

# インテル® 64 および IA-32 アーキテクチャー 最適化リファレンス・マニュアル参考訳

参照ドキュメント番号: 248966-037 July 2017

## 注意事項:

この日本語マニュアルは、インテル コーポレーションのウェブサイト:

<https://software.intel.com/en-us/articles/intel-sdm> (英語)

で公開されている『Intel® 64 and IA-32 Architecture Optimization Reference Manual』の参考訳です。  
インテル社の許可を得て iSUS (IA Software User Society) が翻訳版を作成した iSUS の著作物です。

原文は Intel Corporation の Copyright であり、日本語参考訳版にも適用されます。

† 開発コード名

## 改版履歴

1. 2017 年 7 月 10 日: 036JA  
2017 年 6 月のバージョン 036 をベースとして最初の翻訳版を作成。  
2 章の一部 (2-1、2-2、2-3 節)、および 11 章、12 章、14 章、15 章の全文を追加。
2. 2017 年 8 月 1 日: 037JA\_Rev1  
2017 年 7 月のバージョン 037 をベースとして、2-1 節 (Skylake Server<sup>+</sup> マイクロアーキテクチャー)、2.2.4 節 (Skylake<sup>+</sup> マイクロアーキテクチャーのポーズ・レイテンシー)、8 章 (サブ NUMA クラスタリング)、13 章 (インテル<sup>®</sup>AVX-512) が追加されました。  
既存の 1 章 (はじめに) と 7 章 (キャッシュ利用の最適化) の訳を追加しました。
3. 2017 年 9 月 1 日: 037JA\_Rev2  
2017 年 7 月バージョン 037 をベースとして、付録 A (アプリケーション・パフォーマンス・ツール) と付録 B (パフォーマンス監視イベント) の訳を追加しました。
4. 2017 年 10 月 2 日: 037JA\_Rev3  
2.6 節 (インテル<sup>®</sup> マイクロアーキテクチャー開発コード名 Nehalem) から 2.10 節 (SIMD 技術とアプリケーション・レベル拡張のまとめ)、第 4 章 (SIMD アーキテクチャー向けのコーディング)、第 6 章 (SIMD 浮動小数点アプリケーション向けの最適化)、第 9 章 (マルチコアとハイパースレッディング・テクノロジー)、第 10 章 (64 ビット・モードのコーディング・ガイドライン)、付録 C (命令レイテンシーとスループット) の訳を追加しました。

オリジナルの英語版で既存の説明が追加および改定されている行の左に、赤い縦線でその場所を強調表示しています。

## このドキュメントに含まれる章

### 第 1 章 はじめに

- 1.1 アプリケーションをチューニングする
- 1.2 本書について
- 1.3 関連情報

### 第 2 章 インテル® 64 および IA-32 プロセッサ・アーキテクチャー

#### 2.1 SKYLAKE SERVER† マイクロアーキテクチャー

- 2.1.1 Skylake Server† マイクロアーキテクチャーのキャッシュ
- 2.1.2 Skylake Server† マイクロアーキテクチャー上での非テンポラルなストア

#### 2.2 Skylake† マイクロアーキテクチャー

- 2.2.1 フロントエンド
- 2.2.2 アウトオブオーダー実行エンジン
- 2.2.3 キャッシュとメモリー・サブシステム
- 2.2.4 Skylake† マイクロアーキテクチャーのポーズ・レイテンシー

#### 2.3 インテル® マイクロアーキテクチャー Haswell†

- 2.3.1 フロントエンド
- 2.3.2 アウトオブオーダー・エンジン
- 2.3.3 実行エンジン
- 2.3.4 キャッシュとメモリー・サブシステム
- 2.3.5 Haswell-E† マイクロアーキテクチャー
- 2.3.6 Broadwell† マイクロアーキテクチャー

#### 2.4 インテル® マイクロアーキテクチャー Sandy Bridge†

- 2.4.1 インテル® マイクロアーキテクチャー Sandy Bridge† のパイプライン概要
- 2.4.2 フロントエンド
- 2.4.3 アウトオブオーダー・エンジン
- 2.4.4 実行コア
- 2.4.5 キャッシュ階層
- 2.4.6 システム・エージェント
- 2.4.7 インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge

#### 2.5 インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャー

- 2.5.1 インテル® Core™ マイクロアーキテクチャーのパイプラインの概要
- 2.5.2 フロントエンド
- 2.5.3 実行コア
- 2.5.4 インテル® アドバンスド・メモリー・アクセス
- 2.5.5 インテル® アドバンスド・スマート・キャッシュ

#### 2.6 インテル® マイクロアーキテクチャー開発コード名 Nehalem

- 2.6.1 マイクロアーキテクチャー・パイプライン
- 2.6.2 フロントエンドの概要
- 2.6.3 実行エンジン
- 2.6.4 キャッシュとメモリー・サブシステム
- 2.6.5 ロード操作とストア操作の強化
- 2.6.6 REP (リピート) 文字列の強化
- 2.6.7 システム・ソフトウェアの強化
- 2.6.8 電力消費の効率化
- 2.6.9 インテル® マイクロアーキテクチャー開発コード名 Nehalem におけるハイパースレッディング・テクノロジーのサポート

## 2.7 インテル® ハイパースレッディング・テクノロジー

- 2.7.1 プロセッサ・リソースと HT テクノロジー
- 2.7.2 マイクロアーキテクチャー・パイプラインと HT テクノロジー
- 2.7.3 フロントエンドのパイプライン
- 2.7.4 実行コア
- 2.7.5 リタイア

## 2.8 インテル® 64 アーキテクチャー

### 2.9 SIMD 技術

#### 2.10 SIMD 技術とアプリケーション・レベル拡張のまとめ

- 2.10.1 インテル® MMX® テクノロジー
- 2.10.2 インテル® ストリーミング SIMD 拡張命令
- 2.10.3 インテル® ストリーミング SIMD 拡張命令 2
- 2.10.4 インテル® ストリーミング SIMD 拡張命令 3
- 2.10.5 インテル® ストリーミング SIMD 拡張命令 3 補足命令
- 2.10.6 インテル® ストリーミング SIMD 拡張命令 4.1
- 2.10.7 インテル® ストリーミング SIMD 拡張命令 4.2
- 2.10.8 インテル® AES New Instructions と PCLMULQDQ
- 2.10.9 インテル® アドバンスド・ベクトル・エクステンション
- 2.10.10 半精度浮動小数点変換 (F16C)
- 2.10.11 RDRAND
- 2.10.12 乗算-加算の融合 (FMA) 拡張
- 2.10.13 インテル® アドバンスド・ベクトル・エクステンション 2
- 2.10.14 汎用ビット処理命令
- 2.10.15 インテル® トランザクショナル・シンクロナイズーション・エクステンション
- 2.10.16 RDSEED
- 2.10.17 ADCX と ADOX 命令

## 第 4 章 SIMD アーキテクチャー向けのコーディング

### 4.1 プロセッサによる SIMD 技術のサポートをチェック

- 4.1.1 インテル® MMX® テクノロジーのサポートをチェック
- 4.1.2 インテル® ストリーミング SIMD 拡張命令のサポートをチェック
- 4.1.3 インテル® ストリーミング SIMD 拡張命令 2 のサポートをチェック
- 4.1.4 インテル® ストリーミング SIMD 拡張命令 3 のサポートをチェック
- 4.1.5 インテル® ストリーミング SIMD 拡張命令 3 補足命令のサポートをチェック
- 4.1.6 インテル® ストリーミング SIMD 拡張命令 4.1 のサポートをチェック
- 4.1.7 インテル® ストリーミング SIMD 拡張命令 4.2 のサポートをチェック
- 4.1.8 PCLMULQDQ およびインテル® AES-NI 命令のサポートを検出
- 4.1.9 インテル® AVX 命令の検出
- 4.1.10 VEX エンコードされた AES および VPCLMULQDQ の検出
- 4.1.11 F16C 命令の検出
- 4.1.12 FMA 命令の検出
- 4.1.13 インテル® AVX2 の検出

### 4.2 SIMD プログラミングにおけるコード変換に関する留意事項

- 4.2.1 ホットスポットの特定
- 4.2.2 SIMD 実行コードへの変換にメリットがあるかどうか判定

### 4.3 コーディング手法

- 4.3.1 各種コーディング手法

### 4.4 スタックとデータ・アライメント

- 4.4.1 アライメントとデータ・アクセス・パターンの隣接性
- 4.4.2 128 ビット SIMD 技術向けのスタック・アライメント
- 4.4.3 インテル® MMX® テクノロジー向けのデータ・アライメント

- 4.4.4 128 ビット・データ向けのデータ・アライメント
- 4.5 メモリー使用効率の改善
  - 4.5.1 データ構造体のレイアウト
  - 4.5.2 ストリップマイニング
  - 4.5.3 ループ・ブロッキング
- 4.6 命令の選択
  - 4.6.1 SIMD の最適化とマイクロアーキテクチャー
- 4.7 開発の最終段階におけるアプリケーションのチューニング

## 第 6 章 SIMD 浮動小数点アプリケーション向けの最適化

- 6.1 SIMD 浮動小数点コード向けの一般的な規則
- 6.2 計画上の留意事項
- 6.3 x87 浮動小数点と SIMD 浮動小数点との併用
- 6.4 スカラー浮動小数点コード
- 6.5 データ・アライメント
  - 6.5.1 データ配置
  - 6.5.2 CVTTSS2PI/CVTTSS2SI 命令の使用
  - 6.5.3 ゼロ・フラッシュ・モードと DAZ モード
- 6.6 SIMD の最適化とマイクロアーキテクチャー
  - 6.6.1 インテル® SSE3 を使用した SIMD 浮動小数点プログラミング
  - 6.6.2 ドット積と水平 SIMD 命令
  - 6.6.3 ベクトルの正規化
  - 6.6.4 水平 SIMD 命令セットの使用とデータレイアウト

## 第 7 章 キャッシュ利用の最適化

- 7.1 プリフェッチのコーディングに関する一般的なガイドライン
- 7.2 プリフェッチとキャッシュ制御命令
- 7.3 プリフェッチ
  - 7.3.1 ソフトウェアによるデータ・プリフェッチ
  - 7.3.2 プリフェッチ命令
  - 7.3.3 プリフェッチとロード命令
- 7.4 キャッシュ制御
  - 7.4.1 非テンポラルなストア命令
  - 7.4.2 ストリーミング・ストアの利用モデル
  - 7.4.3 ストリーミング・ストア命令の説明
  - 7.4.4 ストリーミング・ロード命令
  - 7.4.5 FENCE 命令
  - 7.4.6 CLFLUSH 命令
  - 7.4.7 CLFLUSHOPT 命令
- 7.5 プリフェッチを使用したメモリーの最適化
  - 7.5.1 ソフトウェア制御プリフェッチ
  - 7.5.2 ハードウェア・プリフェッチ
  - 7.5.3 ハードウェア・プリフェッチで実効レイテンシーを削減する例
  - 7.5.4 ソフトウェア・プリフェッチ命令でレイテンシーを隠蔽する例
  - 7.5.5 ソフトウェア・プリフェッチを使用する際の確認事項
  - 7.5.6 ソフトウェア・プリフェッチのスケジューリング間隔
  - 7.5.7 ソフトウェア・プリフェッチの連結
  - 7.5.8 ソフトウェア・プリフェッチの数を最小化する
  - 7.5.9 ソフトウェア・プリフェッチ命令と演算命令を混在させる
  - 7.5.10 ソフトウェア・プリフェッチとキャッシュ・ブロッキング

- 7.5.11 ソフトウェア・プリフェッチとキャッシュ・ブロッキング
- 7.5.12 シングルバス実行とマルチバス実行の比較

## 7.6 非テンポラルなストアを使用したメモリーの最適化

- 7.6.1 非テンポラルなストアとソフトウェアによるライトコンバイン
- 7.6.2 キャッシュ管理
- 7.6.3 キャッシュ・パラメーター

## 第 8 章 サブ NUMA クラスタリングの概要

- 8.1 サブ NUMA クラスタリング
- 8.2 クラスタオンダイとの比較
- 8.3 SNC の利用
  - 8.3.1 NUMA 構成をチェックする方法
  - 8.3.2 SNC 向けに MPI を最適化
  - 8.3.3 SNC のパフォーマンス比較

## 第 9 章 マルチコアとハイパースレッディング・テクノロジー

- 9.1 性能および使用モデル
  - 9.1.1 マルチスレッディング
  - 9.1.2 マルチタスキング環境
- 9.2 プログラミング・モデルとマルチスレッディング
  - 9.2.1 並列プログラミング・モデル
  - 9.2.2 機能分解
  - 9.2.3 専用プログラミング・モデル
  - 9.2.4 マルチスレッド・アプリケーション作成用のツール
- 9.3 最適化のガイドライン
  - 9.3.1 スレッド間の同期の主な慣例
  - 9.3.2 システムバス最適化の主な慣例
  - 9.3.3 メモリー最適化の主な慣例
  - 9.3.4 実行リソース最適化の主な慣例
  - 9.3.5 一般性およびパフォーマンスの影響
- 9.4 スレッド間の同期
  - 9.4.1 同期プリミティブの選択
  - 9.4.2 短期間の同期
  - 9.4.3 スピンロックによる最適化
  - 9.4.4 長期間の同期
  - 9.4.5 変更されたデータの共有とフォルス・シェアリングの防止
  - 9.4.6 共有同期変数の配置
- 9.5 システムバスの最適化
  - 9.5.1 バス帯域幅の保持
  - 9.5.2 バスとキャッシュとの相互作用について
  - 9.5.3 過度なソフトウェア・プリフェッチを避ける
  - 9.5.4 キャッシュミスの実効レイテンシーを改善
  - 9.5.5 フルサイズ書き込みトランザクションによる高データレートの実現
- 9.6 メモリー最適化
  - 9.6.1 キャッシュ・ブロッキングのテクニック
  - 9.6.2 メモリー最適化
  - 9.6.3 64KB エイリアスのデータアクセスを排除
- 9.7 フロントエンドの最適化
  - 9.7.1 過度なループアンロールの回避

## 9.8 アフィニティーと共有プラットフォーム・リソースの管理

- 9.8.1 トポロジー共有リソースの列挙
- 9.8.2 NUMA (Non-Uniform Memory Access)

## 9.9 その他の共有リソースの最適化

- 9.9.1 HT テクノロジー最適化の可能性を拡大

## 第 10 章 64 ビット・モードのコーディング・ガイドライン

### 10.1 はじめに

#### 10.2 64 ビット・モードに影響するコーディング規則

- 10.2.1 データサイズが 32 ビットの場合はレガシーの 32 ビット命令を使用
- 10.2.2 追加のレジスターを使用してレジスターへの負荷を削減
- 10.2.3 64 ビット値同士の乗算を有効活用
- 10.2.4 128 ビット整数除算を 128 ビット乗算で置き換え
- 10.2.5 完全な 64 ビットへの符号拡張

#### 10.3 64 ビット・モード向けの代替えコーディング規則

- 10.3.1 64 ビット演算結果では 2 つの 32 ビット・レジスターの代わりに 64 ビット・レジスターを使用
- 10.3.2 CVTSS2SS と CVTSS2SD
- 10.3.3 ソフトウェア・プリフェッチの使用

## 第 12 章 インテル® AVX、FMA およびインテル® AVX2 向けの最適化

### 12.1 インテル® AVX 組込み関数のコーディング

- 12.1.1 インテル® AVX アセンブリーのコーディング

### 12.2 非破壊ソース (NDS)

### 12.3 インテル® AVX コードとインテル® SSE コードの混在

- 12.3.1 関数呼び出しでインテル® AVX とインテル® SSE を混在させる

### 12.4 128 ビット・レーン操作とインテル® AVX

- 12.4.1 レーンの概念とプログラミング
- 12.4.2 ストライドロードの手法
- 12.4.3 レジスター・オーバーラップの手法

### 12.5 データのギャザーとスカッター

- 12.5.1 データギャザー
- 12.5.2 データ・スカッター

### 12.6 インテル® AVX 向けのデータ・アライメント

- 12.6.1 32 バイトにデータをアライメント
- 12.6.2 メモリーがアライメントされていない場合に 16 バイトのメモリーアクセスを考慮する
- 12.6.3 ロードのアライメントよりもストアのアライメントが重要

