のイベントは、FIFO バッファが空のために、D1 ステージがどのような命令も発行できない回数をカウントする。２番目のイベントは、FIFO バッファに命令が１つだけあるために、命令を１つだけ発行する回数をカウントする。その他の２つのイベント、U パイプ実行命令数 (010110) および V パイプ実行命令数 (010110) と組み合わせると、２番目のイベントではペアリング規則のために２つの命令の発行が妨げられた回数を計算できる。

- **MMX 命令データ書き込み数 (110001/1)**

  「データ書き込み数」に類似している。MMX 命令によるアクセス数だけをカウントする。
- **MMX 命令データ書き込みミス数 (110100/1)**

  「データ書き込みミス数」に類似している。MMX 命令によるアクセス数だけをカウントする。
- **誤り分岐予測によるパイプライン・フラッシュ数 (110101/0)、WB ステージで解決された誤り分岐予測によるパイプライン・フラッシュ数 (110101/1)**

  パイプライン内のフローが正しく進行しなかった分岐によるすべてのパイプライン・フラッシュをカウントする。そのようなパイプライン・フラッシュには、分岐が BTB 内に存在しなかった場合、分岐が BTB 内に存在したがその予測がミスされた場合、および分岐が正しく予測されたが分岐先アドレスが誤っていた場合が含まれる。分岐は、実行 (E) ステージかライトバック (WB) ステージで解決される。後者の場合は、予測誤りのペナルティが 1 クロックだけ大きくなる。上記の最初のイベントは、誤って予測され、E ステージか WB ステージで解決された分岐数をカウントする。2 番目のイベントは、誤って予測され、WB ステージで解決された分岐数をカウントする。これら 2 つのカウントの相違は、E ステージで解決された分岐数である。
- **MMX 命令でのアライメントずれデータ・メモリ参照 (110110/0)**

  「アライメントずれデータ・メモリ参照」に類似している。MMX 命令によるアクセス数だけをカウントする。
- **MMX 命令データ・メモリ読み取り待ちによるパイプライン・ストール (110110/1)**

  「データ・メモリ読み取り待ちによるパイプライン・ストール」に類似している。MMX 命令によるアクセス数だけをカウントする。
- **誤って予測されたまたは予測なし復帰数 (110111/0)**

  誤って予測されたか、またはまったく予測されなかった実際の復帰数。このカウント値は、実行された合計復帰数と正しく予測された復帰数との差である。RET 命令だけがカウントされる (つまり、IRET 命令はカウントされない)。
- **(正誤) 予測復帰数 (110111/1)**

  予測が行われた実際の復帰数。RET 命令だけがカウントされる (つまり、IRET 命令はカウントされない)。
- **MMX 命令乗算ユニット・インターロック (111000/0)**

  前の MMX 乗算命令のデスティネーションの用意がまだできていないために生じるパイプ・ストールの間のクロック数をカウントする。他にストールの原因がある場合は、カウンタはインクリメントされない。乗算インターロックが発生するたびに、このイベントは 2 回 (ストールされた命令が乗算後の次のクロックで現れた場合) または 1 回だけ (ストールされた命令が乗算の 2 クロック後に現れた場合) カウントされることがある。
- **先行操作による MOVD/MOVQ ストア・ストール (111000/1)**

  MOVD/MOVQ ストア命令で使用されたデスティネーションに対する直前の MMX 操作のために、D2 ステージでストアがストールしている間のクロック数。

- **復帰数 (111001/0)**

  実際に実行された復帰数。RET 命令だけがカウントされる (つまり、IRET 命令はカウントされない)。RET 命令で生じた例外もすべてこのカウンタを更新する。
- **RSB オーバーフロー数 (111001/1)**

  リターン・スタック・バッファ(RSB)にコール・アドレスを入れることができない回数をカウントする。
- **BTB 偽エントリ数 (111010/0)**

  分岐ターゲット・バッファの偽のエントリ数をカウントする。偽のエントリは、誤り予測でなく予測ミスの原因になる。
- **分岐しない分岐での BTB ミス予測回数 (111010/1)**

  BTB が分岐しない分岐を分岐すると予測した回数をカウントする。
- **MMX 命令実行中フル・ライト・バッファによるストール・クロック数 (111011/0)**

  「フル・ライト・バッファによるストール・クロック数」に類似している。MMX 命令によるアクセス数だけをカウントする。
- **E または M ステート・ラインへの MMX 命令書き込み時ストール (111011/1)**

  「E または M ステート・ラインへの書き込み時ストール」に類似している。MMX 命令によるアクセス数だけをカウントする。

7.2 Pentium® Pro および Pentium II のパフォーマンス監視イベント

本節では、Pentium Pro および Pentium II プロセッサのカウンタについて説明する。表 7-2 に、パフォーマンス監視用のカウンタでカウントでき、RDPMC 命令で読み取れるイベントの一覧を示す。

表の各欄の意味は、以下のとおりである。

- ユニット欄は、イベントを発生するマイクロアーキテクチャ・ユニットまたはバス・ユニットを示す。
- イベント番号欄は、イベント識別用の 16 進数を示す。
- ニーモニック・イベント名欄は、イベントの名前を示す。
- ユニット・マスク欄は、(存在する場合) 必要なユニット・マスクを示す。
- 説明欄では、イベントについて説明する。
- 注釈欄は、イベントに関する追加情報を示す。

これらのパフォーマンス監視イベントは、パフォーマンス・チューニングの目安として利用されることを目的としている。報告されるカウンタ値は、絶対に正確であるとはいえないので、チューニングの相対的な目安として利用されたい。相違の事実が確認されているものについては、その旨明記している。パフォーマンス監視イベントは、すべて Pentium Pro プロセッサ・ファミリに固有のものであり、将来のバージョンのプロセッサでもサポートされる保証はない。表に載っていないパフォーマンス監視イベントのエンコーディングはすべて予約されており、それらを使用した場合はカウンタの結果は不定である。

この表の特定の項目に関する注記については、表の終わりを参照されたい。

表 7 - 2. パフォーマンス監視カウンタ

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
データ・キャッシュ・ユニット (DCU)	43H	DATA_MEM_REFS	00H	任意のメモリ・タイプからのすべてのロード。任意のメモリ・タイプへのすべてのストア。分割の各部は別々にカウントされる。 **注記 :** 80 ビット浮動小数点アクセスは、16 ビットの指数ロードと64ビットの仮数ロードに分解されるので、2 重にカウントされる。 メモリ・アクセスは、実際に行われたときしかカウントされない。たとえば、ロードが前のキャッシュ・ミスが未解決のために同じアドレスに詰め込まれていて、そのロードが最終的に行われた場合、そのロードは 1 回しかカウントされない。 I/O アクセスも、その他のメモリ以外のアクセスも含まない。	
	45H	DCU_LINES_IN	00H	DCU 内に割り当てられた合計ライン数。	
	46H	DCU_M_LINES_IN	00H	DCU 内に割り当てられた修正されたステート・ライン数。	
	47H	DCU_M_LINES_OUT	00H	DCU から立ち退かされた修正されたステート・ライン数。これには、外部スヌープの結果としての立ち退き、内部介入、または自然置換アルゴリズムが含まれる。	
	48H	DCU_MISS_OUT_STANDING	00H	DCU ミスが未解決である間の重み付けサイクル数。いつでも、未解決のキャッシュ・ミスの数だけインクリメントされる。キャッシュ可能な読み取り要求だけがカウントの対象になる。キャッシュ不可能な要求は除外される。所有権のための読み取りは、ライン・フィル、無効化、ストアとしてもカウントされる。	L2 もミスしたアクセスは、2サイクルだけショート変更される (つまり、カウントが N サイクルであれば、N+2 サイクルになる)。同じキャッシュ・ラインへの以降のロードは、追加してカウントされない。カウント値は厳密ではないが、それでも有用である。