### 12.7 L1D キャッシュラインの置き換え

### 12.8 4K エイリアシング

### 12.9 条件付き SIMD パックドロードとストア

- 12.9.1 条件付きループ

### 12.10 整数と浮動小数点コードの混在

### 12.11 ポート 5 への負荷の考慮

- 12.11.1 シャッフルをブレードに置き換える
- 12.11.2 シャッフルの数を減らしたアルゴリズムの設計
- 12.11.3 ロードポートで基本的なシャッフルを実行

### 12.12 除算と平方根命令

- 12.12.1 単精度除算

- 12.12.2 単精度逆数平方根
- 12.12.3 単精度平方根
- 12.13 ARRAY SUB SUM の最適化例
- 12.14 半精度浮動小数点変換
  - 12.14.1 パックド単精度から半精度への変換
  - 12.14.2 パックド半精度から単精度への変換
  - 12.14.3 帯域幅を維持するため半精度 FP の局所性を考慮
- 12.15 乗算-加算融合 (FMA) 命令のガイドライン
  - 12.15.1 FMA と浮動小数点 Add/Mul におけるスループットの最適化
  - 12.15.2 ベクトルシフトによるスループットの最適化
- 12.16 インテル® AVX2 最適化のガイドライン
  - 12.16.1 マルチバッファリングとインテル® AVX2
  - 12.16.2 剰余除算とインテル® AVX2
  - 12.16.3 データ移動に関する考察
  - 12.16.4 Gather 命令に関する考察
  - 12.16.5 インテル® MMX® 命令のスループットの制限によるインテル® AVX2 への変換方法

## 第 13 章 インテル® AVX-512 向けの最適化

- 13.1 インテル® AVX-512 とインテル® AVX2 コーディングの基礎
  - 13.1.1 組み込み関数によるコーディング
  - 13.1.2 アセンブリによるコーディング
- 13.2 マスク処理
  - 13.2.1 マスクの例
  - 13.2.2 マスクのコスト
  - 13.2.3 マスクとブレンド
  - 13.2.4 入れ子になった条件 / マスク集合
  - 13.2.5 メモリーマスクのマイクロアーキテクチャーを改善
  - 13.2.6 ピーリングとリマインダーのマスク
- 13.3 フォワーディングとマスク付き操作
- 13.4 フォワーディングとメモリーマスク
- 13.5 データコンプレス
  - 13.5.1 データコンプレスの例
- 13.6 データ・エキスパンド
  - 13.6.1 データ・エキスパンドの例
- 13.7 三値論理
  - 13.7.1 三値論理の例 1
  - 13.7.2 三値論理の例 2
- 13.8 新しいシャッフル命令
  - 13.8.1 2 つのソースのパーミュートの例
- 13.9 ブロードキャスト処理
  - 13.9.1 組み込みブロードキャスト
  - 13.9.2 ロードポートで実行されるブロードキャスト
- 13.10 組み込み丸め操作
  - 13.10.1 静的丸めモード
- 13.11 スキャッター命令
  - 13.11.1 データ・スキャッターの例
- 13.12 静的丸めモード、すべての例外を抑制 (SAE)



### 13.13 QWORD 命令のサポート

- 13.13.1 算術命令での QUADWORD サポート
- 13.13.2 変換命令での QUADWORD サポート
- 13.13.3 切り捨て変換命令での QUADWORD サポート

### 13.14 ベクトル長の直交性

### 13.15 超越計算サポート向けのインテル® AVX-512 命令

- 13.15.1 VRCP14, VRSQRT14 -  $1/x$ ,  $x/y$ ,  $\sqrt{x}$  向けのソフトウェア・シーケンス
- 13.15.2 VGETMANT, VGETEXP - ベクトル仮数とベクトル指数の取得
- 13.15.3 VRNDSCALE - ベクトル丸めスケール
- 13.15.4 VREDUCE - ベクトルレデュース
- 13.15.5 VSCALEF - ベクトルスケール
- 13.15.6 VFPCLASS - ベクトル浮動小数点クラス
- 13.15.7 VPERM, VPERMI2, VPERMT2 - 小規模テーブル索引の実装

### 13.16 競合検出

- 13.16.1 競合検出とベクトル化
- 13.16.2 VPCONFLICT による疎ドット積

### 13.17 FMA のレイテンシー

### 13.18 インテル® AVX 拡張またはインテル® AVX-512 拡張命令とインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の混在

### 13.19 ZMM ベクトルコードと XMM/YMM コードの混在

### 13.20 単一の FMA ユニットの備える場合

### 13.21 シャッフルのためのギャザー/スカッター (G2S/STS)

- 13.21.1 スライドロードでシャッフルするためのギャザー
- 13.21.2 スライドストアでシャッフルするためのスカッター
- 13.21.3 隣接するロードでシャッフルするためのギャザー

### 13.22 データ・アライメント

- 13.22.1 64 バイトにデータをアライメント

### 13.23 動的メモリー割り当てとメモリーのアライメント

### 13.24 除算と平方根命令

- 13.24.1 除算と平方根命令の近似
- 13.24.2 除算と平方根命令のパフォーマンス
- 13.24.3 近似のレイテンシー
- 13.24.4 コード例

### 13.25 コンパイラーを利用するヒント

## 第 14 章 インテル® TSX の推奨事項

### 14.1 はじめに

- 14.1.1 最適化の概略

### 14.2 アプリケーション・レベルのチューニングと最適化

- 14.2.1 既存の TSX 対応ロック・ライブラリー
- 14.2.2 初期のチェック
- 14.2.3 アプリケーションの実行とプロファイル
- 14.2.4 トランザクション・アポートを最小限に抑える
- 14.2.5 トランザクション実行専用のコードパスの使用
- 14.2.6 アポートが頻発するトランザクション領域またはトランザクション・パスへの対応

### 14.3 インテル® TSX 対応の同期ライブラリーの開発

- 14.3.1 HLE プリフィックスの追加
- 14.3.2 省略に適したクリティカル・セクションのロック
- 14.3.3 ロックの省略における HLE または RTM の使用

- 14.3.4 ロックの省略に RTM を使用するラッパーの例
- 14.3.5 RTM フォールバック・ハンドラーのガイドライン
- 14.3.6 インテル® TSX による省略に適したロックの実装
- 14.3.7 インテル® TSX を使用するアプリケーション固有のメタロックの省略
- 14.3.8 永続的な非省略実行を回避する
- 14.3.9 RTM ベースのライブラリーで省略されたロックの値を読み取る
- 14.3.10 HLE と RTM を混在させる
- 14.4 インテル® TSX のパフォーマンス監視サポートを利用する
  - 14.4.1 トランザクション成功を測定する
  - 14.4.2 省略するロックを特定してすべてのロックが省略されることを確認する
  - 14.4.3 トランザクション・アボートのサンプリング
  - 14.4.4 プロファイリング・ツールを利用してアボートを分類する
  - 14.4.5 RTM フォールバック・ハンドラー向けの XABORT 引数
  - 14.4.6 トランザクション・アボートのコールグラフ
  - 14.4.7 LBR とトランザクション・アボート
  - 14.4.8 インテル® SDE によるインテル® TSX ソフトウェアのプロファイリングとテスト
  - 14.4.9 HLE 固有のパフォーマンス監視イベント
  - 14.4.10 インテル® TSX の有用なメトリックを計算する
- 14.5 パフォーマンスのガイドライン
- 14.6 デバッグのガイドライン
- 14.7 インテル® TSX 用の一般的な組込み関数
  - 14.7.1 RTM C 組込み関数
  - 14.7.2 gcc\* およびその他の Linux\* 互換コンパイラーの HLE 組込み関数
  - 14.7.3 Windows\* C/C++ コンパイラーの HLE 組込み関数

## 第 16 章 Goldmont<sup>†</sup> および Silvermont<sup>†</sup> マイクロアーキテクチャー向けの ソフトウェア最適化

- 16.1 最近の Intel Atom® プロセッサ世代のマイクロアーキテクチャー
  - 16.1.1 Goldmont<sup>†</sup> マイクロアーキテクチャー
  - 16.1.2 Silvermont<sup>†</sup> マイクロアーキテクチャー
- 16.2 Silvermont<sup>†</sup> マイクロアーキテクチャーにおけるコーディングの推奨事項
  - 16.2.1 フロントエンドの最適化
  - 16.2.2 実行コアの最適化
  - 16.2.3 メモリアクセスの最適化
- 16.3 命令レイテンシーとスループット

## 第 17 章 Knights Landing<sup>†</sup> マイクロアーキテクチャーとソフトウェアの最適化

- 17.1 Knights Landing<sup>†</sup> マイクロアーキテクチャー
  - 17.1.1 フロントエンド
  - 17.1.2 アウトオブオーダー・エンジン
  - 17.1.3 アンタイル
- 17.2 Knights Landing<sup>†</sup> マイクロアーキテクチャー向けのインテル® AVX-512 コーディングの推奨事項
  - 17.2.1 Gather および Scatter 命令の利用
  - 17.2.2 拡張された逆数命令の利用
  - 17.2.3 インテル® AVX-512CD 命令の利用
  - 17.2.4 インテル® ハイパースレッディング・テクノロジーの利用
  - 17.2.5 フロントエンドに関する考察
  - 17.2.6 整数実行に関する考察
  - 17.2.7 FP およびベクトル実行の最適化
  - 17.2.8 メモリー最適化

## 付録 A アプリケーション・パフォーマンス・ツール

### A.1 コンパイラー

- A.1.1 インテル® 64 プロセッサと IA-32 プロセッサの推奨される最適化設定
- A.1.2 ベクトル化とループの最適化
- A.1.3 ライブラリー関数のインライン展開 (/Oi, /Oi-)
- A.1.4 プロシージャー間とプロファイルに基づく最適化
- A.1.5 インテル® Cilk™ Plus

### A.2 パフォーマンス・ライブラリー

- A.2.1 インテル® インテグレートッド・パフォーマンス・プリミティブ (インテル® IPP)
- A.2.2 インテル® マス・カーネル・ライブラリー (インテル® MKL)
- A.2.3 インテル® スレッディング・ビルディング・ブロック (インテル® TBB)
- A.2.4 利点のまとめ

### A.3 パフォーマンス・プロファイラー

- A.3.1 インテル® VTune™ Amplifier

### A.4 スレッドとメモリーチェッカー

- A.4.1 インテル® Inspector

### A.5 ベクイトル化のアシスタント

- A.5.1 インテル® Advisor

### A.6 クラスターツール

- A.6.1 インテル® Trace Analyzer & Collector
- A.6.2 インテル® MPI ライブラリー
- A.6.3 インテル® MPI Benchmarks

### A.7 インテル® ACADEMIC COMMUNITY

## 付録 B パフォーマンス監視イベント

### B.1 トップダウン解析法

- B.1.1 トップレベル
- B.1.2 フロントエンド依存
- B.1.3 フロントエンド依存
- B.1.4 メモリー依存
- B.1.5 コア依存
- B.1.6 投機の問題
- B.1.7 リタイア
- B.1.8 TMAM と Skylake<sup>+</sup> マイクロアーキテクチャー

### B.2 パフォーマンス監視とマイクロアーキテクチャー

#### B.3 インテル® Xeon® プロセッサ 5500 番台

#### B.4 インテル® Xeon® プロセッサ 5500 番台のパフォーマンス分析手法

- B.4.1 サイクル・アカウンティングとマイクロオペレーション (μop) フロー
- B.4.2 ストールサイクルの分解とコア・メモリー・アクセス
- B.4.3 コア PMU のプリサイズイベント
- B.4.4 フロントエンドの監視イベント
- B.4.5 アンコア・パフォーマンス監視イベント
- B.4.6 インテル® QuickPath インターコネクットのホームロジック (QHL)
- B.4.7 アンコアからの帯域幅測定

#### B.5 インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge のパフォーマンス・チューニング手法

- B.5.1 パフォーマンスのボトルネックとソース行の関連付け
- B.5.2 階層的なトップダウン・パフォーマンス特性方式とパフォーマンス・ボトルネックの特定
- B.5.3 バックエンドのストール
- B.5.4 メモリー・サブシステム ストール

- B.5.5 実行ストール
- B.5.6 投機の問題
- B.5.7 フロントエンドのストール
- B.6 インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサのパフォーマンス・イベントの使用
  - B.6.1 パフォーマンス・カウンターの結果を理解する
  - B.6.2 比率の解釈
  - B.6.3 特定のイベントに関する注意事項
- B.7 パフォーマンス分析のドリルダウン手法
  - B.7.1 発行ポートでのサイクル構成
  - B.7.2 アウトオブオーダー (OOO) 実行のサイクル構成
  - B.7.3 パフォーマンス・ストールのドリルダウン
- B.8 インテル® Core™ マイクロアーキテクチャーのイベント比率
  - B.8.1 命令リタイアごとのクロック比率 (CPI)
  - B.8.2 フロントエンド比率
  - B.8.3 分岐予測比率
  - B.8.4 実行比率
  - B.8.5 メモリー・サブシステム - アクセス競合の比率
  - B.8.6 メモリー・サブシステム - キャッシュミスの比率
  - B.8.7 メモリー・サブシステム - プリフェッチ
  - B.8.8 メモリー・サブシステム - TLB ミス比率
  - B.8.9 メモリー・サブシステム - コアとの相互作用
  - B.8.10 メモリー・サブシステム - バスの特性

## 付録 C 命令レイテンシーとスループット

- C.1 概要
- C.2 用語説明
- C.3 レイテンシーとスループット
  - C.3.1 レジスターオペランドのレイテンシーとスループット
  - C.3.2 表の脚注について
  - C.3.3 メモリーオペランドを持つ命令

本書では、IA-32 アーキテクチャー・ベースのプロセッサとインテル® 64 アーキテクチャー・ベースのプロセッサのパフォーマンス特性を利用して、ソフトウェアを最適化する方法について説明します。本書で説明する最適化手法は、次のプロセッサ・ファミリーに適用されます。

- Y インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサ
- Y 拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサ
- Y インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサ
- Y インテル® マイクロアーキテクチャー開発コード名 Westmere ベースのプロセッサ
- Y インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge ベースのプロセッサ
- Y インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge ベースのプロセッサ
- Y インテル® マイクロアーキテクチャー開発コード名 Haswell ベースのプロセッサ
- Y インテル® マイクロアーキテクチャー開発コード名 Skylake ベースのプロセッサ
- Y Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサ
- Y インテル® Core™ Duo プロセッサ・ファミリー
- Y インテル® Core™ Solo プロセッサ・ファミリー
- Y インテル® Pentium® M プロセッサ・ファミリー

本書の対象読者は、ソフトウェア・プログラマーとコンパイラ開発者です。本書の読者は、IA-32 アーキテクチャーの基礎知識をよく理解し、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』(全 5 巻) を参照する必要があります。多くを理解する上で、インテル® 64 プロセッサと IA-32 プロセッサについて熟知していることが求められます。また、基礎となるマイクロアーキテクチャーに関する知識が必要です。

本書で説明する高性能ソフトウェア開発向けの設計ガイドラインは、現在および将来の IA-32 プロセッサとインテル® 64 プロセッサのほとんどに適用できます。説明するコーディング規則とコード最適化手法は、インテル® Core® マイクロアーキテクチャー、Intel NetBurst® マイクロアーキテクチャー、インテル® Pentium® M プロセッサ・マイクロアーキテクチャーを対象としています。大部分のコーディング規則は、インテル® 64 アーキテクチャーの 64 ビット・モード、インテル® 64 アーキテクチャーの互換モード、IA-32 モード (IA-32 モードは IA-32 アーキテクチャーとインテル® 64 アーキテクチャーでサポートされる) で実行されるソフトウェアに適用されます。64 ビット・モードに固有のコーディング規則については、個別に記載しています。

## 1.1 アプリケーションをチューニングする

インテル® 64 プロセッサと IA-32 プロセッサ上で最大限の性能を引き出すためアプリケーションをチューニングするには、以下の基礎知識が必要となります。

- Y インテル® 64 および IA-32 プロセッサ・アーキテクチャー
- Y C 言語とアセンブリ言語
- Y パフォーマンスに影響を与えるアプリケーションのホットスポット領域
- Y コンパイラの最適化機能
- Y アプリケーションのパフォーマンスの評価手法

インテル® VTune™ Amplifier (旧製品名: インテル® VTune™ パフォーマンス・アナライザー) は、アプリケーションのホットスポット領域の分析と特定を支援するツールです。このツールは、インテル® Core™ i7 プロセッサ、インテル® Core™2 Duo プロセッサ、インテル® Core™ Duo プロセッサ、インテル® Core™ Solo プロセッサ、インテル® Pentium® 4 プロセッサ、インテル® Xeon® プロセッサ、インテル® Pentium® M プロセッサ上で、一連のパフォーマンス監視イベントを使用してアプリケーションを監視し、コードの実行時に収集されたパフォーマンス

ンス・イベント・データを分析できます。随時最新のプロセッサがサポートされますが詳細については、インテル® VTune™ Amplifier のリリースノートを参照してください。

本書では、インテル® Pentium® 4 プロセッサのパフォーマンス監視イベントによって、パフォーマンス・カウンターを使って収集される情報についても説明します。

## 1.2 本書について

インテル® Xeon® プロセッサ 3000/3200/5100/5300/7200/7300 番台、インテル® Pentium® デスクトップ・プロセッサ、インテル® Core™2 Duo プロセッサ、インテル® Core™2 Quad プロセッサ、インテル® Core™2 Extreme プロセッサは、インテル® Core™ マイクロアーキテクチャーをベースにしています。本書で述べる「インテル® Core™2 Duo プロセッサ」は、インテル® Core™ マイクロアーキテクチャー・ベースのすべてのプロセッサを示します。

インテル® Xeon® プロセッサ 3100/3300/5200/5400/7400 番台、インテル® Core™2 Quad プロセッサ Q8000 番台、インテル® Core™2 Extreme プロセッサ QX9000 番台は、45nm 拡張版インテル® Core™ マイクロアーキテクチャーをベースにしています。

インテル® Core™ i7 プロセッサ、インテル® Xeon® プロセッサ 3400/5500/7500 番台は、45nm インテル® マイクロアーキテクチャー開発コード名 Nehalem をベースにしています。インテル® マイクロアーキテクチャー開発コード名 Westmere は、インテル® マイクロアーキテクチャー開発コード名 Nehalem の 32nm バージョンです。インテル® Xeon® プロセッサ 5600 番台、インテル® Xeon® プロセッサ E7 ファミリー、および各種のインテル® Core™ i7/i5/i3 プロセッサは、インテル® マイクロアーキテクチャー開発コード名 Westmere をベースにしています。

インテル® Xeon® プロセッサ E5 ファミリー、インテル® Xeon® プロセッサ E3-1200 製品ファミリー、インテル® Xeon® プロセッサ E7-8800/4800/2800 製品ファミリー、および第 2 世代インテル® Core™ i7-2000 プロセッサ・シリーズ、インテル® Core™ i5-2000 プロセッサ・シリーズ、インテル® Core™ i3-2000 プロセッサ・シリーズは、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge をベースにしています。

第 3 世代インテル® Core™ プロセッサとインテル® Xeon® プロセッサ E3-1200 v2 製品ファミリーは、インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge をベースにしています。インテル® Xeon® プロセッサ E5 v2 とインテル® Xeon® プロセッサ E7 v2 ファミリーは、Ivy Bridge-E<sup>†</sup> マイクロアーキテクチャーをベースにし、インテル® 64 とプラットフォームで複数の物理プロセッサ・パッケージをサポートします。

第 4 世代インテル® Core™ プロセッサとインテル® Xeon® プロセッサ E3-1200 v3 製品ファミリーは、インテル® マイクロアーキテクチャー開発コード名 Haswell をベースにしています。インテル® Xeon® プロセッサ E5-26xx v3 製品ファミリーは、Haswell-E<sup>†</sup> マイクロアーキテクチャーをベースにし、インテル® 64 とプラットフォームで複数の物理プロセッサ・パッケージをサポートします。

インテル® Core™ M プロセッサ・ファミリーと第 5 世代インテル® Core™ プロセッサは、インテル® マイクロアーキテクチャー開発コード名 Broadwell をベースとし、インテル® 64 アーキテクチャーをサポートします。

第 6 世代インテル® Core™ プロセッサは、インテル® マイクロアーキテクチャー開発コード名 Skylake をベースとし、インテル® 64 アーキテクチャーをサポートします。

本書で述べる「インテル® Pentium® 4 プロセッサ」は、Intel NetBurst® マイクロアーキテクチャー・ベースのすべてのプロセッサを示します。これには、インテル® Pentium® 4 プロセッサと、Intel NetBurst® アーキテクチャーをベースとする多くのインテル® Xeon® プロセッサが含まれます。必要な場合には、それぞれの違いを特記します (例えば一部のインテル® Xeon® プロセッサは 3 次キャッシュを搭載しています)。

デュアルコア インテル® Xeon® プロセッサ LV は、インテル® Core™ Duo プロセッサおよびインテル® Core™ Solo プロセッサと同じアーキテクチャーをベースにしています。

Intel Atom® プロセッサは、Intel Atom® マイクロアーキテクチャーをベースにしています。

以下に、本書に含まれる各章の概要を示します。

- Y **第 1 章: 「はじめに」:** 本書の目的と構成について説明します。
- Y **第 2 章: 「インテル® 64 プロセッサ・アーキテクチャーと IA-32 プロセッサ・アーキテクチャー」:** IA-32 プロセッサ・ファミリーとインテル® 64 プロセッサ・ファミリーのマイクロアーキテクチャー、そしてソフトウェア最適化に関連したその他の機能について説明します。
- Y **第 3 章: 「一般的な最適化ガイドライン」:** コード開発と最適化の一般的な手法について説明します。これらの手法は、インテル® Core™ マイクロアーキテクチャー、拡張版インテル® Core™ マイクロアーキテクチャー、Intel NetBurst® マイクロアーキテクチャー、インテル® Pentium® M プロセッサ・マイクロアーキテクチャーの一般的な機能を利用するすべてのアプリケーションに適用されます。
- Y **第 4 章: 「SIMD アーキテクチャー向けのコーディング」:** インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3)、インテル® ストリーミング SIMD 拡張命令 4.1 (インテル® SSE4.1)でサポートされる、SIMD 整数命令と SIMD 浮動小数点命令を使用するための手法と概念について説明します。
- Y **第 5 章: 「SIMD 整数アプリケーション向けの最適化」:** 128 ビット SIMD 整数命令を使用するアプリケーション向けの最適化のヒントと共通ビルディング・ブロックについて説明します。
- Y **第 6 章: 「SIMD 浮動小数点アプリケーション向けの最適化」:** 単精度および倍精度 SIMD 浮動小数点命令を使用するアプリケーション向けの最適化のヒントと共通ビルディング・ブロックについて説明します。
- Y **第 7 章: 「キャッシュ利用の最適化」:** PREFETCH 命令とキャッシュ制御命令を使用してキャッシュ利用とキャッシュ・パラメーターを最適化する方法について説明します。
- Y **第 8 章: 「サブ NUMA クラスタリングの概要」:** ラスト・レベル・キャッシュ (LLC) からローカルメモリーまでの平均レイテンシーを改善する、サブ NUMA クラスタリング (SNC) モードについて説明します。
- Y **第 9 章: 「マルチコアとハイパースレッディング・テクノロジー」:** 最高の性能スケールンが得られるようにマルチスレッド・アプリケーションを最適化するガイドラインや手法を説明します。これらのガイドラインや手法は、マルチコア・プロセッサ、ハイパースレッディング・テクノロジー対応プロセッサ、またはマルチプロセッサ (MP) システムが対象の場合に適用されます。
- Y **第 10 章: 「64 ビット・モードのコーディング・ガイドライン」:** アプリケーション・ソフトウェアが 64 ビット・モードで動作するように作成するための、追加のコーディング・ガイドラインについて説明します。
- Y **第 11 章: 「テキスト処理/字句解析/構文解析におけるインテル® SSE4.2 と SIMD プログラミング」:** インテル® SSE4.2 とその他の拡張命令を使用してテキスト/文字列処理や字句/構文解析のアプリケーションを改善する SIMD 手法について説明します。
- Y **第 12 章: 「インテル® アドバンスド・ベクトル・エクステンションの最適化」:** これまでの世代の 128 ビットのインテル® SSE ベクトル命令の機能を拡張するとともに、ベクトルのレジスター幅を拡張することで 256 ビットの演算をサポートする、インテル® アドバンスド・ベクトル・エクステンション (インテル® AVX) の最適化について説明します。
- Y **第 13 章: 「インテル® AVX-512 向けの最適化」:** インテル® アドバンスド・ベクトル・エクステンション 512 を使用するアプリケーション向けの最適化の推奨事項と共通のビルディング・ブロックについて説明します。
- Y **第 14 章: 「インテル® トランザクショナル・シンクロナイゼーション・エクステンションの推奨事項」:** 競合するロックを持つマルチスレッド・ソフトウェアを最適化するため、インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) とロックの省略を使用するチューニングの推奨事項を説明します。
- Y **第 15 章: 「モバイル利用時における電力の最適化」:** モバイル・プロセッサの省電力手法に関する背景説明を行い、バッテリー持続時間の延長のために開発者が利用できる推奨事項を紹介します。
- Y **第 16 章: 「インテル® マイクロアーキテクチャー Silvermont<sup>†</sup> と Goldmont<sup>†</sup> ソフトウェアの最適化」:** インテル® マイクロアーキテクチャー開発コード名 Silvermont ベースのプロセッサ・ファミリーのマイクロアーキテクチャーと、インテル® マイクロアーキテクチャー Silvermont<sup>†</sup> ベースのプロセッサ向けのソフトウェア最適化技法について説明します。
- Y **第 17 章: 「インテル® マイクロアーキテクチャー Knights Landing<sup>†</sup> とソフトウェアの最適化」:** Knights Landing<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサ・ファミリーのマイクロアーキテクチャーと、Knights Landing<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサ向けのソフトウェア最適化技法について説明します。

- Y **付録 A: 「アプリケーション・パフォーマンス・ツール」:** アセンブリー・コードを記述することなく、アプリケーション・パフォーマンスの分析と強化を行うツールについて説明します。
- Y **付録 B: 「パフォーマンス監視イベントの使用」:** トップダウン解析でもたらされる情報と、インテル® Xeon® プロセッサ 5500 番台、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge ベースのプロセッサ、およびインテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサ固有のパフォーマンス監視イベントにより収集される情報の使用法について説明します。
- Y **付録 C: 「IA-32 命令のレイテンシーとスループット」:** IA-32 命令のレイテンシーとスループットのデータを示します。また最近のプロセッサ・ファミリー固有の命令タイミングデータについて説明します。
- Y **付録 D: 「Intel Atom® マイクロアーキテクチャーとソフトウェアの最適化」:** Intel Atom® マイクロアーキテクチャー・ベースのプロセッサ・ファミリーのマイクロアーキテクチャーと、Intel Atom® マイクロアーキテクチャー・ベースのプロセッサ向けのソフトウェア最適化技法について説明します。

## 1.3 関連情報

インテル® アーキテクチャー、手法、プロセッサ・アーキテクチャーの用語に関する詳細は、以下の資料を参照してください。

- Y 『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』 (全 5 巻)
- Y 『Developing Multi-threaded Applications: A Platform Consistent Approach』
- Y インテル® C++ コンパイラーのドキュメントとオンラインヘルプ
- Y インテル® Fortran コンパイラーのドキュメントとオンラインヘルプ
- Y インテル® VTune™ Amplifier のドキュメントとオンラインヘルプ
- Y 『Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor MP』

その他の関連リンクを以下に示します。

- Y インテル® デベロッパー・ゾーン:  
<https://software.intel.com/en-us/all-dev-areas> (英語)
- Y プロセッサ・サポートの全般リンク:  
<http://www.intel.com/support/processors/> (英語)
- Y インテル® マルチコア・テクノロジー:  
<https://software.intel.com/en-us/articles/multi-core-introduction> (英語)
- Y インテル® ハイパースレッディング・テクノロジー (HT テクノロジー):  
<http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyperthreading-technology.html> (英語)
- Y インテル® SSE4.1 アプリケーション・ノート: インテル® ストリーミング SIMD 拡張命令 4 による動き予測:  
<https://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4> (英語)
- Y インテル® SSE4 プログラミング・リファレンス:  
<https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf> (英語)
- Y インテル® 64 アーキテクチャー・プロセッサ・トポロジー:  
<https://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration> (英語)
- Y SIMD 拡張命令を使用した複数バッファリング技術:  
<http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communicationsia-multi-buffer-paper.pdf> (英語)
- Y 複数バッファリングを使用した並列ハッシュ処理:  
<http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995> (英語)  
<http://eprint.iacr.org/2012/476.pdf> (英語)
- Y インテル® AES-NI ライブラリーのサンプルコード:  
<http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/> (英語)
- Y PCMMULQDQ に関するリソース:  
<https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode> (英語)



Y 冗長表現とインテル® AVX2 を使用した冪剰余:

[http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3\\_9?LI=true](http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true) (英語)

本章では、最新世代のインテル® 64 プロセッサと IA-32 プロセッサ (インテル® マイクロアーキテクチャー Skylake Server<sup>†</sup>、インテル® マイクロアーキテクチャー Skylake<sup>†</sup>、インテル® マイクロアーキテクチャー Broadwell<sup>†</sup>、インテル® マイクロアーキテクチャー Haswell<sup>†</sup>、インテル® マイクロアーキテクチャー Ivy Bridge<sup>†</sup>、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup>、インテル® Core™ マイクロアーキテクチャー、拡張版インテル® Core™ マイクロアーキテクチャー、インテル® マイクロアーキテクチャー Nehalem<sup>†</sup> ベースのプロセッサ) におけるソフトウェア最適化に関連するプロセッサの機能について説明します。これらの機能には、以下のものが含まれます。

- Y 高クロックレートかつ高スループットでの命令実行が可能なマイクロアーキテクチャー、高速なキャッシュ階層、高速システムバス
- Y インテル® Core™ プロセッサ・ファミリーとインテル® Xeon® プロセッサ・ファミリーで利用可能なマルチコア・アーキテクチャー
- Y インテル® ハイパースレッディング・テクノロジー<sup>1</sup> (HT テクノロジー) のサポート
- Y インテル® 64 プロセッサのインテル® 64 アーキテクチャー
- Y SIMD 拡張命令: インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3)、インテル® ストリーミング SIMD 拡張命令 4.1 (インテル® SSE4.1)、インテル® ストリーミング SIMD 拡張命令 4.2 (インテル® SSE4.2)
- Y インテル® アドバンスド・ベクトル・エクステンション (インテル® AVX)
- Y 半精度浮動小数点変換と RDRAND
- Y 乗算加算融合(FMA)拡張
- Y インテル® アドバンスド・ベクトル・エクステンション 2 (インテル® AVX2)
- Y ADX と RDSEED
- Y インテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512)

インテル® Core™2 プロセッサ・ファミリー、インテル® Core™2 Extreme プロセッサ・ファミリー、インテル® Core™2 Quad プロセッサ・ファミリー、インテル® Xeon® プロセッサ 3000/3200/5100/5300/7300 番台は、電力効率に優れた高性能のインテル® Core™ マイクロアーキテクチャーをベースにしています。インテル® Xeon® プロセッサ 3100/3300/5200/5400/7400 番台、インテル® Core™2 Extreme プロセッサ QX9600/Q9700 番台、インテル® Core™2 Quad プロセッサ Q9000/Q8000 番台は拡張版インテル® Core™ マイクロアーキテクチャーをベースにしています。インテル® Core™ i7 プロセッサは、インテル® マイクロアーキテクチャー Nehalem<sup>†</sup> をベースにしています。インテル® Xeon® プロセッサ 5600 番台、インテル® Xeon® プロセッサ E7 ファミリーとインテル® Core™ i7/i5/i3 プロセッサは、インテル® マイクロアーキテクチャー Westmere<sup>†</sup> をベースにしています。