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
命令フェッチ・ユニット (IFU)	80H	IFU_IFETCH	00H	キャッシュ可能なおよびキャッシュ不可能なフェッチ数を合わせた命令フェッチ数。UC フェッチ数を含む。	フェッチされた各キャッシュ可能ラインにつき1、およびフェッチされた各キャッシュ不使用命令につき 1、それぞれインクリメントされる。
	81H	IFU_IFETCH_ MISS	00H	命令フェッチ・ミス数。IFU をヒットしない、つまりメモリ要求を生じるすべての命令フェッチ数。UC アクセスを含む。	
	85H	ITLB_MISS	00H	ITLB ミス数。	
	86H	IFU_MEM_ STALL	00H	なんらかの理由による命令フェッチ・ストールのサイクル数。IFU キャッシュ・ミス、ITLB ミス、ITLB フォルト、およびその他のマイナーなストールを含む。	
	87H	ILD_STALL	00H	プロセッサ・パイプラインの命令長デコーダ・ステージ・ストールのサイクル数。	
L2 キャッシュ	28H	L2_IFETCH	MESI 0FH	L2 の命令フェッチ数。このイベントは、通常の命令フェッチがL2によって受け取られたことを示す。カウントは、L2キャッシュ可能命令フェッチ数だけを含む。つまり、UC 命令フェッチ数は含まない。ITLB ミス・アクセス数も含まない。	
	29H	L2_LD	MESI 0FH	L2 データ・ロード数。このイベントは、通常のロックされていないメモリ・ロード・アクセスがL2によって受け取られたことを示す。カウントは、L2 キャッシュ可能メモリ・アクセスだけを含む。つまり、I/O アクセス、その他の メモリ以外のアクセス、UC/WT メモリ・アクセスのようなメモリ・アクセスは含まない。L2 キャッシュ可能なTLB ミス・メモリ・アクセスは含む。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
L2 キャッシュ (続き)	2AH	L2_ST	MESI 0FH	L2 データ・ストア数。このイベントは、通常のロックされていないメモリ・ストア・アクセスがL2によって受け取られたことを示す。特に、DCUが所有権のための読み取り要求をL2に送ったことを示す。このイベントは、DCUがL2に送ったInvalid to Modified 要求も含む。カウントは、L2 キャッシュ可能メモリ・ストア・アクセスだけを含む。つまり、I/O アクセス、その他のメモリ以外のアクセス、UC/WT メモリ・アクセスのようなメモリ・アクセスは含まない。L2 キャッシュ可能なTLB ミス・メモリ・アクセスは含む。	
	24H	L2_LINES_IN	00H	L2に割り当てられたライン数。	
	26H	L2_LINES_OUT	00H	なんらかの理由でL2から削除されたライン数。	
	25H	L2_M_LINES_ INM	00H	L2に割り当てられた修正されたステート・ライン数。	
	27H	L2_M_LINES_ OUTM	00H	なんらかの理由でL2から削除された修正されたステート・ライン数。	
	2EH	L2_RQSTS	MESI 0FH	すべての L2 要求の合計数。	
	21H	L2_ADS	00H	L2 アドレス・ストア数。	
	22H	L2_DBUS_BUSY	00H	L2 キャッシュ・データ・バスがビジーであった間のサイクル数。	
	23H	L2_DBUS_BUSY_ RD	00H	データ・バスがL2 からプロセッサへの読み取りデータ転送のためにビジーであった間のサイクル数。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
外部バス論理 (EBL)(2)	62H	BUS_DRDY_CLOCKS	00H (Self) 20H (Any)	DRDY#がアサートされていた間のクロック数。基本的に、外部システム・データ・バスの利用率。	ユニット・マスク = 00H は、プロセッサが DRDY をドライブしていた間のバス・クロックをカウントする。ユニット・マスク = 20H は、任意のエージェントが DRDY をドライブしていた間のプロセッサ・クロックをカウント・インする。
	63H	BUS_LOCK_CLOCKS	00H (Self) 20H (Any)	外部システム・バスで LOCK#がアサートされていた間のクロック数。	常にプロセッサ・クロックをカウント・インする。
	60H	BUS_REQ_OUTSTANDING	00H (Self)	未解決のバス要求数。このカウンタは、与えられた任意のサイクルにおける未解決のキャッシュ可能読み取りバス要求数だけインクリメントされる。	DCU フル・ライン・キャッシュ可能読み取りだけをカウントする。所有権のための読み取り、書き込み、命令フェッチ、その他のいずれもカウントしない。『バス完了 (最後に受け取られるデータのまとまり) 待ち」をカウントする。
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	バス・バースト読み取りトランザクション数。	
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	完了した所有権のためのバス読み取りトランザクション数。	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	完了したバス・ライトバック・トランザクション数。	
	68H	BUS_TRAN_IFETCH	00H (Self) 20H (Any)	完了したバス命令フェッチ・トランザクション数。	
	69H	BUS_TRAN_INVAL	00H (Self) 20H (Any)	完了したバス無効化トランザクション数。	
	6AH	BUS_TRAN_PWR	00H (Self) 20H (Any)	完了したバス・パーシャル書き込みトランザクション数。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
外部バス論理 (EBL)(2) (続き)	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	完了したバス・パーシャル・トランザクション数。	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	完了したバス I/O トランザクション数。	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	完了したバス据え置きトランザクション数。	
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	完了したバス・バースト・トランザクション数。	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	すべての完了したバス・トランザクション数。最少アドレス・バス占有率を知ることにより、アドレス・バス利用率を計算できる。特殊サイクルなどを含む。	
	6FH	BUS_TRAN_MEM	00H (Self) 20H (Any)	完了したメモリ・トランザクション数。	
	64H	BUS_DATA_RCV	00H (Self)	このプロセッサがデータを受け取っていた間のバス・クロック・サイクル数。	
	61H	BUS_BNR_DRV	00H (Self)	このプロセッサがBNR ピンをドライブしていた間のバス・クロック・サイクル数。	
	7AH	BUS_HIT_DRV	00H (Self)	このプロセッサが HIT ピンをドライブしていた間のバス・クロック・サイクル数。	スヌープ・ストールによるサイクルを含む。
	7BH	BUS_HITM_DRV	00H (Self)	このプロセッサが HITM ピンをドライブしていた間のバス・クロック・サイクル数。	スヌープ・ストールによるサイクルを含む。
	7EH	BUS_SNOOP_ STALL	00H (Self)	バスがスヌープ・ストールしていた間のクロック・サイクル数。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
浮動小数点ユニット	C1H	FLOPS	00H	リタイアした浮動小数点演算数。トラップまたはアシスタンスを生じる浮動小数点演算は除外する。アシスト・ハンドラによって実行される浮動小数点演算を含む。 超越演算命令などの複素浮動小数点命令の内部サブオペレーションを含む。浮動小数点ロードおよびストアは除外する。	カウンタ 0 のみ。
	10H	FP_COMP_OPS_EXE	00H	実行された浮動小数点演算　数。FADD、FSUB、FCOM、FMUL、整数 MUL および　MUL、FDIV、FPREM、FSQRTS、整数 DIV および IDIV の数。サイクル数ではなく演算数をカウントすることに注意されたい。このイベントは、超越演算フローの中間に使用されている FADD を独立した FADD 命令と区別しない。	カウンタ 0 のみ。
	11H	FP_ASSIST	00H	マイクロコードによって処理された浮動小数点例外ケース数。	カウンタ 1 のみ。このイベントは、推論的な実行によるカウントを含む。
	12H	MUL	00H	乗算数。 注 - 整数および FP 乗算を含む。	カウンタ 1 のみ。このイベントは、推論的な実行によるカウントを含む。
	13H	DIV	00H	除算数。 注 - 整数および FP 除算を含む。	カウンタ 1 のみ。このイベントは、推論的な実行によるカウントを含む。
	14H	CYCLES_DIV_BUSY	00H	除算器がビジーであり、新しい除算を受け付けられなかった間のサイクル数。 注 - 整数および FP 除算、FPREM、FPSQRT などを含む。	カウンタ 0 のみ。このイベントは、推論的な実行によるカウントを含む。

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
メモリ順序化	03H	LD_BLOCKS	00H	ストア・バッファ・ブロック数。アドレスが未知である先行ストア、アドレスが衝突することがわかっているがデータは未知である先行ストア、およびロードと衝突するがロードと完全にオーバーラップする先行ストアによって生じるカウントを含む。	
	04H	SB_DRAINS	00H	ストア・バッファ・ドレイン・サイクル数。ストア・バッファがドレイン中の各サイクルごとにインクリメントされる。ドレインは、CPUID のような逐次化操作、XCHG のような同期化操作、割り込み認識、その他のキャッシュ・フラッシュなどの条件によって生じる。	
	05H	MISALIGN_MEM_REF	00H	アライメントずれデータ・メモリ参照数。Pentium® Pro のロード、ストアいずれかのパイプラインがアライメントずれ µop をディスパッチするサイクルごとに 1 インクリメントされる。カウントは、参照の前半分または後半分で、あるいは参照がブロックされた、詰め込まれた、またはミスした場合に行われる。 このコンテキストでは、アライメントずれとは 64 ビット境界にまたがることを意味する。	MISALIGN_MEM_REF は、真のアライメントずれメモリ参照数の近似値にすぎないことに注意されたい。返される値は、およそアライメントずれメモリ・アクセス数、すなわち問題の大きさに比例する。
命令デコード/リタイア	C0H	INST_RETIRED	00H	リタイアした合計命令数。	
	C2H	UOPS_RETIRED	00H	リタイアした合計 µop 数。	
	D0H	INST_DECODER	00H	デコードされた合計命令数。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
割り込み	C8H	HW_INT_RX	00H	受け取られた合計ハードウェア割り込み数。	
	C6H	CYCLES_INT_ MASKED	00H	割り込みがディスエーブルにされていた間のプロセッサ ・サイクル数。	
	C7H	CYCLES_INT_ PENDING_AND_ MASKED	00H	割り込みがディスエーブルにされていて、かつ割り込みが処理待ちしていた間の合計プロセッサ ・サイクル数。	
分岐	C4H	BR_INST_ RETIRED	00H	リタイアした合計分岐命令数。	
	C5H	BR_MISS_PRED_ RETIRED	00H	リタイア ・ポイントに達した分岐誤り予測の合計数。分岐しない条件付き分岐を含む。	
	C9H	BR_TAKEN_ RETIRED	00H	リタイアした分岐した分岐の合計数。	
	CAH	BR_MISS_PRED_ TAKEN_RET	00H	リタイア ・ポイントに達した、分岐すると予測されたが予測が誤っていた分岐の合計数。分岐したときのみ条件付き分岐を含む。	
	E0H	BR_INST_ DECODED	00H	デコードされた分岐命令の合計数。	
	E2H	TB_MISSES	00H	BTB が予測しなかった分岐の合計数。	
	E4H	BR_BOGUS	00H	予測したが、実際には分岐しなかった分岐予測の合計数。	
	E6H	BACLEARS	00H	BACLEAR がアサートされた合計回数。これは、デコーダによって静的分岐予測が行われた回数である。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
ストール	A2H	RESOURCE_ STALLS	00H	リソース関係のストールが生じる各サイクルごとに 1 インクリメントされる。レジスタ・リネーミング・バッファ・エントリ、メモリ・バッファ・エントリを含む。バス・キュー・フル、キャッシュ・ミス過多などによるストールは含まれない。リソース関係ストールの他に、このイベントは一部の他のイベントをカウントする。 分岐誤り予測の回復中に生じるストール、たとえば、誤り予測分岐のリタイアが遅延した場合に生じるストール、および同期操作によるストア・バッファのドレイン中に生じるストールを含む。	
	D2H	PARTIAL_RAT_ STALLS	00H	パーシャル・ストールの間のサイクル数またはイベント数。 注 - フラグ・パーシャル・ストールを含む。	
セグメント・レジスタ・ロード	06H	SEGMENT_REG_ LOADS	00H	セグメント・レジスタ・ロード数。	
クロック	79H	CPU_CLK_ UNHALTED	00H	プロセッサが停止されていなかった間のサイクル数。	
MMX® テクノロジ命令イベント					
MMX 命令実行	B0H	MMX_INSTR_ EXEC	00H	実行された MMX 命令数。	
飽和あり MMX 命令実行	B1H	MMX_SAT_ INSTR_EXEC	00H	実行された飽和あり MMX 命令数。	
MMX µop 実行	B2H	MMX_UOPS_ EXEC	0FH	実行された MMX µop 数。	