インテル® Xeon® プロセッサ E5 ファミリー、インテル® Xeon® プロセッサ E3-1200 製品ファミリー、インテル® Xeon® プロセッサ E7-8800/4800/2800 製品ファミリー、インテル® Core™ i7-3930K プロセッサ、および第 2 世代インテル® Core™ i7-2xxx プロセッサ・シリーズ、第 2 世代インテル® Core™ i5-2xxx プロセッサ・シリーズ、第 2 世代インテル® Core™ i3-2xxx プロセッサ・シリーズは、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> をベースにしています。

インテル® Xeon® プロセッサ E3-1200 v2 製品ファミリーと第 3 世代インテル® Core™ プロセッサは、Ivy Bridge<sup>†</sup> マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。インテル®

<sup>1</sup> ハイパースレッディング・テクノロジーを利用するには、ハイパースレッディング・テクノロジーに対応したインテル® プロセッサを搭載したコンピューター・システム、および同技術に対応したチップセットと BIOS、OS が必要です。性能は使用するハードウェアやソフトウェアによって異なります。

Xeon® プロセッサ E5 v2 ファミリーとインテル® Xeon® プロセッサ E7 v2 ファミリーは、Ivy Bridge-E<sup>+</sup> マイクロアーキテクチャーをベースにし、インテル® 64 とプラットフォームで複数の物理プロセッサ・パッケージをサポートします。

インテル® Xeon® プロセッサ E3-1200 v3 製品ファミリーと第 4 世代インテル® Core™ プロセッサは、Haswell<sup>+</sup> マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。インテル® Xeon® プロセッサ E5-26xx v3 製品ファミリーは、Haswell-E<sup>+</sup> マイクロアーキテクチャーをベースにし、インテル® 64 とプラットフォームで複数の物理プロセッサ・パッケージをサポートします。

インテル® Core™ M プロセッサ、第 5 世代インテル® Core™ プロセッサ、およびインテル® Xeon® プロセッサ E3-1200 v4 製品ファミリーは、Broadwell<sup>+</sup> マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。

第 6 世代インテル® Core™ プロセッサ、インテル® Xeon® プロセッサ E3-1500M v5 製品ファミリーは、Skylake<sup>+</sup> マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。

インテル® Xeon® プロセッサ・スケーラビリティ・ファミリーは、Skylake Server<sup>+</sup> マイクロアーキテクチャーをベースにし、インテル® 64 アーキテクチャーをサポートします。

## 2.1 SKYLAKE SERVER<sup>+</sup> マイクロアーキテクチャー

インテル® Xeon® スケーラブル・プロセッサ・ファミリーは、Skylake Server<sup>+</sup> マイクロアーキテクチャーをベースとしています。Skylake<sup>+</sup> マイクロアーキテクチャー・ベースのプロセッサは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 4』の第 2 章の表 2-1 に記載される CPUID の DisplayFamily\_DisplayModel シグネチャーを使用して検出することができます。

Skylake Server<sup>+</sup> マイクロアーキテクチャーは、アプリケーションのパフォーマンスと電力消費を最適化するため、次の新機能<sup>2</sup>を導入しています。

- Y Skylake Server<sup>+</sup> マイクロアーキテクチャー・ベースの新しいコアは、Kaby Lake<sup>+</sup> マイクロアーキテクチャーを基に改善を行っています。
- Y インテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512) のサポート。
- Y ソケットあたりのコア数の増加 (最大 28 対最大 22)。
- Y Skylake<sup>+</sup> マイクロアーキテクチャーではソケットあたり 6 本のメモリーチャンネル (Broadwell<sup>+</sup> マイクロアーキテクチャーは 4 チャンネル)。
- Y より大きな L2 キャッシュと小さな非インクルーシブな L3 キャッシュ。
- Y インテル® Optane™ テクノロジーをサポート。
- Y インテル® Omni-Path アーキテクチャー (インテル® OPA)。
- Y サブ NUMA クラスタリング (SNC) をサポート。

図 2-1 に記される星印は、クライアント向け Skylake<sup>+</sup> マイクロアーキテクチャーに対する Skylake Server<sup>+</sup> マイクロアーキテクチャーでの新機能を示します: 1MB の L2 キャッシュとポート 5 に追加されたインテル® AVX-512 ユニット (製品によって利用可能)。

ポート 0 とポート 1 は 256 ビット幅であるため、インテル® AVX-512 操作はポート 0 にディスパッチされポート 0 とポート 1 の両方で実行されます。しかし、*lea* などのほかの操作はポート 1 で並列に実行できます。ポート 0 と 1 のフュージョンは図 2-1 の赤い囲みで行われます。

クライアント版の Skylake<sup>+</sup> マイクロアーキテクチャーとは異なり、Skylake Server<sup>+</sup> マイクロアーキテクチャーでは、フロントエンドのループストリーム検出器 (LSD) が無効化されていることに注意してください。

<sup>2</sup> いくつかの機能はすべてのプロセッサ上で利用できるとは限りません。

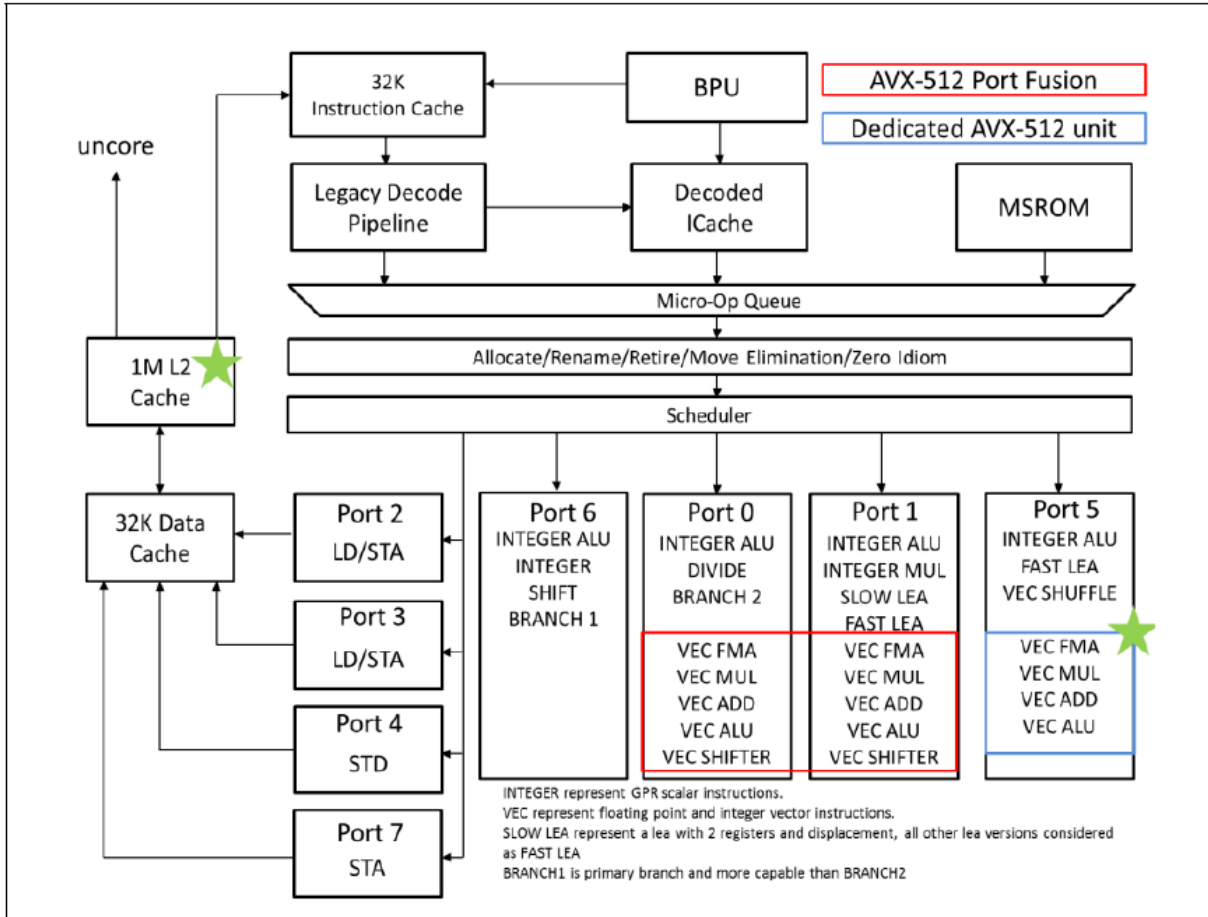


図 2-1 Skylake Server+ マイクロアーキテクチャーの CPU コア・パイプラインの機能

## 2.1.1 Skylake Server+ マイクロアーキテクチャーのキャッシュ

Skylake Server+ マイクロアーキテクチャー・ベースのインテル® Xeon® スケーラブル・プロセッサ・ファミリーは、以前の世代の Broadwell+ マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサと比較していくつかのコンポーネントのパフォーマンスとスケーラビリティを改善するためコアとアンコア・アーキテクチャーが大幅に変更されています。

### 2.1.1.1 より大きな中間レベルキャッシュ

Skylake Server+ マイクロアーキテクチャーは、1MB の容量を持つ中間レベル (L2) を実装し、読み出してから使用するまで最小 14 サイクルのレイテンシーを提供します。この中間レベルキャッシュは、以前の世代のインテル® Xeon® プロセッサにおける実装より 4 倍の容量を持っています。中間レベルのキャッシュのラインサイズは 64 バイトで 16 ウェイの連想性を備えています。中間レベルのキャッシュはそれぞれのコアでプライベートです。

中間レベルのキャッシュにデータを収めるように最適化されているソフトウェアは、Skylake Server+ マイクロアーキテクチャーの容量が増えた中間レベルキャッシュの利点を得るため修正する必要があるかもしれません。

### 2.1.1.2 非インクルーシブなラスト・レベル・キャッシュ (LLC)

Skylake Server+ のラスト・レベル・キャッシュは、非インクルーシブな分散方式の共有キャッシュです。ラスト・レベル・キャッシュの各バンクサイズは、バンクごとに 1.375MB にシュリンクされています。ラスト・レベル・キャッシュが非インクルーシブであることから、あるコアの中間レベルキャッシュに送られたブロックは、ラスト・レベル・キャッシュ

のバンクにコピーを持たない可能性があります。アクセスパターン、アクセスされたコードとデータのサイズ、そしてコア間でのキャッシュブロック共有の挙動に基づいて、ラスト・レベル・キャッシュは中間レベルキャッシュのビクティムキャッシュのように見え、コアごとの総キャッシュ容量はそれぞれのコアのプライベートな中間レベルキャッシュとラスト・レベル・キャッシュの部分的な組み合わせとして見えるかもしれません。

### 2.1.1.3 Skylake Server<sup>†</sup> マイクロアーキテクチャーにおけるキャッシュの推奨事項

Skylake Server<sup>†</sup> マイクロアーキテクチャーのキャッシュと以前の Broadwell<sup>†</sup> マイクロアーキテクチャー世代のキャッシュの簡単な比較を表に示します。

表 2-1 Skylake<sup>†</sup> マイクロアーキテクチャーと Broadwell<sup>†</sup> マイクロアーキテクチャーのキャッシュの比較

キャッシュレベル	カテゴリー	Broadwell <sup>†</sup> マイクロアーキテクチャー	Skylake Server <sup>†</sup> マイクロアーキテクチャー
L1 データ・キャッシュ・ユニット (DCU)	サイズ [KB]	32	32
	レイテンシー [サイクル]	4-6	4-6
	最大帯域幅 [バイト/サイクル]	96	192
	持続帯域幅 [バイト/サイクル]	93	133
	連想性 [ウェイ]	8	8
L2 中間キャッシュ (MLC)	サイズ [KB]	256	1024 (1MB)
	レイテンシー [サイクル]	12	14
	最大帯域幅 [バイト/サイクル]	32	64
	持続帯域幅 [バイト/サイクル]	25	52
	連想性 [ウェイ]	8	16
L3 ラスト・レベル・キャッシュ (LLC)	サイズ [KB]	最大 2.5	最大 1.375 <sup>1</sup>
	レイテンシー [サイクル]	50-60	50-70
	最大帯域幅 [バイト/サイクル]	16	16
	持続帯域幅 [バイト/サイクル]	14	15

**注意:**

1. Skylake Server<sup>†</sup> 製品ではいくつかのコアが無効化されて、コアあたり 1.375MB 以上の L3 キャッシュを保持するものがあります。

次の表は、Skylake Server<sup>†</sup> マイクロアーキテクチャーのメモリー・バランスが、高いレイテンシーの共有分散方式から低レイテンシーのプライベート・ローカル方式に移行していることを示しています。

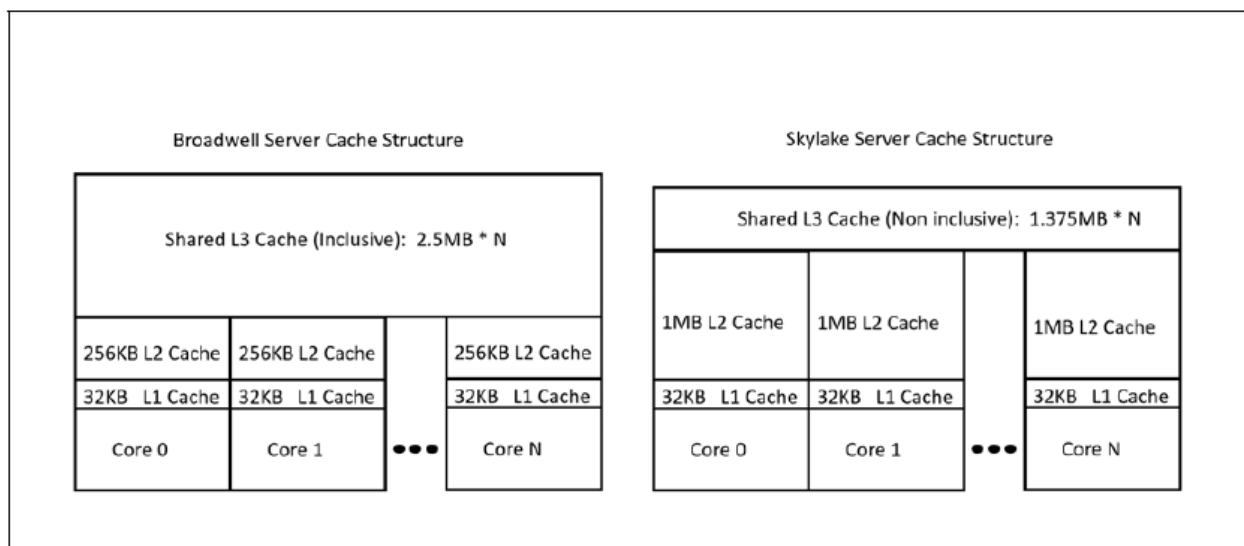


図 2-2 Broadwell<sup>+</sup> マイクロアーキテクチャーと Skylake Server<sup>+</sup> マイクロアーキテクチャーのキャッシュ構造

このキャッシュ・アーキテクチャーの変更から得られるパフォーマンス・ゲインの可能性は高く、ソフトウェアは新しいキャッシュサイズに適合するようにメモリーのタイル化を導入する必要があります。

**推奨事項:** アプリケーションの共有およびプライベート・データのサイズを、小さな非インクルーシブ L3 キャッシュと大きな L2 キャッシュに合わせて再度バランスを取ります。

キャッシュ・ブロッキングは、アプリケーションの帯域幅要件と、アプリケーション間の切り換えに基づいて選択する必要があります。4 倍の L2 キャッシュサイズと 2 倍の L2 キャッシュ帯域幅は、以前の Broadwell<sup>+</sup> マイクロアーキテクチャー世代と比較すると、アプリケーションによっては L1 に変わって L2 へのブロック化することを可能にし、それによりパフォーマンスを向上させます。

**推奨事項:** L2 キャッシュがアプリケーションの帯域幅要件を満たすことができるのであれば、Skylake Server<sup>+</sup> マイクロアーキテクチャーでは L2 へのブロック化を検討します。

ラスト・レベル・キャッシュがインクルーシブから非インクルーシブに変更されたことは、中間レベルキャッシュとラスト・レベル・キャッシュの容量を合計できることを意味します。実行時にコアごとのキャッシュ容量を求めるプログラムでは、有効なキャッシュサイズを算出するためコアごとに中間レベルキャッシュとラスト・レベル・キャッシュのサイズを合計します。コアごとにラスト・レベル・キャッシュのサイズをだけを算出すると、利用可能なオンチップキャッシュを適切に利用できない可能性があります。詳細は、2.1.2 節をご覧ください。

**推奨事項:** データを共有しない場合、アプリケーションは コアあたりのキャッシュ容量を L3 キャッシュだけでなく L2 と L3 キャッシュサイズの合計であると見なすべきです。

## 2.1.2 Skylake Server<sup>+</sup> マイクロアーキテクチャー上での非テンポラルなストア

Skylake Server<sup>+</sup> マイクロアーキテクチャーにおいてラスト・レベル・キャッシュの各バンクのサイズが変更されたことにより、アプリケーション、ライブラリー、またはドライバーがコアごとのオンチップキャッシュのサイズを決定するためラスト・レベル・キャッシュだけを考慮していると、Skylake Server<sup>+</sup> マイクロアーキテクチャーでは減少して見え、小さなブロックのメモリー書き込みとともに非テンポラルなストアを使用する可能性があります。非テンポラルなストアはキャッシュラインを追い出してメモリーに書き戻しを行うため、以前の世代のインテル® Xeon® プロセッサ・ファミリーと比べると、Skylake Server<sup>+</sup> マイクロアーキテクチャーでは以降のキャッシュミスとメモリー帯域幅の要求が増加するかもしれません。

また、Skylake Server<sup>+</sup> マイクロアーキテクチャーによる非テンポラルなストアによって生じるアクセスの扱いが変更されたことにより、以前の世代のインテル® Xeon® プロセッサ・ファミリーと比べると、同じようなアクセスを行っても各コア内のリソースが長時間ビジーのままになります。その結果、そのような命令シーケンスが実行されると、プロセッサはリソース不足からストールし、各コアからのメモリー書き込み帯域幅が制限される可能性があります。

非テンポラルなストアの多用からキャッシュミスが増加すると、非テンポラルなストアによるコアごとのメモリー書き込み帯域幅の制限からパフォーマンスが低下するアプリケーションもあります。

Skylake Server<sup>+</sup> マイクロアーキテクチャーで前述のパフォーマンスの問題を回避するには、アプリケーション、ライブラリー、およびドライバーが各コアで利用可能なオンチップキャッシュを調査する際に、それぞれのコアのラスト・レベル・キャッシュに加え中間キャッシュの容量を含めます。Skylake Server<sup>+</sup> マイクロアーキテクチャーで利用可能なオンチップキャッシュの容量をこのように求めるのは、非インクルーシブなラスト・レベル・キャッシュが実装されていることを考慮しているためです。

## 2.2 Skylake<sup>+</sup> マイクロアーキテクチャー

Skylake<sup>+</sup> マイクロアーキテクチャーは、Haswell<sup>+</sup> および Broadwell<sup>+</sup> マイクロアーキテクチャーの成功の上に構築されています。図 2-3 に Skylake<sup>+</sup> マイクロアーキテクチャーの基本パイプライン機能を示します。

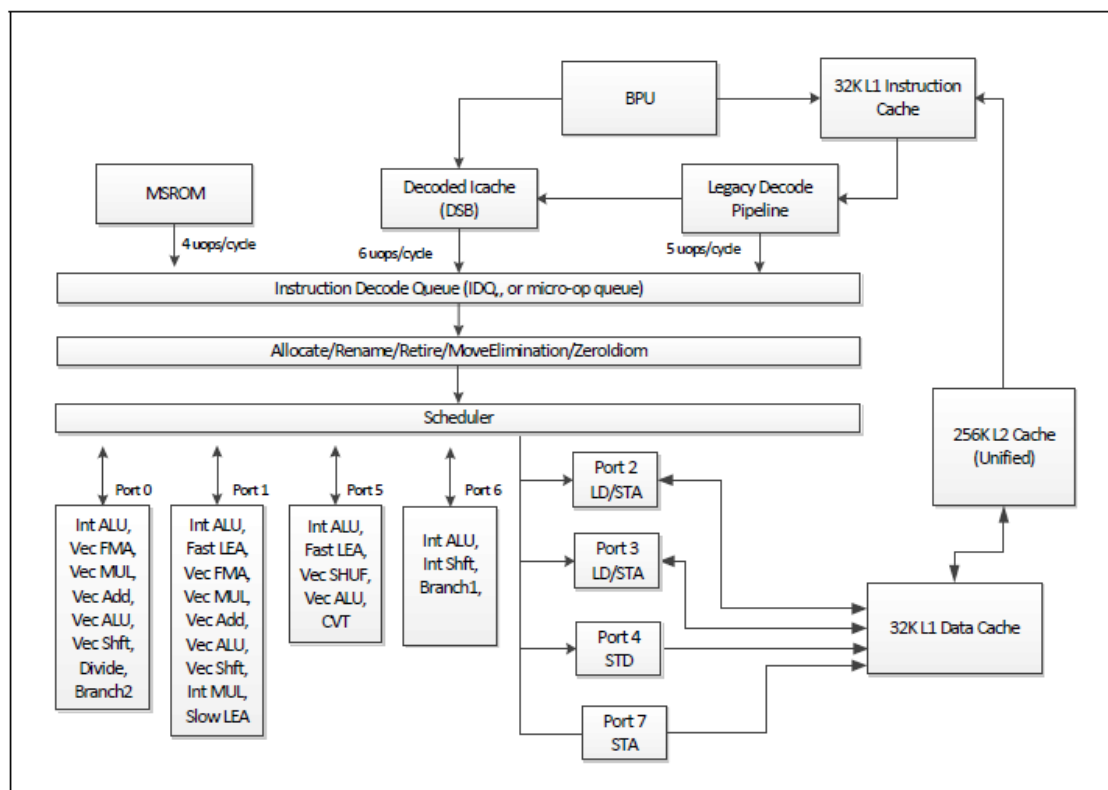


図 2-3 Skylake+ マイクロアーキテクチャーの CPU コア・パイプラインの機能

Skylake+ マイクロアーキテクチャーには次の拡張が含まれます。

- Y 深い OOO 実行と高いキャッシュ帯域幅を可能にする大きな内部バッファ。
- Y フロントエンドのスループットを改善。
- Y 分岐予測器の改善。
- Y 除算器のスループットとレイテンシーの改善。
- Y 低消費電力化。
- Y ハイパースレッディング・テクノロジーと SMT パフォーマンスの改善。
- Y バランスの取れた浮動小数点 ADD、MUL、FMA 命令のスループットとレイテンシー。

マイクロアーキテクチャーは、リング接続された複数スライスの L3 (オフダイの L4 はオプション)、プロセッサ・グラフィックス、統合メモリー・コントローラー、インターコネクト・ファブリックなどから構成される、共有アンコア・サブシステムと複数プロセッサ・コアの柔軟な統合をサポートします。4 コア構成では、図 2-5 に示す配置と同様の構成がサポートされます。

## 2.2.1 フロントエンド

Skylake+ マイクロアーキテクチャーのフロントエンドでは、1 世代前のマイクロアーキテクチャーに対し次のような改善が行われています。

- Y レガシー・デコード・パイプラインは、前世代の 4  $\mu\text{op}$  に対してサイクルあたり IDQ (命令デコードキュー) へ 5 つの  $\mu\text{op}$  を供給します。
- Y DSB (デコード済み命令キャッシュ) は、前世代の 4  $\mu\text{op}$  に対してサイクルあたり IDQ へ 6 つの  $\mu\text{op}$  を供給します。
- Y 同じコア上の 2 つの論理プロセッサがアクティブである場合、IDQ は論理プロセッサあたり 64  $\mu\text{op}$  を保持できます。以前の世代では 28  $\mu\text{op}$  しか保持できませんでした (コアごとに 2 x 64 vs. 2 x 28)。

- Y コア上で 1 つの論理プロセッサのみがアクティブである場合、IDQ は 64  $\mu$ op を保持できます (ST 操作において 64 vs. 56)。
- Y IDQ 内の LSD (ループ・ストリーム・ディテクター) は、ST または SMT 操作にかかわらず論理プロセッサあたり最大 64  $\mu$ op を検出できます。
- Y 分岐予測器の改善。

## 2.2.2 アウトオブオーダー実行エンジン

Skylake<sup>+</sup> マイクロアーキテクチャーのアウトオブオーダーと実行エンジンには、次の変更が含まれます。

- Y バッファーが拡大されたことで、前世代に比べ深い OOO 実行を可能にします。
- Y 除算/平方根および逆数近似のスループットとレイテンシーが改善されました。
- Y FMA ユニットで実行されるすべての操作のレイテンシーとスループットが同一になりました。
- Y 長いポーズ・レイテンシーは、さらに高い電力効率と SMT パフォーマンス・リソースの利用を可能にします。
- Y 表 2-2 は、異なる操作タイプを各種ポートへディスパッチする OOO エンジンの役割をまとめています。

表 2-2 Skylake<sup>+</sup> マイクロアーキテクチャーのディスパッチ・ポートと実行スタック

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

表 2-3 は、実行ユニットとこれらのユニットに関連する代表的な命令を示します。インテル® SSE、インテル® AVX、および汎用命令セット全体のスループット向上は、対応する命令向けのユニット数、および特定のユニットを使用して実行する命令のパリエーションに関連しています。



表 2-3 Skylake<sup>+</sup> マイクロアーキテクチャーの実行ユニットと主要な命令<sup>1</sup>

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)blendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtsi2sd
Shuffle	1	(v)shufp*, (v)perm*, (v)pack*, (v)unpck*, (v)punpck*, (v)psluf*, (v)pslldq, (v)alignr, (v)pmovzx*, (v)pbroadcast*, (v)pslldq, (v)psrldq, (v)blendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

注意:

1. インテル® MMX® 命令にマッピングされる実行ユニットは、この表ではカバーされていません。12.16.5 節のインテル® MMX® 命令スレープットの制限に対するインテル® AVX2 変換の対策をご覧ください。

インテル® SSE、インテル® AVX、および汎用命令の重要な部分では、レイテンシーも改善されています。付録 C に詳細を示します。ソフトウェアから見えるレイテンシーは、生産側のマイクロオペレーション (μop) のフローと消費側の μop のフロー間の関係に依存して、追加の遅延を含む可能性があります。例えば、VPMULLD などの 2 μop 命令は、それぞれの 2 μop VPMULLD から 1 サイクルのバイパス遅延を累積する可能性があります。

表 2-4 は、生産側の μop と消費側の μop 間の、サイクルにおけるバイパス遅延を示しています。最左の列は、生産側の μop のさまざまな状況における特性を示します。最上位行は、消費側の μop のさまざまな状況における特性を示します。

表 2-4 生産側と消費側の μop 間のバイパス遅延

	SIMD/0,1/ 1	FMA/0,1/ 4	VIMUL/0,1/ 4	SIMD/5/1,3	SHUF/5/1, 3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

バイパスにおける生産/消費の μop に関連する属性は、省略形/1 つ以上のポート数/μop のレイテンシー・サイクルの 3 項目です。次に例を示します。

- Y “SIMD/0,1/1” は、1 サイクルのベクトル SIMD  $\mu\text{op}$  が、ポート 0 または 1 のいずれかにディスパッチされることを示します。
- Y “VIMUL/0,1/4” は、4 サイクルのベクトル整数乗算  $\mu\text{op}$  が、ポート 0 または 1 のいずれかにディスパッチされることを示します。
- Y “SIMD/5/1,3” は、1 もしくは 3 サイクルの非シャッフル  $\mu\text{op}$  が、ポート 5 にディスパッチされることを示します。

### 2.2.3 キャッシュとメモリー・サブシステム

Skylake<sup>+</sup> マイクロアーキテクチャーのキャッシュ階層では、次のような拡張が行われています。

- Y 前世代に比べ高いキャッシュ帯域幅。
- Y バッファ数が増えたことにより、多くのロードとストアを同時に処理することが可能になりました。
- Y Haswell<sup>+</sup> マイクロアーキテクチャーやそれ以前の世代と比べて、プロセッサは並列に 2 つのページウォークができるようになりました。
- Y ページ分割ロードのペナルティーは、前世代の 100 サイクルから大幅に軽減され 5 サイクルになりました。
- Y L3 の書き込み帯域幅は、前世代の 1 ラインあたり 4 サイクルから、2 ラインあたり 4 サイクルに増加しています。
- Y CLFLUSHOPT 命令でキャッシュラインをフラッシュし、SFENCE 命令を使用してフラッシュされたデータのメモリー順序を管理することが可能になりました。
- Y NULL ポインターを指定するソフトウェア・プリフェッチのパフォーマンス・ペナルティーが軽減されました。
- Y L2 の連想性が 8 ウェイから 4 ウェイに変更されました。

表 2-5 Skylake<sup>+</sup> マイクロアーキテクチャーのキャッシュ・パラメーター

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency <sup>1</sup>	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

**注意:**

1. ソフトウェアから見えるレイテンシーは、アクセスパターンとその他の要因に依存するため異なります。

TLB の階層は、命令キャッシュ向けの TLB、L1D 向けの TLB、さらに L2 向けのユニファイド TLB から成ります。表 2-6 の Partition (パーティション) カラムは、ハイパースレッディング・テクノロジーが有効である際のリソース共有ポリシーを示します。

表 2-6 Skylake<sup>+</sup> マイクロアーキテクチャーの TLB パラメーター

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

## 2.2.4 Skylake<sup>+</sup> マイクロアーキテクチャーのポーズ・レイテンシー

PAUSE 命令は、一般に、同一プロセッサ・コアに配置される 2 つの論理プロセッサで実行されるソフトウェア・スレッドが、ロックの開放を待機する際に使用されます。このような短いループは、数十から数百サイクルで終了する傾向があります。そのためパフォーマンスの観点からは、OS に任せるよりは短時間 CPU を占有する方が望ましいでしょう。待機ループが数千サイクル以上続くことが予測される場合、Windows\* の WaitForSingleObject や Linux\* の futex などの OS が提供する同期 API 関数を呼び出し、OS に任せことが推奨されます。

PAUSE 命令は次のことを意図します。

- Y 共有ハードウェア・リソースを競合する兄弟論理プロセッサ（スピルループを抜けて続行する準備ができている）を一時的に提供します。マイクロアーキテクチャー的にリソースを競合しながら共有する兄弟論理プロセッサは、次に示す Skylake<sup>+</sup> マイクロアーキテクチャーを利用できます。
  - デコード済み命令キャッシュ、LSD、および IDQ のフロントエンド・スロット
  - RS の実行スロット
- Y 次の構成で等価なスピルループ命令シーケンスを実行するのと比べると、プロセッサ・コアで消費される電力を節約します。
  - 1 つの論理プロセッサがアクティブではない（例えば、C ステートに入っている）
  - 同じコアの両方の論理プロセッサが PAUSE 命令を実行する
  - HT が無効化されている（BIOS オプションを使用して）

前の世代のマイクロアーキテクチャーの PAUSE 命令のレイテンシーは、およそ 10 サイクルでしたが、Skylake<sup>+</sup> マイクロアーキテクチャーでは最大 140 サイクルまで拡大しています。