表 7 - 2. パフォーマンス監視カウンタ　続き）

ユニット	イベント番号	ニーモニック・イベント名	ユニット・マスク	説明	注釈
MMX 命令実行	B3H	MMX_INSTR_TYPE_EXEC	01H 02H 04H 08H 10H 20H	実行されたMMXパックド乗算命令数。 実行された MMX パックド・シフト命令数。 実行されたMMXパックド操作命令数。 実行されたMMXアンパック操作命令数。 実行されたMMXパックド論理演算命令数。 実行されたMMXパックド算術演算命令数。	
MMX 遷移	CCH	FP_MMX_TRANS	00H 01H	MMX 命令から FP 命令への遷移数。 FP 命令から MMX 命令への遷移数。	
MMX アシスタンス	CDH	MMX_ASSIST	00H	MMX アシスタンス数。	MMXアシスタンス数は実行された EMMS 命令の数である。
リタイア済み MMX 命令	CEH	MMX_INSTR_RET	00H	リタイアした MMX 命令数。	
セグメント・レジスタ・リネーミング・ストール	D4H	SEG_RENAME_STALLS	01H 02H 04H 08H 0FH	セグメント・レジスタ ES セグメント・レジスタ DS セグメント・レジスタ FS セグメント・レジスタ FS セグメント・レジスタ ES + DS + FS + GS	
リネームされたセグメント・レジスタ	D5H	SEG_REG_RENAMES	01H 02H 04H 08H 0FH	セグメント・レジスタ ES セグメント・レジスタ DS セグメント・レジスタ FS セグメント・レジスタ FS セグメント・レジスタ ES + DS + FS + GS	
リネームされたリタイア済みセグメント・レジスタ	D6H	RET_SEG_RENAMES	00H	リタイアしたセグメント・レジスタ・リネーミング・イベント数。	

注記：

1. この注番号が施してあるいくつかのL2キャッシュ・イベントは、PerfEvtSel0およびPerfEvtSel1レジスタのユニット・マスク(UMSK)フィールドを使用してさらに限定することができる。ユニット・マスク・フィールドの下位4ビットは、L2イベントと組み合わされて、関与した1つ以上のキャッシュ状態を示す。Pentium® Proプロセッサ・ファミリでは、"MESI"プロトコルを使用してキャッシュ状態を識別し、結果としてユニット・マスク・フィールドの各ビットは4つの状態の1つを表す。つまり、UMSK[3]=M (8h)状態、UMSK[2]=E (4h)状態、UMSK[1]=S (2h)状態、およびUMSK[0]=I (1h)状態である。すべての状態のデータを収集するには、UMSK[3:0]=MESI (Fh)を使用する。UMSK=0hを指定した場合、つまり該当するイベントに対しては、何もカウントされない。
2. この注番号が施してある場合を除き、外部バス論理(EBL)イベントはすべて、PerfEvtSel0およびPerfEvtSel1レジスタのユニット・マスク(UMSK)フィールドを使用してさらに限定することができる。UMSKフィールドのビット5は、EBLイベントと組み合わされて、プロセッサ自身が生成したトランザクション(UMSK[5]=0)、またはバス上の任意のプロセッサが生成したトランザクション(UMSK[5]=1)のどちらをカウントするかを示す。

7.3 RDPMC命令

RDPMC (プロセッサ・モニタ・カウンタ読み取り)命令は、CR4レジスタのビット#8 (CR4.PCE)がセットされている場合に、CPL=3でパフォーマンス監視カウンタを読み取る。これは、RDTSC (タイム・スタンプ・カウンタ読み取り)命令に類似しているが、この命令の場合は、CR4のタイム・スタンプ・ディスエーブル・ビット(CR4.TSD)がディスエーブルにされていない場合にCLP=3でイネーブルにされる。パフォーマンス監視用の制御およびイベント選択レジスタ(CESR)へのアクセスは、CPL=3では不可能である。

7.3.1 命令の仕様

オペコード： 0F 33

説明： ECXによって指示されるイベント・モニタ・カウンタをEDX:EAXに読み込む。

操作： EDX:EAX ← イベント・カウンタ[ECX]

ECXの値(0または1)は、プロセッサの2つの40ビット・イベント・カウンタの1つを指定する。EDXには上位32ビットがロードされ、EAXには下位32ビットがロードされる。

```
IF CR4.PCE = 0 AND CPL <> 0 THEN # GP(0)
IF ECX = 0 THEN EDX:EAX := PerfCntr0
IF ECX = 1 THEN EDX:EAX := PerfCntr1
ELSE #GP(0)
END IF
```

保護モード例外および実アドレス・モード例外：

ECXが有効なカウンタ(0または1)を指定していない場合は#GP(0)。

CPL<>0でRDPMCを使用し、かつCR4.PCE=0の場合は#GP(0)。

備考：

16ビット・コード - RDPMCは16ビット・コード、VMモードで動作するが、32ビットの結果を生じる。この命令は、全ECXビットのインデックスを使用する。

intel

A

整数ペアリング表

付録 A
整数ペアリング表

この付録の整数命令ペア リング表のペア リング欄には、以下の略語が使用 されている。
NP - ペア リング不可能、U パイプで実行
PU - U パイプに発行 された場合ペア リング可能
PV - V パイプに発行 された場合ペア リング可能
UV - どちらのパイプでもペア リング可能
I/O 命令は、ペア リングすることはできない。

A.1 整数命令ペアリング表

表 A-1. 整数命令ペアリング

命令	フォーマット	ペアリング
AAA - 加算後に ASC II 調整		NP
AAD - 除算前に AX を ASC II 調整		NP
AAM - 乗算後に AX を ASC II 調整		NP
AAS - 減算後に AL を ASC II 調整		NP
ADC - キャリーあり加算		PU
ADD - 加算		UV
AND - 論理積		UV
ARPL - セレクタの RPL フィールド調整		NP
BOUND - 境界に対する配列チェック		NP
BSF - 前方ビット・スキャン		NP
BSR - 後方ビット・スキャン		NP
BSWAP - バイト・スワップ		NP
BT - ビット・テスト		NP
BTC - ビット・テストと補数変換		NP
BTR - ビット・テストとリセット		NP
BTS - ビット・テストとセット		NP

表 A-1. 整数命令ペアリング　続き)

命令	フォーマット	ペアリング
CALL - プロシージャ・コール (同一セグメント内)		
直接コール	1110 1000 : フル・ディスプレースメント	PV
レジスタ間接コール	1111 1111 : 11010 reg	NP
メモリ間接コール	1111 1111 : mod 010 r/m	NP
CALL - プロシージャ・コール (他セグメント)		NP
CBW - バイトからワードへの変換 CWDE - ワードからダブルワードへの変換		NP
CLC - キャリー・フラグをクリア		NP
CLD - 方向フラグをクリア		NP
CLI - 割り込みフラグをクリア		NP
CLTS - CR0 内タスク・スイッチド・フラグをクリア		NP
CMC - キャリー・フラグ補数変換		NP
CMP - 2 つのオペランドを比較		UV
CMPS/CMPSB/CMPSW/CMPSD - ストリング・オペランドを比較		NP
CMPXCHG - 比較と交換		NP
CMPXCHG8B - 8 バイト比較と交換		NP
CWD - ワードからダブルワードへの変換 CDQ - ダブルワードからクワッドワードへの変換		NP
DAA - 加算後に AL を 10 進調整		NP
DAS - 減算後に AL を 10 進調整		NP
DEC - 1 デクリメント		UV
DIV - 符号なし除算		NP
ENTER - プロシージャ・パラメータ用スタック・フレーム作成		NP
HLT - 停止		
IDIV - 符号付き除算		NP
IMUL - 符号付き乗算		NP
INC - 1 インクリメント		UV
INTn - 割り込みタイプ n		NP
INT - シングル・ステップ割り込み 3		NP
INTO - オーバーフロー時割り込み 4		NP

表 A-1. 整数命令ペアリング　続き)