レイテンシーが増加したことは、高度にスレッド化されたアプリケーションにおいてはわずかですが 1-2% のパフォーマンスが向上します（続行する準備が整っている論理プロセッサに、競合しながら共有されるマイクロアーキテクチャー上のリソースをより効率良く利用することを可能にします）。進行が固定回数のループ内の PAUSE 命令によってブロックされていない場合、それほどスレッド化されていないアプリケーションではごくわずかな影響を受けると予想されます。2 コアや 4 コアのシステムでは消費電力の利点もわずかです。

PAUSE 命令のレイテンシーがかなり増加したことで、PAUSE のレイテンシーに影響を受けやすいワークロードはパフォーマンスの低下を被ります。

以下の例では、動的な反復カウントを持つループでの PAUSE 命令の使い方を示します。Skylake<sup>+</sup> マイクロアーキテクチャーでは、RDTSC 命令は現在のプロセッサ・クロック（INVARIANT TSC 特性を参照）とは独立してマシンの保証された P1 周波数をカウントします。そのため、インテル® ターボブーストが有効である場合、遅延は一定のままですが実行された可能性がある命令数は変化します。

ロックで PollDelay 関数を使用して、保証された P1 周波数サイクルだけ待機します（“clocks” 変数で指定される）。

例 2-1 動的なポーズループの例

```
#include <x86intrin.h>
#include <stdint.h>

/* ラップする可能性があるタイムスタンプを処理する便利な述語関数。
a は b の前か? タイムスタンプがラップされる可能性があるため、
a から b へ時計回りにすべきか、逆回りにすべきかを提起しています。
時計回りの方が反時計回りよりも時間が短い場合、
将来であり、その他は過去です。例えば、a= MAX-1、b = MAX +1 (=0)
の場合、a > b (真) は a が b に到達したことを意味しません。
signed(a) = -2、signed(b) = 0 は、実際の差を示します */

static inline bool before(uint64_t a, uint64_t b)
{
    return ((int64_t)b - (int64_t)a) > 0;
}

void pollDelay(uint32_t clocks)
{
    uint64_t endTime = _rdtsc()+ clocks;
    for (; before(_rdtsc(), endTime); )
        _mm_pause();
}
```

以下のベースラインの例で示す競合スピンロックでは、マシン上のスレッド間で引き起こされる競合によるパフォーマンスの低下を避けるため、ロックがビジーである場合、指数バックオフを推奨します。これは、マシン上のスレッド数を増やし、競合状態を悪化させる可能性があるアーキテクチャーを変更する場合さらに重要となります。共有メモリを持つ複数ソケットのインテル® サーバー・プロセッサでは、同じロックを使用するスレッド数が増えるに従って、スレッド間の競合を解決する時間はさらに長くなります。指数バックオフはパフォーマンス低下の可能性を避けるため、スレッド間の競合を回避するように設計されています。以下の例では、チューニングの対象である MAX\_BACKOFF に到達するまで、PAUSE 命令の数は 2 倍に増加することに注意してください。

例 2-2 バックオフを増やした競合ロックの例

```
/* **** */
/* ベースライン版      */
/* **** */

// atomic {if (lock == free) ならロック状態をビジーに変更 }
while (cmpxchg(lock, free, busy) == fail)
{
    while (lock == busy)
    {
        __asm__ ("pause");
    }
}
```

```

/*****
/* 改善版          */
/*****

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
    while (lock == busy)
    {
        for (int i=mask; i; --i){
            __asm__ ("pause");
        }
        mask = mask < max ? mask<<1 : max;
    }
}

```

### 2.3 インテル® マイクロアーキテクチャー Haswell<sup>†</sup>

Haswell<sup>†</sup> マイクロアーキテクチャーは、Sandy Bridge<sup>†</sup> および Ivy Bridge<sup>†</sup> マイクロアーキテクチャーの成功の上に構築されています。図 2-4 に Haswell<sup>†</sup> マイクロアーキテクチャーの基本パイプライン機能を示します。一般に、2.3.1 節から 2.3.4 節で示される多くの機能は、Broadwell<sup>†</sup> マイクロアーキテクチャーにも適用されます。Broadwell<sup>†</sup> マイクロアーキテクチャーの拡張は、2.3.6 節にまとめられています。

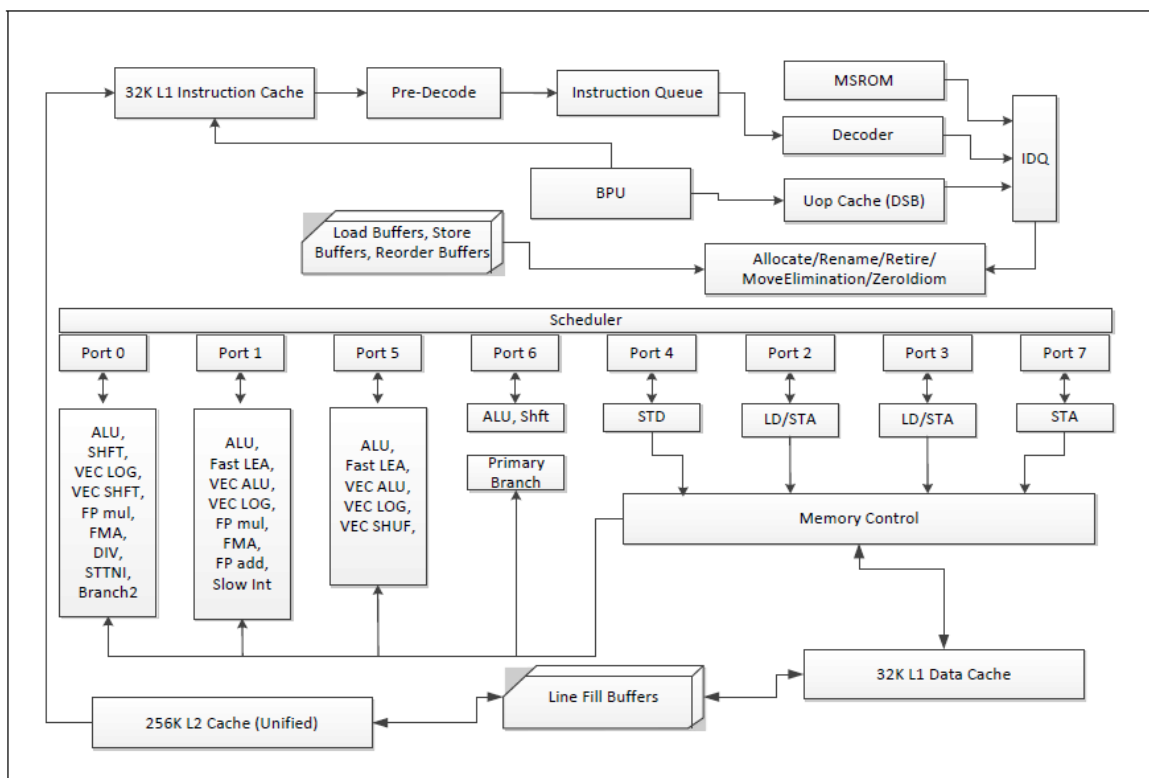


図 2-4 インテル® マイクロアーキテクチャー Haswell<sup>†</sup> の CPU コア・パイプライン

この新しいマイクロアーキテクチャーの基本パイプライン (図 2-2 を参照) は、以下の革新的な機能を提供します。

- Y インテル® アドバンスド・ベクトル・エクステンション 2 (インテル® AVX2)、FMA のサポート
- Y 整数値演算と暗号化を高速化する新しい汎用命令

- Y インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) のサポート
- Y 各コアでサイクルごとに最大 8 マイクロオペレーション (μop) をディスパッチ可能
- Y メモリー操作、FMA、インテル® AVX 浮動小数点実行ユニット、インテル® AVX2 整数実行ユニット用の 256 ビット・データ・パス
- Y L1 データキャッシュと L2 キャッシュの帯域幅が増加
- Y 2 つの FMA 実行パイプライン
- Y 4 つの数値演算ユニット (ALU)
- Y 3 つのストア・アドレス・ポート
- Y 2 つの分岐実行ユニット
- Y IA プロセッサ・コアおよびアンコア・サブシステム向けの高度な電力管理機能
- Y オプションの L4 キャッシュをサポート

インテル® マイクロアーキテクチャー Haswell<sup>†</sup> は、L3 (オプションでオフダイの L4 も) の複数のスライスへのリング・インターコネクト、プロセッサ・グラフィックス、統合型メモリー・コントローラー、インターコネクト・ファブリックなどを含むいくつかの要素で構成される共有アンコア・サブシステムと、複数のプロセッサ・コアとの柔軟な統合をサポートしています。図 2-5 に、4 CPU コアとアンコア要素で構成されるシステム統合の例を示します。

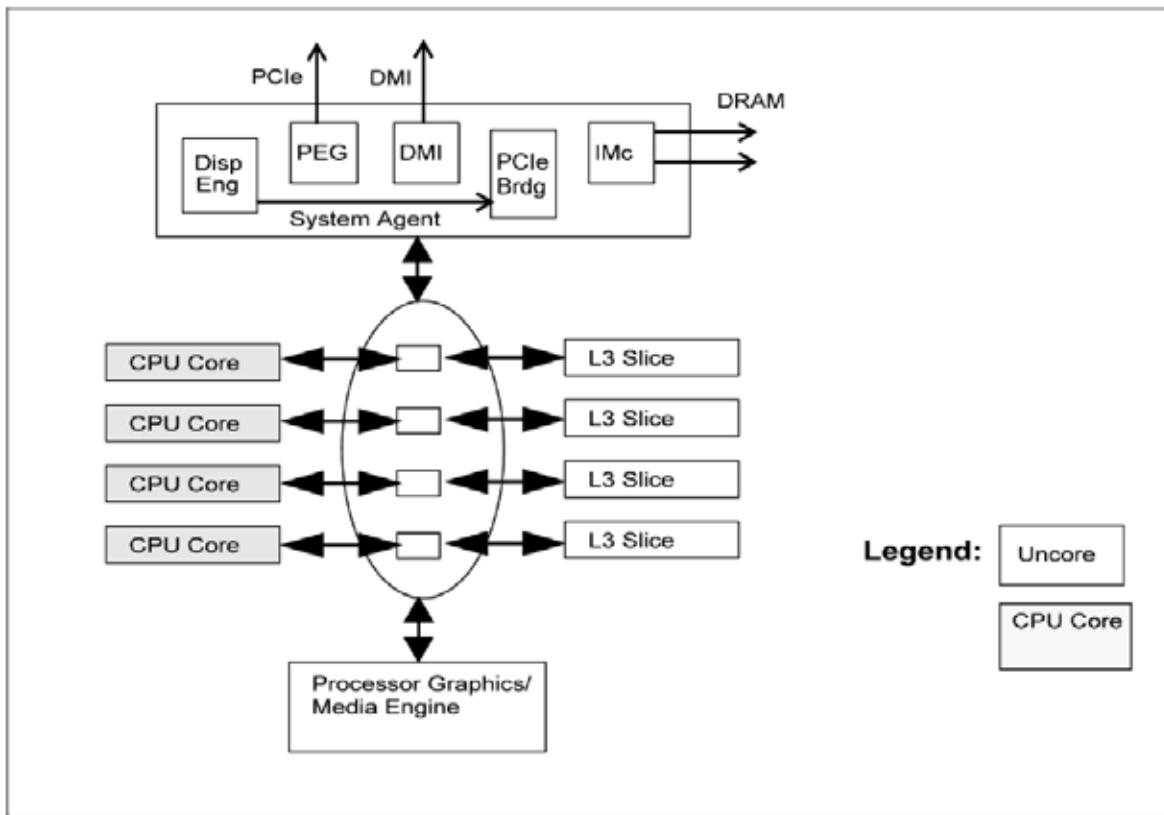


図 2-5 インテル® マイクロアーキテクチャー Haswell<sup>†</sup> の 4 コアのシステム統合

### 2.3.1 フロントエンド

インテル® マイクロアーキテクチャー Haswell<sup>†</sup> のフロントエンドは、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> (2.4.2 節) とインテル® マイクロアーキテクチャー Ivy Bridge<sup>†</sup> (2.4.7 節) をベースに開発され、次の点が拡張されています。

- Y マイクロオペレーション (μop) キャッシュ (またはデコード済み命令キャッシュ) は、2 つの論理プロセッサ間で均等に分割されます。
- Y 命令デコーダーは、アクティブな論理プロセッサ間で交互に使用される。1 つの論理プロセッサがアイドル状態の場合は、もう一方のアクティブな論理プロセッサがデコーダーを続けて使用されます。

Y ループストリーム検出器 (LSD)/マイクロオペレーション (μop) は、56 マイクロオペレーション (μop) までの小さなループを検出できます。56 エントリーのマイクロオペレーション (μop) キューは、ハイパースレッディング・テクノロジーが有効な場合、2 つの論理プロセッサによって共有されます (インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> では、各コアに 28 エントリーのマイクロオペレーション (μop) キューの複製が提供されます)。

## 2.3.2 アウトオブオーダー・エンジン

以下に、アウトオブオーダー・エンジンの主要構成要素と主な改善点を示します。

**リネーマー:** リネーマーは、マイクロオペレーション (μop) キューからスケジューラーのディスパッチ・ポートへマイクロオペレーション (μop) を移動し、実行リソースにバインドする。ゼロイディオム、1 イディオム、ゼロレイテンシーのレジスター移動命令はリネーマーによって実行され、スケジューラーと実行コアを解放することでパフォーマンスを向上できます。

**スケジューラー:** スケジューラーは、ディスパッチ・ポートへのマイクロオペレーション (μop) のディスパッチを制御します。アウトオブオーダー実行コアをサポートするため 8 つのディスパッチ・ポートがあり、そのうち 4 つは計算処理用の実行リソースを提供し、残り 4 つは 1 サイクルで最大 2 つの 256 ビット・ロード操作と 1 つの 256 ビット・ストア操作をサポートします。

**実行コア:** スケジューラーは、各ポートで 1 つずつ、サイクルごとに最大 8 つのマイクロオペレーション (μop) をディスパッチできます。計算リソースを提供する 4 つのポートには ALU が 1 つずつあり、実行パイプのうち 2 つは FMA ユニット専用です。除算/平方根を除き、STTNI (String and Text New Instructions)/インテル® AES-NI (Advanced Encryption Standard New Instructions) ユニット、ほとんどの浮動小数点および整数 SIMD 実行ユニットは 256 ビット幅です。メモリー操作用の 4 つのディスパッチ・ポートは、2 つのロード/ストアアドレス操作用のデュアルユース・ポート、ストアアドレス専用のポート、1 つのストアデータ専用ポートで構成されており、すべてのポートで 256 ビットのメモリー・マイクロオペレーション (μop) を処理できます。浮動小数点のピーク・スループットは、FMA を使用した場合、単精度では 1 サイクルあたり 32 マイクロオペレーション (μop)、倍精度では 16 マイクロオペレーション (μop) であり、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> の 2 倍です。

アウトオブオーダー・エンジンは、同時に 192 マイクロオペレーション (μop) を処理できます (インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> では 168 マイクロオペレーション (μop) です)。

## 2.3.3 実行エンジン

表 2-7 に、各ポートでディスパッチ可能なマイクロオペレーション (μop) を示します。

表 2-7 ディスパッチ・ポートと実行スタック

ポート 0	ポート 1	ポート 2, 3	ポート 4	ポート 5	ポート 6	ポート 7
ALU, Shift	ALU, Fast LEA,	Load_addr, Store_addr	Store_data	ALU, Fast LEA	ALU, Shift, JEU	Store_addr, Simple_AGU
SIMD_Log, STTNI, SIMD_Shifts	SIMD_ALU, SIMD_Log			SIMD_ALU, SIMD_Log		
FMA/FP_mul, Div	FMA/FP_mul, FP_add			FP/Int Shuffle		
2nd_Jeu	slow_int					

表 2-8 は、実行ユニットとこれらのユニットに関連する代表的な命令を示します。表 2-8 はまた、Broadwell<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサのみで利用可能な命令をいくつか含んでいます。