命令	フォーマット	ペアリング
INVD - キャッシュ無効化		NP
INVLPG - TLB エントリ無効化		NP
IRET/IRETD - 割り込み復帰		NP
Jcc - 条件成立時ジャンプ		PV
JCXZ/JECXZ - CX/ECX ゼロ時ジャンプ		NP
JMP - 無条件ジャンプ (同一セグメント内)		
ショート・ジャンプ	1110 1011: 8ビット・ディスプレースメント	PV
直接ジャンプ	1110 1001: フル・ディスプレースメント	PV
レジスタ間接ジャンプ	1111 1111 : 11 100 reg	NP
メモリ間接ジャンプ	1111 1111 : mod 100 r/m	NP
JMP - 無条件ジャンプ (他セグメント)		NP
LAHF - AH レジスタにフラグをロード		NP
LAR - アクセス権バイトをロード		NP
LDS - DS にポインタをロード		NP
LEA - 実効アドレスをロード		UV
LEAVE - 高水準プロシージャ終了		NP
LES - ES にポインタをロード		NP
LFS - FS にポインタをロード		NP
LGDT - グローバル・ディスクリプタ・テーブル・レジスタをロード		NP
LGS - GS にポインタをロード		NP
LIDT - 割り込みディスクリプタ・テーブル・レジスタをロード		NP
LLDT - ローカル・ディスクリプタ・テーブル・レジスタをロード		NP
LMSW - マシン・ステータス・ワードをロード		NP
LOCK - LOCK# 信号アサート・プリフィックス		NP
LODS/LODSB/LODSW/LODSD - ストリング・オペランドをロード		NP
LOOP - ループ・カウント		NP
LOOPZ/LOOPE - ゼロ / イコール中のループ・カウント		NP
LOOPNZ/LOOPNE - 非ゼロ / 非イコール中のループ・カウント		NP

表 A-1. 整数命令ペアリング　続き）

命令	フォーマット	ペアリング
LSL - セグメント範囲をロード		NP
LSS - SS にポインタをロード	0000 1111 : 1011 0010 : mod reg r/m	NP
LTR - タスク・レジスタをロード		NP
MOV - データを移動		UV
MOV - 制御レジスタとのデータ移動		NP
MOV - デバッグ・レジスタとのデータ移動		NP
MOV - セグメント・レジスタとのデータ移動		NP
MOVS/MOVSB/MOVSW/MOVSD - ストリング / ストリング間データ移動		NP
MOVSX - 符号拡張して移動		NP
MOVZX - ゼロ拡張して移動		NP
MUL - AL、AX、または EAX の符号なし乗算		NP
NEG - 2 の補数のネゲーション		NP
NOP - オペレーションなし	1001 0000	UV
NOT - 1 の補数のネゲーション		NP
OR - 包含的論理和		UV
POP - スタックから 1 ワードをポップ		
レジスタ・ポップ	1000 1111 : 11 000 reg	UV
または	0101 1 reg	UV
メモリ・ポップ	1000 1111 : mod 000 r/m	NP
POP - スタックからセグメント・レジスタをポップ		NP
POPA/POPAD - すべての汎用レジスタをポップ		NP
POPF/POPFD - FLAGS または EFLAGS レジスタにスタックをポップ		NP
PUSH - スタックにオペランドをプッシュ		
レジスタ・プッシュ	1111 1111 : 11 110 reg	UV
または	0101 0 reg	UV
メモリ・プッシュ	1111 1111 : mod 110 r/m	NP
即値プッシュ	0110 10s0 : 即値データ	UV
PUSH - スタックにセグメント・レジスタをプッシュ		NP
PUSHA/PUSHAD - すべての汎用レジスタをプッシュ		NP

表 A-1. 整数命令ペアリング　続き)

命令	フォーマット	ペアリング
PUSHF/PUSHFD - スタックにフラグ・レジスタをプッシュ		NP
RCL - キャリー経由左にローテート		
レジスタを 1 だけローテート	1101 000w : 11 010 reg	PU
メモリを 1 だけローテート	1101 000w : mod 010 r/m	PU
レジスタを CL だけローテート	1101 001w : 11 010 reg	NP
メモリを CL だけローテート	1101 001w : mod 010 r/m	NP
レジスタを即値カウントだけローテート	1100 000w : 11 010 reg : imm8 データ	PU
メモリを即値カウントだけローテート	1100 000w : mod 010 r/m : imm8 データ	PU
RCR - キャリー経由右にローテート		
レジスタを 1 だけローテート	1101 000w : 11 011 reg	PU
メモリを 1 だけローテート	1101 000w : mod 011 r/m	PU
レジスタを CL だけローテート	1101 001w : 11 011 reg	NP
メモリを CL だけローテート	1101 001w : mod 011 r/m	NP
レジスタを即値カウントだけローテート	1100 000w : 11 011 reg : imm8 データ	PU
メモリを即値カウントだけローテート	1100 000w : mod 011 r/m : imm8 データ	PU
RDMSR - モデル固有レジスタから読み取り		NP
REP LODS - ストリングをロード		NP
REP MOVS - ストリングを移動		NP
REP STOS - ストリングをストア		NP
REPE CMPS - ストリングを比較 (不一致検出)		NP
REPE SCAS - ストリングをスキャン(非AL/AX/AEX検出)		NP
REPNE CMPS - ストリングを比較 (一致検出)		NP
REPNE SCAS - ストリングをスキャン (AL/AX/AEX 検出)		NP
RET - プロシージャから復帰 (同一セグメントへ)		NP
RET - プロシージャから復帰 (他セグメントへ)		NP
ROL - (キャリー非経由)左にローテート		
レジスタを 1 だけローテート	1101 000w : 11 000 reg	PU
メモリを 1 だけローテート	1101 000w : mod 000 r/m	PU
レジスタを CL だけローテート	1101 001w : 11 000 reg	NP
メモリを CL だけローテート	1101 001w : mod 000 r/m	NP

表 A-1. 整数命令ペアリング　続き)

命令	フォーマット	ペアリング
レジスタを即値カウントだけローテート	1100 000w : 11 000 reg : imm8 データ	PU
メモリを即値カウントだけローテート	1100 000w : mod 000 r/m : imm8 データ	PU
ROR - (キャリー非経由)右にローテート		
レジスタを 1 だけローテート	1101 000w : 11 001 reg	PU
メモリを 1 だけローテート	1101 000w : mod 001 r/m	PU
レジスタを CL だけローテート	1101 001w : 11 001 reg	NP
メモリを CL だけローテート	1101 001w : mod 001 r/m	NP
レジスタを即値カウントだけローテート	1100 000w : 11 001 reg : imm8 データ	PU
メモリを即値カウントだけローテート	1100 000w : mod 001 r/m : imm8 データ	PU
RSM - システム ・マネージメント ・モードから再開		NP
SAHF - フラグに AH をストア		NP
SAL - 算術左シフト - SHL と同じ命令		
SAR - 算術右シフト		
レジスタを 1 だけシフト	1101 000w : 11 111 reg	PU
メモリを 1 だけシフト	1101 000w : mod 111 r/m	PU
レジスタを CL だけシフト	1101 001w : 11 111 reg	NP
メモリを CL だけシフト	1101 001w : mod 111 r/m	NP
レジスタを即値カウントだけシフト	1100 000w : 11 111 reg : imm8 データ	PU
メモリを即値カウントだけシフト	1100 000w : mod 111 r/m : imm8 データ	PU
SBB - ボローあり整数減算		PU
SCAS/SCASB/SCASW/SCASD - ストリングをスキャン		NP
SETcc - 条件付きバイト設定		NP
SGDT - グローバル ・ディスクリプタ ・テーブル ・レジスタをストア		NP
SHL - 左シフト		
レジスタを 1 だけシフト	1101 000w : 11 100 reg	PU
メモリを 1 だけシフト	1101 000w : mod 100 r/m	PU
レジスタを CL だけシフト	1101 001w : 11 100 reg	NP
メモリを CL だけシフト	1101 001w : mod 100 r/m	NP
レジスタを即値カウントだけシフト	1100 000w : 11 100 reg : imm8 データ	PU

表 A-1. 整数命令ペアリング　続き）

命令	フォーマット	ペアリング
メモリを即値カウントだけシフト	1100 000w : mod 100 r/m : imm8 データ	PU
SHLD - 倍精度左シフト		
レジスタを即値カウントだけシフト	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8	NP
メモリを即値カウントだけシフト	0000 1111 : 1010 0100 : mod reg r/m : imm8	NP
レジスタを CL だけシフト	0000 1111 : 1010 0101 : 11 reg2 reg1	NP
メモリを CL だけシフト	0000 1111 : 1010 0101 : mod reg r/m	NP
SHR - 右シフト		
レジスタを 1 だけシフト	1101 000w : 11 101 reg	PU
メモリを 1 だけシフト	1101 000w : mod 101 r/m	PU
レジスタを CL だけシフト	1101 001w : 11 101 reg	NP
メモリを CL だけシフト	1101 001w : mod 101 r/m	NP
レジスタを即値カウントだけシフト	1100 000w : 11 101 reg : imm8 データ	PU
メモリを即値カウントだけシフト	1100 000w : mod 101 r/m : imm8 データ	PU
SHRD - 倍精度右シフト		
レジスタを即値カウントだけシフト	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8	NP
メモリを即値カウントだけシフト	0000 1111 : 1010 1100 : mod reg r/m : imm8	NP
レジスタを CL だけシフト	0000 1111 : 1010 1101 : 11 reg2 reg1	NP
メモリを CL だけシフト	0000 1111 : 1010 1101 : mod reg r/m	NP
SIDT - 割り込みディスクリプタ・テーブル・レジスタをストア		NP
SLDT - ローカル・ディスクリプタ・テーブル・レジスタをストア		NP
SMSW - マシン・ステータス・ワードをストア		NP
STC - キャリー・フラグをセット		NP
STD - 方向フラグをセット		NP
STI- 割り込みフラグをセット		
STOS/STOSB/STOSW/STOSD - ストリング・データをストア		NP
STR - タスク・レジスタをストア		NP