表 2-8 Haswell<sup>†</sup> マイクロアーキテクチャーの実行ユニットと主要な命令

Execution Unit	# of Ports	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa
SHFT	2	sal, shl, rol, adc, sarx, (adcx, adox) <sup>1</sup> etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blmsk, bzhi, etc
SIMD Log	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)blendp*, vblendd
SIMD_Shft	1	(v)psl*, (v)psr*
SIMD ALU	2	(v)padd*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)pcmpgt*
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)pblendw
SIMD Misc	1	(v)pmul*, (v)pmadd*, STTNI, (v)pclmulqdq, (v)psadw, (v)pcmpgtq, vpsllvd, (v)bendv*, (v)plendw,
FP Add	1	(v)addp*, (v)cmpp*, (v)max*, (v)min*,
FP Mov	1	(v)movap*, (v)movup*, (v)movsd/ss, (v)movd gpr, (v)andp*, (v)orp*
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

**注意:**

1. Broadwell<sup>†</sup> マイクロアーキテクチャー・ベースと CPUID ADX 機能フラグをサポートするプロセッサのみで利用可能。

リザベーション・ステーション (RS) が 60 エントリーに拡大され (インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> では 54 エントリー)、マイクロオペレーション (μop) の実行準備ができている場合、サイクルごとに最大 8 つのマイクロオペレーション (μop) をディスパッチできます。RS でマイクロオペレーション (μop) は特定のデータ型やデータの粒度を処理するスタックに分けられ、発行ポートから特定の実行クラスターにディスパッチされます。

あるスタックで実行されるマイクロオペレーション (μop) のソースが、別のスタックで実行されるマイクロオペレーション (μop) から取得される場合、遅延が生じる可能性があります。インテル® SSE 整数操作とインテル® SSE 浮動小数点操作の間の遷移でも遅延が発生します。これは、命令フローに追加されるマイクロオペレーション (μop) によって、データ遷移が行われるためです。実行後にライトバックされるデータを、後続のマイクロオペレーション (μop) 実行にバイパスする方法とその遅延サイクル数を表 2-30 に示します。



表 2-9 マイクロオペレーション (uop) 間のパイプスによる遅延 (サイクル数)

遷移元/遷移先	整数	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
整数		<ul style="list-style-type: none"> <li>• uOP (ポート 5)</li> <li>• uOP (ポート 6) + 1 サイクル遅延</li> </ul>	<ul style="list-style-type: none"> <li>• uOP (ポート 5)</li> <li>• uOP (ポート 6) + 1 サイクル遅延</li> </ul>	uOP (ポート 5) + 3 サイクル遅延
SSE-INT/ AVX-INT	uOP (ポート 1)		1 サイクル遅延	
SSE-FP/ AVX-FP_LOW	uOP (ポート 1)	1 サイクル遅延		uOP (ポート 5) + 1 サイクル遅延
X87/ AVX-FP_High	uOP (ポート 1) + 3 サイクル遅延		uOP (ポート 5) + 1 サイクル遅延	
ロード		1 サイクル遅延	1 サイクル遅延	2 サイクル遅延

### 2.3.4 キャッシュとメモリー・サブシステム

キャッシュ階層は前世代と類似しており、各コアに L1 命令キャッシュ、L1 データキャッシュ、L2 ユニファイド・キャッシュがあります。さらに、L3 ユニファイド・キャッシュもあり、そのサイズは製品構成に依存します。L3 キャッシュは複数のキャッシュスライスで構成されており、各スライスのサイズはリング・インターコネクで接続される製品構成に依存します。キャッシュトポロジーの詳細は、CPUID leaf 4 で確認できます。L3 キャッシュは、すべてのプロセッサ・コアで共有される “アンコア” サブシステムにあります。一部の製品構成では L4 キャッシュもサポートされています。表 2-28 にキャッシュ階層の詳細を示します。

表 2-10 インテル® マイクロアーキテクチャー Haswell<sup>†</sup> のキャッシュ・パラメーター

レベル	容量/アソシアティブ (ウェイ)	ラインサイズ (バイト)	最小レイテンシー <sup>1</sup>	スループット (クロック数)	ピーク帯域幅 (バイト/サイクル数)	アップデート方式
L1 データ	32KB/8	64	4 サイクル	0.5 <sup>2</sup>	64 (ロード) + 32 (ストア)	ライトバック
命令	32KB/8	64	なし	なし	なし	なし
L2	256KB/8	64	11 サイクル	それぞれ異なる	64	ライトバック
L3 (共有)	それぞれ異なる	64	~ 34	それぞれ異なる		ライトバック

**注意:**

1. ソフトウェアから検知できるレイテンシーは、アクセスパターンやその他の要因により異なります。
2. L1 データキャッシュは、最大 32 バイトのデータをフェッチ可能なロード操作を各サイクルで 2 つ処理できません。

TLB (Translation Lookaside Buffer) 階層は、L1 命令キャッシュ用の TLB、L1 データキャッシュ用の TLB、L2 ユニファイド・キャッシュ用の TLB で構成されます。

表 2-11 インテル® マイクロアーキテクチャー Haswell<sup>†</sup> の TLB パラメーター

レベル	ページサイズ	エントリー	アソシアティブ (ウェイ)	パーティション
命令	4KB	128	4 ウェイ	動的
命令	2MB/4MB	スレッドあたり 8		固定
L1 データ	4KB	64	4	固定
L1 データ	2MB/4MB	32	4	固定
L1 データ	1GB	4	4	固定
L2	4KB、2MB/4MB ページで共有	1024	8	固定

### 2.3.4.1 ロード操作とストア操作の拡張

L1 データキャッシュは各サイクルで 2 つの 256 ビット・ロード操作と 1 つの 256 ビット・ストア操作を処理でき、L2 ユニファイド・キャッシュは各サイクルで 1 つのキャッシュライン (64 バイト) を処理できます。さらに、マイクロオペレーション (μop) の同時実行をサポートするため、72 のロードバッファと 42 のストアバッファが装備されています。

### 2.3.5 Haswell-E<sup>†</sup> マイクロアーキテクチャー

Haswell-E<sup>†</sup> マイクロアーキテクチャーは、Haswell<sup>†</sup> マイクロアーキテクチャーで説明した同じプロセッサ・コアを包括するインテル® プロセッサをベースとしていますが、より高度なアンコアと統合 I/O 能力を提供します。Haswell-E<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサは、複数ソケットのプラットフォームをサポートします。

Haswell-E<sup>†</sup> マイクロアーキテクチャーは、スケーラビリティと高いパフォーマンス向けに多面的なプロセッサ・アーキテクチャーをサポートします。Haswell-E<sup>†</sup> マイクロアーキテクチャーのアンコアと統合 I/O サブシステムで提供される機能を以下に示します。

- Y 複数ソケット構成で複数のインテル® QuickPath インターコネクトをサポートします。
- Y 物理コアごとに最大 2 つのメモリー・コントローラーが統合されました。
- Y 物理プロセッサごとに最大 40 レーンの PCI Express\* 3.0 リンクを提供します。
- Y それぞれの物理プロセッサで、最大 18 個のプロセッサ・コアが 2 つのリング・インターコネクトによって L3 接続されます。

図 2-6 に、Haswell-E<sup>†</sup> マイクロアーキテクチャーを使用する 12 コアのプロセッサ実装の例を示します。アンコアと統合 I/O サブシステムの能力は、Haswell-E<sup>†</sup> マイクロアーキテクチャーを実装するプロセッサ・ファミリーごとに異なります。詳細については、インテル® Xeon® プロセッサ E5 v3 ファミリーのデータシートをご覧ください。

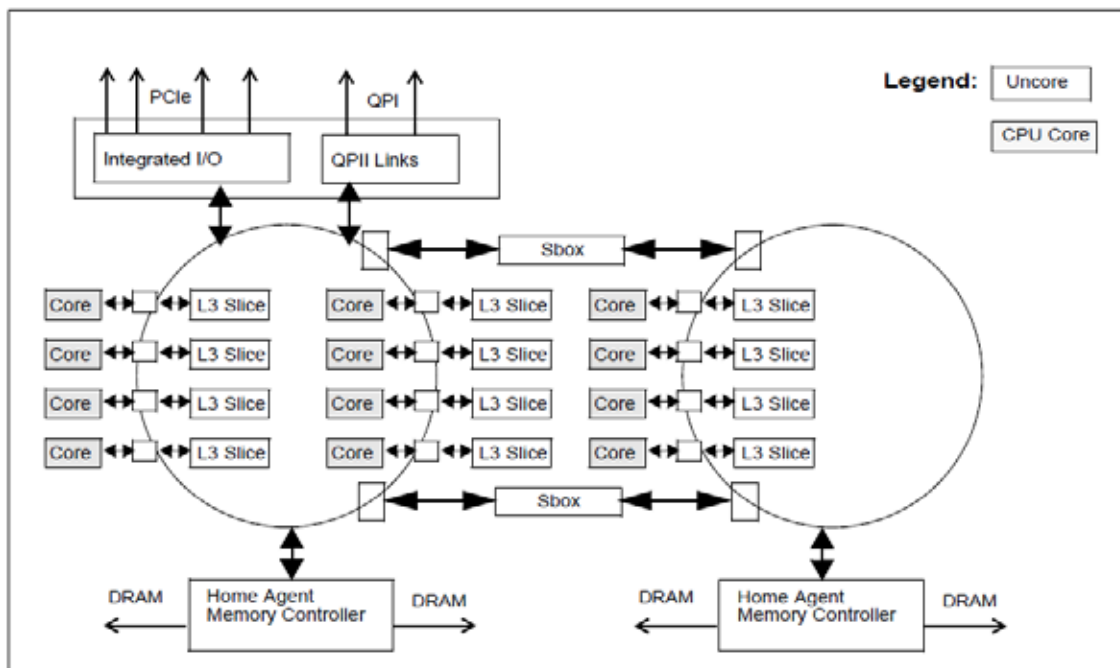


図 2-6 プロセッサ・コアをサポートする Haswell-E+ マイクロアーキテクチャーの例

### 2.3.6 Broadwell+ マイクロアーキテクチャー

インテル® Core™ M プロセッサは、Broadwell+ マイクロアーキテクチャーをベースにしています。Broadwell+ マイクロアーキテクチャーは、Haswell+ マイクロアーキテクチャーから派生し、いくつかの拡張を実装しています。このセクションでは、Broadwell+ マイクロアーキテクチャーの拡張機能について説明します。

- Y 浮動小数点乗算命令のレイテンシーが、前の世代の 5 サイクルから Broadwell+ マイクロアーキテクチャーでは 3 サイクルに改善されています。これは、インテル® AVX、インテル® SSE、および FP 命令セットに適用されます。
- Y ギャザー命令のスループットが大幅に向上しました。表 C-5 をご覧ください。
- Y Broadwell+ マイクロアーキテクチャーでは、PCLMULQDQ 命令が単一  $\mu\text{op}$  で実装されており、レイテンシーとスループットが改善されました。

TLB の階層は、命令キャッシュ向けの TLB、L1D 向けの TLB、さらに L2 向けのユニファイド TLB から成ります。

表 2-12 Broadwell+ マイクロアーキテクチャーの TLB パラメーター

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2MB pages	1536	6	fixed
Second Level	1GB pages	16	4	fixed

## 2.4 インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup>

インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> は、インテル® Core™ マイクロアーキテクチャーおよびインテル® マイクロアーキテクチャー Nehalem<sup>†</sup> の成功を受けて開発されました。以下の革新的な機能を提供しています。

- Y インテル® アドバンスド・ベクトル・エクステンション (インテル® AVX)
  - 128 ビットのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の 256 ビット浮動小数点命令セットへの拡張により、128 ビットのコードと比べパフォーマンスが最大 2 倍向上
  - 非破壊デスティネーションの採用により、より柔軟なコーディング手法を実現
  - 256 ビットのインテル® AVX コード、128 ビットのインテル® AVX コード、128 ビットのインテル® SSE レガシーコード間の柔軟な移行および共存をサポート
- Y 機能が強化されたフロントエンドおよび実行エンジン
  - 新たなデコード済み命令キャッシュの実装により、フロントエンドの帯域幅を向上および分岐の予測ミスのペナルティーを軽減
  - 高度な分岐予測
  - マクロフュージョンの追加サポート
  - ダイナミック・エグゼキューション範囲の拡大
  - マルチ精度整数算術の拡張 (ADC/SBB, MUL/IMUL)
  - LEA 帯域幅の向上
  - 一般的な実行ストール (読み出しポート、ライトバック競合、バイパス・レイテンシー、パーシャルストール) の削減
  - 高速な浮動小数点例外処理
  - XSAVE/XRSTORE 命令のパフォーマンスの向上、および新しい XSAVEOPT 命令の追加
- Y キャッシュ階層の改善によるデータパスの拡大
  - メモリー操作のための 2 つのシンメトリックなポートにより、帯域幅が倍増
  - バッファ増加により、より多くの実行中のロードおよびストアを同時に操作
  - 各サイクルで 2 つのロードと 1 つのストアを実行可能な内部帯域幅
  - プリフェッチの改善
  - 高帯域幅および低レイテンシーの LLC アーキテクチャー
  - オンダイ・インターコネクトの高帯域幅リング・アーキテクチャー
- Y システムオンチップのサポート
  - 第 2 世代インテル® Core™ プロセッサでグラフィックス・エンジンとメディアエンジンの統合
  - PCIE コントローラーの統合
  - メモリー・コントローラーの統合
- Y インテル® ターボ・ブースト・テクノロジー 2.0
  - TDP ヘッドルームの改善により、CPU コアおよび内蔵グラフィックス・ユニットのパフォーマンスを向上

### 2.4.1 インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> のパイプライン概要

図 2-7 に、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> をベースとするプロセッサ・コアのパイプラインおよび主要構成要素を示します。パイプラインは以下で構成されています。

- Y 命令をフェッチし、マイクロオペレーション (μop) にデコードするインオーダー発行フロントエンド。フロントエンドは、プログラムが実行する可能性の最も高いパスからマイクロオペレーション (μop) の連続的なストリームを次のパイプライン・ステージに供給します。
- Y サイクルあたり最大 6 つのマイクロオペレーション (μop) をディスパッチするアウトオブオーダー・スーパースケラー実行エンジン。入力ソースが準備でき、実行リソースが利用可能になり次第実行できるように、割り当て/リネームブロックがマイクロオペレーション (μop) を「データフロー」の順序に並べ替えます。

Y 検出された例外など、マイクロオペレーション (μop) の実行結果が元のプログラム順序になることを確実にするインオーダー・リタイアメント・ユニット。

パイプライン中の命令の流れは以下に要約できます。

1. 分岐予測ユニットは、プログラムから次に実行するコードブロックを選択します。プロセッサは、次のリソース内で (順番どおりに) コードを検索します。
  - a. デコード済み命令キャッシュ
  - b. レガシーのデコード・パイプラインによってデコードされた命令キャッシュ
  - c. 必要に応じて、L2 キャッシュ、ラスト・レベル・キャッシュ (LLC) およびメモリー