表 A-1. 整数命令ペアリング　続き）

命令	フォーマット	ペアリング
SUB - 整数減算		UV
TEST - 論理比較		
レジスタ1 とレジスタ2 を比較	1000 010w : 11reg1 reg2	UV
メモリとレジスタを比較	1000 010w : mod reg r/m	UV
即値とレジスタを比較	1111 011w : 11 000 reg : 即値データ	NP
即値とアキュムレータを比較	1010 100w : 即値データ	UV
即値とメモリを比較	1111 011w : mod 000 r/m : 即値データ	NP
VERR - 読み取りのためセグメントを検証		NP
VERW - 書き込みのためセグメントを検証		NP
WAIT - 待機	1001 1011	NP
WBINVD - ライトバックとデータ・キャッシュ無効化		NP
WRMSR - モデル固有レジスタへの書き込み		NP
XADD - 交換と加算		NP
XCHG - レジスタ/ メモリをレジスタと交換		NP
XLAT/XLATB - テーブル・ルックアップ・トランスレーション		NP
XOR - 排他的論理和		UV

intel

B

浮動小数点ペアリング表

付録 B
浮動小数点ペアリング表

この付録の浮動小数点命令ペア リング表には、以下の省略語が使用 されている。

FX - FXCH とのペア リング

NP - ペア リングなし

表 B-1. 浮動小数点命令ペアリング

命令	フォーマット	ペアリング
F2XM1 - $2^{ST(0)} - 1$ を計算		NP
FABS - 絶対値		FX
FADD - 加算		FX
FADDP - 加算とポップ		FX
FBLD - 2 進化 10 進数をロード		NP
FBSTP - 2 進化 10 進数のストアとポップ		NP
FCHS - 符号を変更		FX
FCLEX - 例外をクリア		NP
FCOM - 実数を比較		FX
FCOMP - 実数の比較とポップ		FX
FCOMPP - 実数比較と 2 回ポップ		
FCOS - ST(0) の余弦計算		NP
FDECSTP - スタックの トップ ・ポインタをデクリメント		NP
FDIV - 除算		FX
FDIVP - 除算とポップ		FX
FDIVR - 逆除算		FX
FDIVRP - 逆除算とポップ		FX
FFREE - ST(i) レジスタを解放		NP
FIADD - 整数を加算		NP
FICOM - 整数を比較		NP
FICOMP - 整数の比較とポップ		NP
FIDIV		NP
FIDIVR		NP
FILD - 整数をロード		NP

表 B-1. 浮動小数点命令ペアリング　続き)

命令	フォーマット	ペアリング
FMUL		NP
FNCSTP - スタック・ポインタをインクリメント		NP
FNINIT - 浮動小数点ユニットを初期化		NP
FIST - 整数をストア		NP
FISTP - 整数のストアとポップ		NP
FISUB		NP
FISUBR		NP
FLD - 実数をロード		
メモリから32 ビットをロード	11011 001 : mod 000 r/m	FX
メモリから64 ビットをロード	11011 101 : mod 000 r/m	FX
メモリから80 ビットをロード	11011 011 : mod 101 r/m	NP
ST(i)にロード	11011 001 : 11 000 ST(i)	FX
FLD1 - ST(0)に+1.0 をロード		NP
FLDCW - 制御ワードをロード		NP
FLDENV - FPU 環境をロード		NP
FLDL2E - ST(0)に $\log_2(e)$ をロード		NP
FLDL2T - ST(0)に $\log_2(10)$ をロード		NP
FLDLG2 - ST(0)に $\log_{10}(2)$ をロード		NP
FLDLN2 - ST(0)に $\log_e(2)$ をロード		NP
FLDPI - ST(0)にp をロード		NP
FLDZ - ST(0)に+0.0 をロード		NP
FMUL - 乗算		FX
FMULP - 乗算		FX
FNOP - オペレーションなし		NP
FPATAN - 部分逆正接計算		NP
FPREM - 部分剰余		NP
FPREM1 - 部分剰余 (IEEE)		NP
FPTAN - 部分正接計算		NP
FRNDINT - 整数に丸め		
FRSTOR - FPU 状態をリストア		NP

表 B-1. 浮動小数点命令ペアリング　続き)

命令	フォーマット	ペアリング
FSAVE - FPU 状態をストア		NP
FSCALE - スケール		NP
FSIN - 正弦計算		NP
FSINCOS - 正弦および余弦計算		NP
FSQRT - 平方根		NP
FST - 実数をストア		NP
FSTCW - 制御ワードをストア		NP
FSTENV - FPU 環境をストア		NP
FSTP - 実数のストアとポップ		NP
FSTSW - AX にステータス・ワードをストア		NP
FSTSW - メモリにステータス・ワードをストア		NP
FSUB - 減算		FX
FSUBP - 減算とポップ		FX
FSUBR - 逆減算		FX
FSUBRP - 逆減算とポップ		FX
FTST - テスト		FX
FUCOM - 実数順序化不可能比較		FX
FUCOMP - 実数順序化不可能比較とポップ		FX
FUCOMPP - 実数順序化不可能比較と 2 回ポップ		FX
FXAM - 検査		NP
FXCH - ST(0) と ST(i) を交換		
FXTRACT - 指数および仮数を抽出		NP
FYL2X - ST(1)xLog2(ST(0)) 計算		NP
FYL2XP1 - ST(1)xlog2(ST(0) + 1.0) 計算		NP
FWAIT - FPU レディまで待機		

intel

C

Pentium® Pro プロセッサ 命令デコード仕様

付録 C
Pentium® Pro プロセッサ命令デコード仕様

以下の表に、マクロ命令と各命令からデコードされるμop数の一覧を示す。

AAA	1
AAD	3
AAM	4
AAS	1
ADC AL,imm8	2
ADC eAX,imm16/32	2
ADC mem16/32,imm16/32	4
ADC mem16/32,reg16/32	4
ADC mem8,imm8	4
ADO mem8,reg8	4
ADC reg16/32,imm16/32	2
ADC reg16/32,mem16/32	3
ADC reg16/32,rm16/32	2
ADC reg8,imm8	2
ADC reg8,mem8	3
ADC reg8,rm8	2
ADC rm16/32,reg16/32	2
ADC rm8,reg8	2
ADD AL,imm8	1
ADD eAX,imm16/32	1
ADD mem16/32,imm16/32	4
ADD mem16/32,reg16/32	4
ADD mem8,imm8	4
ADD mem8,reg8	4

ADD reg16/32,imm16/32	1
ADD reg16/32,imm8	1
ADD reg16/32,mem16/32	2
ADD reg16/32,rm16/32	1
ADD reg8,imm8	1
ADD reg8,mem8	2
ADD reg8,rm8	1
ADD rm16/32,reg16/32	1
ADD rm8,reg8	1
AND AL,imm8	1
AND eAx,imm16/32	1
AND mem16/32,imm16/32	4
AND mem16/32,reg16/32	4
AND mem8,imm8	4
AND mem8,reg8	4
AND reg16/32,imm16/32	1
AND reg16/32,imm8	1
AND reg16/32,mem16/32	2
AND reg16/32,rm16/32	1
AND reg8,imm8	1
AND reg8,mem8	2
AND reg8,rm8	1
AND rm16/32,reg16/32	1
AND rm8,reg8	1

ARPL m16	コンプレックス
ARPL rm16,r16	コンプレックス
BOUND r16,m16/32&16/32	コンプレックス
BSF r16/32,m16/32	3
BSF r16/32,rm16/32	2
BSR r16/32,m16/32	3
BSR r16/32,rm16/32	2
BSWAP r32	2
BT m16/32,imm8	2
BT m16/32,r16/32	コンプレックス
BT rm16/32,imm8	1
BT rm16/32,r16/32	1
BTC m16/32,imm8	4
BTC m16/32,r16/32	コンプレックス
BTC rm16/32,imm8	1
BTC rm 16/32,r16/32	1
BTR m16/32,imm8	4
BTR m16/32,r16/32	コンプレックス
BTR m16/32,imm8	1
BTR m16/32,r16/32	1
BTS m16/32,imm8	4
BTS m16/32,r16/32	コンプレックス
BTS rm16/32,imm8	1
BTS rm16/32,r16/32	1
CALL m16/32 near	コンプレックス
CALL m16	コンプレックス
CALL ptr16	コンプレックス
CALL r16/32 near	コンプレックス
CALL rel16/32 near	4
CBW	1
CLC	1

CLD	4
CLI	コンプレックス
CLTS	コンプレックス
CMC	1
CMOVB/NAE/C r16/32,m16/32	3
CMOVB/NAE/C r16/32,r16/32	2
CMOVBE/NA r16/32,m16/32	3
CMOVBE/NA r16/32,r16/32	2
CMOVE/Z r16/32,m16/32	3
OMOVE/Z r16/32,r16/32	2
CMOVL/NGE r16/32,m16/32	3
CMOVL/NGE r16/32,r16/32	2
CMOVLE/NG r16/32,m16/32	3
CMOVLE/NG r16/32,r16/32	2
CMOVNB/AE/NC r16/32,m16/32	3
CMOVNB/AE/NC r16/32,r16/32	2
CMOVNBE/A r16/32,m16/32	3
CMOVNBE/A r16/32,r16/32	2
CMOVNE/NZ r16/32,m16/32	3
CMOVNE/NZ r16/32,r16/32	2
CMOVNL/GE r16/32,m16/32	3
CMOVNL/GE r16/32,r16/32	2
CMOVNLE/G r16/32,m16/32	3
CMOVNLE/G r16/32,r16/32	2
CMOVNO r16/32,m16/32	3
CMOVNO r16/32,r16/32	2
CMOVNP/PO r16/32,m16/32	3
CMOVNP/PO r16/32,r16/32	2
CMOVNS r16/32,m16/32	3
CMOVNS r16/32,r16/32	2
CMOVO r16/32,m16/32	3

CMOVO r16/32,r16/32	2
CMOVP/PE r16/32,m16/32	3
CMOVP/PE r16/32,r16/32	2
CMOVS r16/32,m16/32	3
CMOVS r16/32,r16/32	2
CMP AL,imm8	1
CMP eAX,imm16/32	1
CMP m16/32,imm16/32	2
CMP m16/32,imm8	2
CMP m16/32,r16/32	2
CMP m8,imm8	2
CMP m8,imm8	2
CMP m8,r8	2
CMP r16/32,m16/32	2
CMP r16/32,rm16/32	1
CMP r8,m8	2
CMP r8,rm8	1
CMP rm16/32,imm16/32	1
CMP rm16/32,imm8	1
CMP rm16/32,r16/32	1
CMP rm8,imm8	1
CMP rm8,imm8	1
CMP rm8,r8	1
CMPSB/W/D m8/16/32,m8/16/32	コンプレックス
CMPXCHG m16/32,r16/32	コンプレックス
CMPXCHG m8,r8	コンプレックス
CMPXCHG rm16/32,r16/32	コンプレックス
CMPXCHG rm8,r8	コンプレックス
CMPXCHG8B rm64	コンプレックス
CPUID	コンプレックス
CWD/CDQ	1

CWDE	1
DAA	1
DAS	1
DEC m16/32	4
DEC m8	4
DEC r16/32	1
DEC m16/32	1
DEC rm8	1
DIV AL,m8	3
DIV AX,m16/32	4
DIV AX,m8	4
DIV AX,rm16/32	4
ENTER	コンプレックス
F2XM1	コンプレックス
FABS	1
FADD ST(i),ST	1
FADD ST,ST(i)	1
FADD m32real	2
FADD m64real	2
FADDP ST(i),ST	1
FBLD m80dec	コンプレックス
FBSTP m80dec	コンプレックス
FCHS	3
FCMOVB STi	2
FCMOVBE STi	2
FCMOVE STi	2
FCMOVNB STi	2
FCMOVNBE STi	2
FCMOVNE STi	2
FCMOVNU STi	2
FCMOVU STi	2

FCOM STi	1
FCOM m32real	2
FCOM m64real	2
FCOM2 STi	1
FCOMI STi	1
FCOMIP STi	1
FCOMP STi	1
FCOMP m32real	2
FCOMP m64real	2
FCOMP3 STi	1
FCOMP5 STi	1
FCOMPP	2
FCOS	コンプレックス
FDECSTP	1
FDISI	1
FDIV ST(i),ST	1
FDIV ST,ST(i)	1
FDIV m32real	2
FDIV m64real	2
FDIVP ST(i),ST	1
FDIVR ST(i),ST	1
FDIVR ST,ST(i)	1
FDIVR m32real	2
FDIVR m64real	2
FDIVRP ST(i),ST	1
FENI	1
FFREE ST(i)	1
FFREEP ST(i)	2
FIADD m16int	コンプレックス
FIADD m32int	コンプレックス
FICOM m16int	コンプレックス

FICOM m32int	コンプレックス
FICOMP m16int	コンプレックス
FICOMP m32int	コンプレックス
FIDIV m16int	コンプレックス
FIDIV m32int	コンプレックス
FIDIVR m16int	コンプレックス
FIDIVR m32int	コンプレックス
FILD m16int	4
FILD m32int	4
FILD m64int	4
FIMUL m16int	コンプレックス
FIMUL m32int	コンプレックス
FINCSTP	1
FIST m16int	4
FIST m32int	4
FISTP m16int	4
FISTP m32int	4
FISTP m64int	4
FISUB m16int	コンプレックス
FISUB m32int	コンプレックス
FISUBR m16int	コンプレックス
FISUBR m32int	コンプレックス
FLD STi	1
FLD m32real	1
FLD m64real	1
FLD m80 real	4
FLDI	2
FLDCW m2byte	3
FLDENV m14/28byte	コンプレックス
FLDL2E	2
FLDL2T	2

FLDLG2	2
FLDLN2	2
FLDPI	2
FLDZ	1
FMUL ST(i),ST	1
FMUL ST,ST(i)	1
FMUL m32real	2
FMUL m64real	2
FMULP ST(i),ST	1
FNCLEX	3
FNINIT	コンプレックス
FNOP	1
FNSAVE m94/108byte	コンプレックス
FNSTCW m2byte	3
FNSTENV m14/28byte	コンプレックス
FNSTSW AX	3
FNSTSW m2byte	3
FPATAN	コンプレックス
FPREM	コンプレックス
FPREM1	コンプレックス
FPTAN	コンプレックス
FRNDINT	コンプレックス
FRSTOR m94/108byte	コンプレックス
FSCALE	コンプレックス
FSETPM	1
FSIN	コンプレックス
FSINCOS	コンプレックス
FSQRT	1
FST STi	1
FST m32real	2
FST m64real	2

FSTP STi	1
FSTP m32real	2
FSTP m64real	2
FSTP m80 real	コンプレックス
FSTPISTi	1
FSTPB STi	1
FSTP9 STi	1
FSUB ST(i),ST	1
FSUB ST,ST(i)	1
FSUB m32real	2
FSUB m64real	2
FSUBP ST(i),ST	1
FSUBR ST(i),ST	1
FSUBR ST,ST(i)	1
FSUBR m32real	2
FSUBR m64real	2
FSUBRP ST(i),ST	1
FTST	1
FUCOM STi	1
FUCOMISTi	1
FUCOMIP STi	1
FUCOMP STi	1
FUCOMPP	2
FWAIT	2
FXAM	1
FXCH STi	1
FXCH4 STi	1
FXCH7 STi	1
FXTRACT	コンプレックス
FYL2X	コンプレックス
FYL2XP1	コンプレックス

HALT	コンプレックス
DIV AL,rm8	3
DIV AX,rm16/32	4
DIV AX,rm8	4
DIV eAX,rm16/32	4
MUL rm16	4
MUL rm32	4
MUL rm8	2
MUL r16/32,rm16/32	2
MUL r16/32,rm16/32	1
MUL r16/32,rm16/32,imm8/16/32	2
MUL r16/32,rm16/32,imm8/16/32	1
MUL rm16	3
MUL rm32	3
MULrm8	1
IN eAX,DX	コンプレックス
IN eAX,imm8	コンプレックス
INCrm16/32	4
INCrma	4
INCr16/32	1
INCrm16/32	1
INCrm8	1
INSB/W/D m8/16/32,DX	コンプレックス
INT1	コンプレックス
INT3	コンプレックス
INTN	3
INTO	コンプレックス
INVD	コンプレックス
INVLPG m	コンプレックス
IRET	コンプレックス
JB/NAE/C rel16/32	1
JB/NAE/C rel8	1
JBE/NA rel16/32	1
JBE/NA rel8	1
JCXZ/JECXZ rel8	2
JE/Z rel16/32	1
JE/Z rel8	1
JL/NGE rel16/32	1
JL/NGE rel8	1
JLE/NG rel16/32	1
JLE/NG rel8	1
JMP m16	コンプレックス
JMP near m16/32	2
JMP near reg16/32	1
JMP ptr16	コンプレックス
JMP rel16/32	1
JMP rel8	1
JNB/AE/NC rel16/32	1
JNB/AE/NC rel8	1
JNBE/A rel16/32	1
JNBE/A rel8	1
JNE/NZ rel16/32	1
JNE/NZ rel8	1
JNL/GE rel16/32	1
JNL/GE rel8	1
JNLE/G rel16/32	1
JNLE/G rel8	1
JNO rel16/32	1
JNO rel8	1
JNP/PO rel16/32	1
JNP/PO rel8	1
JNS rel16/32	1

JNS rel8	1
JO rel16/32	1
JO rel8	1
JP/PE rel16/32	1
JP/PE rel8	1
JS rel16/32	1
JS rel8	1
LAHF	1
LAR m16	コンプレックス
LAR rm16	コンプレックス
LDS r16/32,m16	コンプレックス
LEA r16/32,m	1
LEAVE	3
LES r16/32,m16	コンプレックス
LFS r16/32,m16	コンプレックス
LGDT m16&32	コンプレックス
LGS r16/32,m16	コンプレックス
LIDT m16&32	コンプレックス
LLDT m16	コンプレックス
LLDT rm16	コンプレックス
LMSW m16	コンプレックス
LMSW r16	コンプレックス
LOCK ADC m16/32,imm16/32	コンプレックス
LOCK ADC m16/32,r16/32	コンプレックス
LOCK ADC m8,imm8	コンプレックス
LOCK ADC m8,r8	コンプレックス
LOCK ADD m16/32,imm16/32	コンプレックス
LOCK ADD m16/32,r16/32	コンプレックス
LOCK ADD m8,imm8	コンプレックス
LOCK ADD m8,r8	コンプレックス
LOCK AND m16/32,imm16/32	コンプレックス

LOCK AND m16/32,r16/32	コンプレックス
LOCK AND m8,imm8	コンプレックス
LOCK AND m8,r8	コンプレックス
LOCK BTC m16/32,imm8	コンプレックス
LOCK BTC m16/32,r16/32	コンプレックス
LOCK BTR m16/32,imm8	コンプレックス
LOCK BTR m16/32,r16/32	コンプレックス
LOCK BTS m16/32,imm8	コンプレックス
LOCK BTS m16/32,r16/32	コンプレックス
LOCK CMPXCHG m16/32,r16/32	コンプレックス
LOCK CMPXCHG m8,r8	コンプレックス
LOCK CMPXCHG8B rm64	コンプレックス
LOCK DECm 16/32	コンプレックス
LOCK DECm8	コンプレックス
LOCK INCm16/32	コンプレックス
LOCK INCm8	コンプレックス
LOCK NEGm16/32	コンプレックス
LOCK NEGm8	コンプレックス
LOCK NOTm16/32	コンプレックス
LOCK NOTm8	コンプレックス
LOCK ORm16/32,imm16/32	コンプレックス
LOCK ORm16/32,r16/32	コンプレックス
LOCK ORm8,imm8	コンプレックス
LOCK ORm8,r8	コンプレックス
LOCK SBB m16/32,imm16/32	コンプレックス
LOCK SBB m16/32,r16/32	コンプレックス
LOCK SBB m8,imm8	コンプレックス
LOCK SBB m8,r8	コンプレックス
LOCK SUB m16/32,imm16/32	コンプレックス
LOCK SUB m16/32,r16/32	コンプレックス
LOCK SUB m8,imm8	コンプレックス

LOCK SUB m8,r8	コンプレックス
LOCK XADD m16/32,r16/32	コンプレックス
LOCK XADD m8,r8	コンプレックス
LOCK XCHG m16/32,r16/32	コンプレックス
LOCK XCHG m8,r8	コンプレックス
LOCK XOR m16/32,imm16/32	コンプレックス
LOCK XOR m16/32,r16/32	コンプレックス
LOCK XOR m8,imm8	コンプレックス
LOCK XOR m8,r8	コンプレックス
LODSB/W/D m8/16/32,m8/16/32	2
LOOP rel8	4
LOOPE rel8	4
LOOPNE rel8	4
LSL m16	コンプレックス
LSL rm16	コンプレックス
LSS r16/32,m16	コンプレックス
LTR m16	コンプレックス
LTR rm16	コンプレックス
MOV AL,moffs8	1
MOV CR0,r32	コンプレックス
MOV CR2,r32	コンプレックス
MOV CR3,r32	コンプレックス
MOV OR4,r32	コンプレックス
MOV DRx,r32	コンプレックス
MOV DS,m16	4
MOV DS,rm16	4
MOV ES,m16	4
MOV ES,rm16	4
MOV FS,m16	4
MOV FS,rm16	4
MOV GS,m16	4

MOV GS,rm16	4
MOV SS,m16	4
MOV SS,rm16	4
MOV eAX,moffs16/32	1
MOV m16,CS	3
MOV m16,DS	3
MOV m16,ES	3
MOV m16,FS	3
MOV m16,GS	3
MOV m16,SS	3
MOV m16/32,imm16/32	2
MOV m16/32,r16/32	2
MOV m8,imm8	2
MOV m8,r8	2
MOV moffs16/32,eAX	2
MOV moffs8,AL	2
MOV r16/32,imm16/32	1
MOV r16/32,m16/32	1
MOV r16/32,rm16/32	1
MOV r32,CR0	コンプレックス
MOV r32,CR2	コンプレックス
MOV r32,CR3	コンプレックス
MOV r32,CR4	コンプレックス
MOV r32,DRx	コンプレックス
MOV r8,imm8	1
MOV r8,m8	1
MOV r8,rm8	1
MOV rm16,CS	1
MOV rm16,DS	1
MOV rm16,ES	1
MOV rm16,FS	1

MOV rm 16,GS	1
MOV rm16,SS	1
MOV rm16/32,imm16/32	1
MOV rm16/32,r16/32	1
MOV rm8,imm8	1
MOV rm8,r8	1
MOVSB/W/D m8/16/32,m8/16/32	コンプレックス
MOVSX r16,m8	1
MOVSX r16,rm8	1
MOVSX r16/32,m16	1
MOVSX r32,m8	1
MOVSX r32,rm16	1
MOVSX r32,rm8	1
MOVZX r16,m8	1
MOVZX r16,rm8	1
MOVZX r32,m16	1
MOVZX r32,m8	1
MOVZX r32,rm16	1
MOVZX r32,rm8	1
MUL AL,m8	2
MUL AL,rm8	1
MULAX,m16	4
MULAX,rm16	3
MUL EAX,m32	4
MUL EAX,rm32	3
NEGm16/32	4
NEGm8	4
NEGrm16/32	1
NEGrm8	1
NOP	1
NOTm16/32	4
NOTm8	4
NOTrm16/32	1
NOTrm8	1
ORAL,imm8	1
OReAX,imm16/32	1
ORm16/32,imm16/32	4
ORm16/32,r16/32	4
ORm8,imm8	4
ORm8,r8	4
ORr16/32,imm16/32	1
ORr16/32,imm8	1
ORr16/32,m16/32	2
ORr16/32,m16/32	1
ORr8,imm8	1
ORr8,m8	2
ORr8,rm8	1
ORrm16/32,r16/32	1
ORrm8,r8	1
OUT DX,eAX	コンプレックス
OUT imm8,eAX	コンプレックス
OUTSB/W/D DX,m8/16/32	コンプレックス
POP DS	コンプレックス
POP ES	コンプレックス
POP FS	コンプレックス
POP GS	コンプレックス
POP SS	コンプレックス
POP eSP	3
POP m16/32	コンプレックス
POP r16/32	2
POP r16/32	2
POPA/POPAD	コンプレックス

POPF	コンプレックス
POPFD	コンプレックス
PUSH CS	4
PUSH DS	4
PUSH ES	4
PUSH FS	4
PUSH GS	4
PUSH SS	4
PUSH imm16/32	3
PUSH imm8	3
PUSH m16/32	4
PUSH r16/32	3
PUSH r16/32	3
PUSHA/PUSHAD	コンプレックス
PUSHF/PUSHFD	コンプレックス
ROL m16/32,1	4
RCL m16/32,CL	コンプレックス
RCL m16/32,imm8	コンプレックス
RCL m8,1	4
RCL m8,CL	コンプレックス
RCL m8,imm8	コンプレックス
RCL rm16/32,1	2
RCL rm16/32,CL	コンプレックス
RCL rm16/32,imm8	コンプレックス
RCL rm8,1	2
RCL rm8,CL	コンプレックス
RCL rm8,imm8	コンプレックス
RCR m16/32,1	4
RCR m16/32,CL	コンプレックス
RCR m16/32,imm8	コンプレックス
RCR m8,1	4

RCR m8,CL	コンプレックス
ROR m8,imm8	コンプレックス
RCR rm16/32,1	2
RCR rm16/32,CL	コンプレックス
RCR rm16/32,imm8	コンプレックス
RCR rm8,1	2
RCR rm8,CL	コンプレックス
RCR rm8,imm8	コンプレックス
RDMSR	コンプレックス
RDPMC	コンプレックス
RDTSC	コンプレックス
REP CMPSB/W/D m8/16/32,m8/16/32	コンプレックス
REP INSB/W/D m8/16/32,DX	コンプレックス
REP LODSB/W/D m8/16/32,m8/16/32	コンプレックス
REP MOVSB/W/D m8/16/32,m8/16/32	コンプレックス
REP OUTSB/W/D DX,m8/16/32	コンプレックス
REP SCASB/W/D m8/16/32,m8/16/32	コンプレックス
REP STOSB/W/D m8/16/32,m8/16/32	コンプレックス
RET	4
RET	コンプレックス
RET near	4
RET near iw	コンプレックス
ROL m16/32,1	4
ROL m16/32,CL	4
ROL m16/32,imm8	4
ROL m8,1	4
ROL m8,CL	4
ROL m8,imm8	4
ROL rm16/32,1	1
ROL rm16/32,CL	1
ROL rm16/32,imm8	1

ROL rm8,1	1
ROL rm8,CL	1
ROL rm8,imm8	1
ROR m16/32,1	4
ROR m16/32,CL	4
ROR m16/32,imm8	4
ROR m8,1	4
ROR m8,CL	4
ROR m8,imm8	4
ROR rm16/32,1	1
ROR rm16/32,CL	1
ROR rm16/32,imm8	1
ROR rm8,1	1
ROR rm8,cl	1
ROR rm8,imm8	1
RSM	コンプレックス
SAHF	1
SAR m16/32,1	4
SAR m16/32,CL	4
SAR m16/32,imm8	4
SAR m8,1	4
SAR m8,CL	4
SAR m8,imm8	4
SAR rm16/32,1	1
SAR rm16/32,CL	1
SAR rm16/32,imm8	1
SAR rm8,1	1
SAR rm8,CL	1
SAR rm8,imm8	1
SBB AL,imm8	2
SBB eAX,imm16/32	2

SBB m 16/32,imm16/32	4
SBB m16/32,r16/32	4
SBB m8,imm8	4
SBB m8,r8	4
SBB r16/32,imm16/32	2
SBB r16/32,m16/32	3
SBB r16/32,rm16/32	2
SBB r8,imm8	2
SBB r8,m8	3
SBB r8,rm8	2
SBB rm16/32,r16/32	2
SBB rm8,r8	2
SCASB/W/D m8/16/32,m8/16/32	3
SETB/NAE/C m8	3
SETB/NAE/C rm8	1
SETBE/NA m8	3
SETBE/NA rm8	1
SETE/Z m8	3
SETE/Z rm8	1
SETL/NGE m8	3
SETL/NGE rm8	1
SETLE/NG m8	3
SETLE/NG rm8	1
SETNB/AE/NC m8	3
SETNB/AE/NC rm8	1
SETNBE/A m8	3
SETNBE/A rm8	1
SETNE/NZ m8	3
SETNE/NZ rm8	1
SETNL/GE m8	3
SETNL/GE rm8	1

SETNLE/G m8	3
SETNLE/G rm8	1
SETNO m8	3
SETNO rm8	1
SETNP/PO m8	3
SETNP/PO rm8	1
SETNS m8	3
SETNS rm8	1
SETO m8	3
SETO rm8	1
SETP/PE m8	3
SETP/PE rm8	1
SETS m8	3
SETS rm8	1
SGDT m16&32	4
SHL/SAL m16/32,1	4
SHL/SAL m16/32,1	4
SHL/SAL m16/32,CL	4
SHL/SAL m16/32,CL	4
SHL/SAL m16/32,imm8	4
SHL/SAL m16/32,imm8	4
SHL/SAL m8,1	4
SHL/SAL m8,1	4
SHL/SAL m8,CL	4
SHL/SAL m8,CL	4
SHL/SAL m8,imm8	4
SHL/SAL m8,imm8	4
SHL/SAL L/ rm16/32,1	1
SHL/SAL rm16/32,1	1
SHL/SAL rm16/32,CL	1
SHL/SAL rm16/32,CL	1

SHL/SAL rm16/32,imm8	1
SHL/SAL rm16/32,imm8	1
SHL/SAL rm8,1	1
SHL/SAL rm8,1	1
SHL/SAL rm8,CL	1
SHL/SAL rm8,CL	1
SHL/SAL rm8,imm8	1
SHL/SAL rm8,imm8	1
SHLD m16/32,r16/32,CL	4
SHLD m16/32,r16/32,imm8	4
SHLD rm16/32,r16/32,CL	2
SHLD rm16/32,r16/32,imm8	2
SHR m16/32,1	4
SHR m16/32,CL	4
SHR m16/32,imm8	4
SHR m8,1	4
SHR m8,CL	4
SHR m8,imm8	4
SHR rm16/32,1	1
SHR rm16/32,CL	1
SHR rm16/32,imm8	1
SHR rm8,1	1
SHR rm8,CL	1
SHR rm8,imm8	1
SHRD m16/32,r16/32,CL	4
SHRD m16/32,r16/32,imm8	4
SHRD rm16/32,r16/32,CL	2
SHRD rm16/32,r16/32,imm8	2
SIDT m16&32	コンプレックス
SLDT m16	コンプレックス
SLDT rm16	4

SMSW m16	コンプレックス
SMSW rm16	4
STC	1
STD	4
STI	コンプレックス
STOSB/W/D m8/16/32,m8/16/32	3
STR m16	コンプレックス
STR rm16	4
SUB AL,imm8	1
SUB eAX,imm16/32	1
SUB m16/32,imm16/32	4
SUB m16/32,r16/32	4
SUB m8,imm8	4
SUB m8,r8	4
SUB r16/32,imm16/32	1
SUB r16/32,imm8	1
SUB r16/32,m16/32	2
SUB r16/32,rm 16/32	1
SUB r8,imm8	1
SUB r8,m8	2
SUB r8,rm8	1
SUB rm 16/32,r16/32	1
SUB rm8,r8	1
TEST AL,imm8	1
TEST eAX,imm16/32	1
TEST m16/32,imm16/32	2
TEST m16/32,imm16/32	2
TEST m16/32,r16/32	2
TEST m8,imm8	2
TEST m8,imm8	2
TEST m8,r8	2

TEST rm16/32,imm 16/32	1
TEST rm16/32,r16/32	1
TEST rm8,imm8	1
TEST rm8,r8	1
VERR m16	コンプレックス
VERR rm16	コンプレックス
VERW m16	コンプレックス
VERW rm16	コンプレックス
WBINVD	コンプレックス
WRMSR	コンプレックス
XADD m16/32,r16/32	コンプレックス
XADD m8,r8	コンプレックス
XADD rm16/32,r16/32	4
XADD rm8,r8	4
XCHG eAX,r16/32	3
XCHG m16/32,r16/32	コンプレックス
XCHG m8,r8	コンプレックス
XCHG rm16/32,r16/32	3
XCHG rm8,r8	3
XLAT/B	2
XOR AL,imm8	1
XOR eAX,imm16/32	1
XOR m 16/32,imm16/32	4
XOR m16/32,r16/32	4
XOR m8,imm8	4
XOR m8,r8	4
XOR r16/32,imm16/32	1
XOR r16/32,imm8	1
XOR r16/32,m16/32	2
XOR r16/32,rm16/32	1
XOR r8,imm8	1

XOR r8,m8	2
XOR r8,rm8	1

XOR m16/32,r16/32	1
XOR rm8,r8	1

intel

D

Pentium® IIプロセッサ MMX® 命令デコード仕様

付録 D
Pentium® II プロセッサ MMX® 命令デコード仕様

EMMS	コンプレックス
MOVD m32,mm	2
MOVD mm,ireg	1
MOVD mm,m32	1
MOVQ mm,m64	1
MOVQ mm,mm	1
MOVQ m64,mm	2
MOVQ mm,mm	1
PACKSSDW mm,m64	2
PACKSSDW mm,mm	1
PACKSSWB mm,m64	2
PACKSSWB mm,mm	1
PACKUSWB mm,m64	2
PACKUSWB mm,mm	1
PADDB mm,m64	2
PADDB mm,mm	1
PADDD mm,m64	2
PADDD mm,mm	1
PADDSB mm,m64	2
PADDSB mm,mm	1
PADDSW mm,m64	2
PADDSW mm,mm	1
PADDUSB mm,m64	2
PADDUSB mm,mm	1
PADDUSW mm,m64	2
PADDUSW mm,mm	1
PADDW mm,m64	2

PADDW mm,mm	1
PAND mm,m64	2
PAND mm,mm	1
PANDN mm,m64	2
PANDN mm,mm	1
PCMPEQB mm,m64	2
PCMPEQB mm,mm	1
PCMPEQD mm,m64	2
PCMPEQD mm,mm	1
PCMPEQW mm,m64	2
PCMPEQW mm,mm	1
PCMPGTB mm,m64	2
PCMPGTB mm,mm	1
PCMPGTD mm,m64	2
PCMPGTD mm,mm	1
PCMPGTW mm,m64	2
PCMPGTW mm,mm	1
PMADDWD mm,m64	2
PMADDWD mm,mm	1
PMULHW mm,m64	2
PMULHW mm,mm	1
PMULLW mm,m64	2
PMULLW mm,mm	1
POR mm,m64	2
POR mm,mm	1
PSLLD mm,m64	2
PSLLD mm,mm	1

PSLLimmDmm,imm8	1
PSLLimmQmm,imm8	1
PSLLimmWmm,imm8	1
PSLLQmm,m64	2
PSLLQmm,mm	1
PSLLWmm,m64	2
PSLLWmm,mm	1
PSRADmm,m64	2
PSRADmm,mm	1
PSRAimmDmm,imm8	1
PSRAimmWmm,imm8	1
PSRAWmm,m64	2
PSRAWmm,mm	1
PSRLDmm,m64	2
PSRLDmm,mm	1
PSRLimmDmm,imm8	1
PSRLimmQmm,imm8	1
PSRLimmWmm,imm8	1
PSRLQmm,m64	2
PSRLQmm,mm	1
PSRLWmm,m64	2
PSRLWmm,mm	1
PSUBBmm,m64	2
PSUBBmm,mm	1
PSUBDmm,m64	2
PSUBDmm,mm	1

PSUBSBmm,m64	2
PSUBSBmm,mm	1
PSUBSWmm,m64	2
PSUBSWmm,mm	1
PSUBUSBmm,m64	2
PSUBUSBmm,mm	1
PSUBUSWmm,m64	2
PSUBUSWmm,mm	1
PSUBWmm,m64	2
PSUBWmm,mm	1
PUNPCKHBWmm,m64	2
PUNPCKHBWmm,mm	1
PUNPCKHDQmm,m64	2
PUNPCKHDQmm,mm	1
PUNPCKHWDmm,m64	2
PUNPCKHWDmm,mm	1
PUNPCKLBWmm,m32	2
PUNPCKLBWmm,mm	1
PUNPCKLDQmm,m32	2
PUNPCKLDQmm,mm	1
PUNPCKLWDmm,m32	2
PUNPCKLWDmm,mm	1
PXORmm,m64	2
PXORmm,mm	1