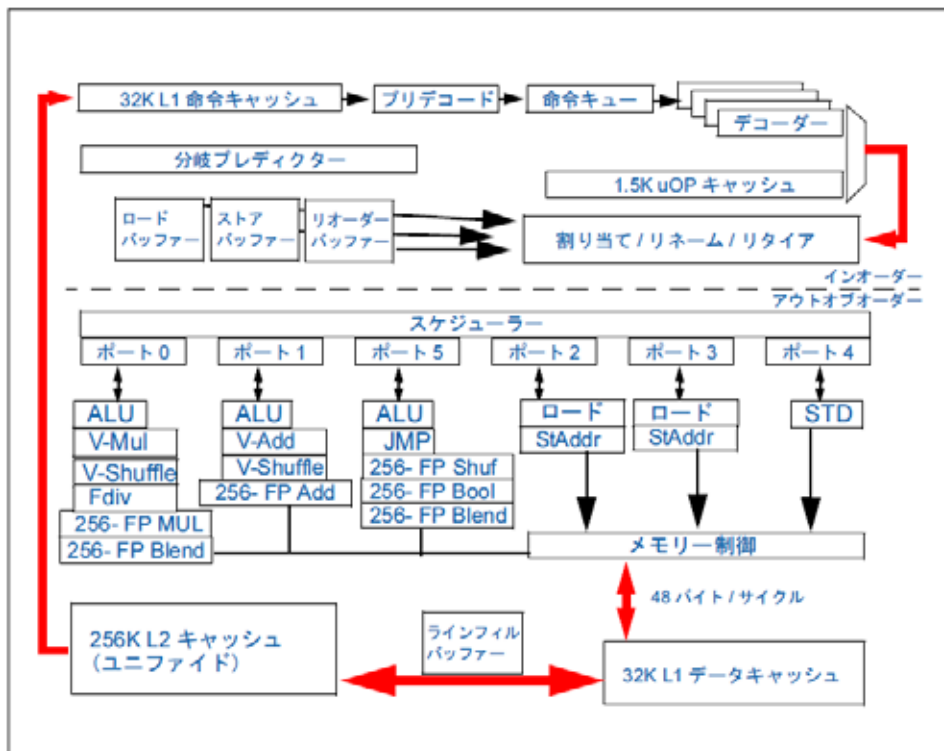


図 2-7 インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> のパイプラインの構造

2. コードに対応するマイクロオペレーション (μop) がリネーム/リタイアメント・ブロックに送信されます。プログラムの順番どおりにスケジューラーに入りますが、データフロー順序に従ってスケジューラーから実行そして割り当て解除されます。同時に準備できたマイクロオペレーション (μop) は、一般に FIFO 順序で操作されます。3つのスタックに配置されている実行リソースを使用して、マイクロオペレーション (μop) が実行されます。各スタックの実行ユニットは、命令のデータ型に関連付けられます。

分岐の予測ミスは分岐実行時に通知されます。正しいパスからマイクロオペレーション (μop) を供給するようにフロントエンドに指示されます。プロセッサは、分岐の予測ミスよりも前の処理と正しいパスからの処理をオーバーラップできます。

3. 並列性および最高のパフォーマンスを実現するため、メモリー操作が管理およびリオーダーされます。L1 データキャッシュにミスがあった場合、L2 キャッシュを参照します。データキャッシュはノンブロッキングで、複数のミスを同時に処理できます。
4. 例外 (フォルト、トラップ) は、フォルトが発生した命令のリタイアメント (または、リタイアメントの試行) 時に通知されます。

インテル® ハイパースレッディング・テクノロジーが有効であれば、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> をベースとする各プロセッサ・コアは、2つの論理プロセッサをサポートできます。

## 2.4.2 フロントエンド

この節では、フロントエンドの主な特性について説明します。表 2-13 に、フロントエンドの構成要素、その機能、扱う問題の一覧を示します。

表 2-13 インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> のフロントエンドの構成要素

構成要素	機能	パフォーマンスの課題
命令キャッシュ	命令バイトの 32K バイトのバッキングストア	ホットな命令バイトへの高速アクセス
レガシー・デコード・パイプライン	マイクロオペレーション (μOP) キューおよびデコード済み命令キャッシュに供給された命令をマイクロオペレーション (μOP) にデコードする。	レガシーのインテル® プロセッサと同じデコード・レイテンシーおよび帯域幅を提供する。 デコード済み命令キャッシュのウォームアップ
デコード済み命令キャッシュ	マイクロオペレーション (μOP) キューにマイクロオペレーション (μOP) のストリームを供給する。	レガシー・デコード・パイプラインよりも低いレイテンシーおよび低い消費電力でより高いマイクロオペレーション (μOP) 帯域幅を提供する。
MSROM	レガシー・デコード・パイプラインとデコード済み命令キャッシュの両方からアクセス可能な、複雑な命令のマイクロオペレーション (μOP) のセットを保持	
分岐予測ユニット (BPU)	次に実行されるコードブロックを判断し、デコード済み命令キャッシュとレガシー・デコード・パイプラインの参照を推進する。	分岐の予測ミスを軽減することで、パフォーマンスおよび電力効率を向上する。
マイクロオペレーション (μOP) キュー	デコード済み命令キャッシュとレガシー・デコード・パイプラインからマイクロオペレーション (μOP) をキューに格納する。	フロントエンド・バブルの隠蔽。一定の速度で実行マイクロオペレーション (μOP) を提供する。

### 2.4.2.1 レガシー・デコード・パイプライン

レガシー・デコード・パイプラインは、命令トランслーション・ルックアサイド・バッファ (ITLB)、命令キャッシュ (ICache)、命令プリデコーダー、命令デコードユニットで構成されます。

#### 命令キャッシュと ITLB

命令フェッチは、ITLB を通じて命令キャッシュに対して 16 バイト境界で参照します。命令キャッシュは、命令プリデコーダーにサイクルごとに 16 バイトを供給できます。表 2-12 に、命令キャッシュと ITLB の前世代との比較を示します。

表 2-14 インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> の命令キャッシュと ITLB

構成要素	インテル® マイクロアーキテクチャー Sandy Bridge	インテル® マイクロアーキテクチャー Nehalem
命令キャッシュのサイズ	32K バイト	32K バイト
命令キャッシュの連想性 (ウェイ)	8	4
ITLB 4K ページエントリー数	128	128
ITLB ラージページ (2M または 4M) エントリー数	8	7

ITLB でミスが生じた場合、DTLB と ITLB に共通の第 2 レベルの TLB (STLB) をルックアップします。ITLB ミスおよび STLB ヒットのペナルティは 7 サイクルです。

## 命令プリデコーダー

プリデコード・ユニットは、命令キャッシュから 16 バイトを受け入れて、命令長を判断します。

以下のレングス変更プリフィクス (LCP) は、デフォルトの命令長と異なる命令長を意味します。そのため、命令長のデコード時に LCP あたり 3 サイクルの追加ペナルティーが発生します。以前のプロセッサでは、1 つまたは複数の LCP を含む各 16 バイト・チャンクに対して 6 サイクル・ペナルティーが発生します。通常、16 バイト・チャンクに含まれる LCP は 1 つだけであるため、ほとんどの場合、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> では、以前のプロセッサと比べペナルティーは改善されています。

- Y オペランド・サイズ・オーバーライド (66H): ワード/ダブルワード即値データを持つ命令の先頭に付きます。このプリフィクスは、コードが 32 ビットより小さい 16 ビット・データ型、Unicode 処理、画像処理を使用している場合に出現します。
- Y アドレス・サイズ・オーバーライド (67H): リアルモード、ビッグリアルモード、16 ビット保護モード、または 32 ビット保護モードで、mod r/m を持つ命令の先頭に付きます。このプリフィクスは、ブート・コード・シーケンスに出現します。
- Y インテル® 64 命令セットの REX プリフィクス (4xh) は、2 つの命令 (MOV オフセットと MOV 即値) のサイズを変更できます。ただし、LCP のペナルティーは発生しないので、LCP とはみなされません。

## 命令デコード

命令をマイクロオペレーション (μop) にデコードする 4 つのデコードユニットで構成されます。最初のデコーダーは、最大で 4 つのマイクロオペレーション (μop) で構成されるすべての IA-32 命令とインテル® 64 命令をデコードします。残りの 3 つのデコードユニットは、単一マイクロオペレーション (μop) 命令を処理します。4 つすべてのデコードユニットは、マイクロフュージョンやマクロフュージョンなど、一般的なケースにおける単一マイクロオペレーション (μop) のフローをサポートしています。

デコーダーから発行されたマイクロオペレーション (μop) は、マイクロオペレーション (μop) キューとデコード済み命令キャッシュに送られます。4 マイクロオペレーション (μop) よりも長い命令は、MSROM からマイクロオペレーション (μop) が生成されます。MSROM からの帯域幅は 1 サイクルあたり 4 マイクロオペレーション (μop) です。MSROM からマイクロオペレーション (μop) が生成される命令は、レガシー・デコード・パイプラインまたはデコード済み命令キャッシュから開始できます。

## マイクロフュージョン

マイクロフュージョンでは、同一命令の複数のマイクロオペレーション (μop) が単一の複雑なマイクロオペレーション (μop) に融合されます。複雑なマイクロオペレーション (μop) は、マイクロフュージョンされなかった場合と同じ回数、アウトオブオーダー実行コアにディスパッチされます。

マイクロフュージョンを利用すると、デコード帯域幅を損ねることなく、複合命令セット・コンピューター (CISC) 命令セットを使いメモリーとレジスター間の操作を行い、実際のプログラム動作を表現できます。マイクロフュージョンでは、デコードからリタイアメントに供給される命令帯域幅が向上し、消費電力が減少します。

単一マイクロオペレーション (μop) 命令を使用して命令シーケンスをコーディングすると、コードサイズが大きくなり、レガシー・パイプラインからのフェッチ帯域幅を圧迫します。

すべてのデコーダーで処理が可能なマイクロフュージョンされたマイクロオペレーション (μop) の例を以下に示します。

- Y 即値のストアを含めた、メモリーに対するすべてのストア。ストアは、内部でストア・アドレスとストア・データの 2 つの異なる機能として実行されます。

Y ロード操作と演算操作を組み合わせたすべての命令 (ロード + op),

例:

- ADDPS XMM9, QWORD PTR [RSP+40]
- FADD DOUBLE PTR [RDI+RSI\*8]
- XOR RAX, QWORD PTR [RBP+32]

Y 「ロードおよびジャンプ」形式のすべての命令。

例:

- JMP [RDI+200]
- RET

Y 即値とメモリーオペランドを持つ CMP および TEST 命令。

RIP 相対アドレス指定を行う命令は、以下の場合にはマイクロフュージョンの対象になりません。

Y 即値の追加が必要な場合。

例:

- CMP [RIP+400], 27
- MOV [RIP+3000], 142

Y 命令が、RIP 相対アドレス指定を使用して間接ターゲットが指定された制御フロー命令である場合。

例:

- JMP [RIP+5000000]

このような場合、マイクロフュージョンできない命令により、デコーダー 0 は 2 つのマイクロオペレーション ( $\mu\text{op}$ ) を発行する必要があり、デコード帯域幅の損失を招きます。

64 ビット・コードでは、グローバルデータに RIP 相対アドレス指定を使用するのが一般的です。このようなケースではマイクロフュージョンできないため、32 ビット・コードを 64 ビット・コードに移植する際にパフォーマンスが低下することがあります。

## マクロフュージョン

マクロフュージョンでは、2 つの命令が単一のマイクロオペレーション ( $\mu\text{op}$ ) にマージされます。インテル® Core™ マイクロアーキテクチャーでは、このハードウェアによる最適化は、マクロフュージョン可能な命令ペアの組み合わせ (1 番目と 2 番目) に特定の条件があります。

Y マクロフュージョンされるペアの 1 番目の命令はフラグを変更します。以下の命令はマクロフュージョン可能です。

- インテル® マイクロアーキテクチャー Nehalem<sup>†</sup>: CMP, TEST
- インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup>: CMP, TEST, ADD, SUB, AND, INC, DEC
- 以下の場合に、これらの命令をマクロフュージョンできます。
  - z 1 番目のソース/ デスティネーション・オペランドがレジスターである。
  - z 2 番目のソースオペランド (存在する場合) が即値、レジスター、非 RIP 相対メモリーのいずれかである。

Y マクロフュージョン可能なペアの 2 番目の命令は条件分岐です。表 3-1 にマクロフュージョン可能な分岐との組み合わせを示します。

1 番目の命令がキャッシュラインのバイト 63 で終了し、2 番目の命令が次のキャッシュラインのバイト 0 で始まる条件分岐である場合、マクロフュージョンは行われません。

このような命令ペアは多くのアプリケーションに含まれる可能性があるため、再コンパイルされない既存のバイナリーでも、マクロフュージョンによってパフォーマンスが向上します。



マクロフュージョンされたそれぞれの命令は、単一のディスパッチで実行されます。これによって、レイテンシーが減少し、実行リソースが解放されます。また、リネーム/リタイア帯域幅の向上、仮想ストレージの拡大、少ないビット数で処理量を増加することによる省電力も実現できます。

## 2.4.2.2 デコード済み命令キャッシュ

デコード済み命令キャッシュは、基本的にはレガシー・デコード・パイプラインのアクセラレーターです。デコード済み命令キャッシュでは、デコードされた命令を格納することにより、以下の機能が可能になります。

- Y 分岐予測ミスでのレイテンシーの減少
- Y アウトオブオーダー・エンジンへのマイクロオペレーション (μop) 供給帯域幅の拡大
- Y フロントエンドの消費電力の軽減

デコード済み命令キャッシュは、命令デコーダーの出力をキャッシュに蓄えます。実行のためにマイクロオペレーション (μop) が次に要求されると、デコード済み命令キャッシュからデコードされたマイクロオペレーション (μop) が取り出されます。これにより、マイクロオペレーション (μop) のフェッチステージとデコードステージをスキップでき、フロントエンドの消費電力およびレイテンシーが軽減されます。デコード済み命令キャッシュはマイクロオペレーション (μop) の 80% 以上の平均ヒット率を達成し、さらに「Hotspot」のヒット率は通常 100% 近くになります。

一般的な整数プログラムの平均命令長は、1 命令あたり 4 バイト未満で、フロントエンドがバックエンドに先立って、スケジューラーが命令レベルの並列処理を見つけやすいように広い範囲を満たします。ただし、インテル® SSE メディア・アルゴリズムや過度にアンロールされたループなど、基本ブロックが多くの命令で構成されるパフォーマンスが高いコードの場合は、1 サイクルあたり 16 命令バイトが制約になることがあります。デコード済み命令キャッシュの 32 バイト指向はそのようなコードに役立ち、この制約が回避されます。

デコード済み命令キャッシュは、時間的局所性と空間的局所性でプログラムのパフォーマンスを自動的に向上させます。ただし、デコード済み命令キャッシュの潜在的な能力を十分に利用するには、内部構成を理解する必要があります。デコード済み命令キャッシュは 32 のセットで構成されます。各セットには 8 つのウェイが含まれます。各ウェイは最大 6 つのマイクロオペレーション (μop) を保持できます。デコード済み命令キャッシュは最大 1536 のマイクロオペレーション (μop) を保持できます。

以下は、デコード済み命令キャッシュにマイクロオペレーション (μop) が格納される際のルールの一部です。

- Y ウェイのすべてのマイクロオペレーション (μop) は、コード内で静的に隣接し、同じアライメントされた 32 バイト領域内の EIP を持ちます。
- Y 同じ 32 バイトにアライメントされたチャンクに最大 3 つのウェイを割り当てることができるため、オリジナルの IA プログラムの 32 バイト領域あたり、合計 18 のマイクロオペレーション (μop) をキャッシュに格納できます。
- Y 複数のマイクロオペレーション (μop) をウェイに分割することはできません。
- Y 1 ウェイあたり、最大 2 つの分岐が許容されます。
- Y MSROM を使用する命令があると、ウェイ全体が消費されます。
- Y 条件分岐以外の分岐は、ウェイの最後のマイクロオペレーション (μop) です。
- Y マクロフュージョンされたマイクロオペレーション (μop) (ロード + op およびストア) は 1 つのマイクロオペレーション (μop) として保持されます。
- Y マクロフュージョンされた命令のペアは 1 つのマイクロオペレーション (μop) として保持されます。
- Y 64 ビット即値を伴う命令では、即値を保持するのに 2 つのスロットを必要とします。

こうした制約によりマイクロオペレーション (μop) をデコード済み命令キャッシュに格納できない場合、レガシー・デコード・パイプラインから供給されます。レガシー・パイプラインからマイクロオペレーション (μop) が供給された場合、デコード済み命令キャッシュからのマイクロオペレーション (μop) のフェッチは次の分岐のマイクロオペレーション (μop) までは再開できません。頻繁に切り替えると、ペナルティーが発生する可能性があります。

デコード済み命令キャッシュは、事実上、命令キャッシュと ITLB に含まれます。つまり、デコード済み命令キャッシュにマイクロオペレーション (μop) がある命令は、オリジナルの命令バイトが命令キャッシュに存在します。命令キャッシュの排出はデコード済み命令キャッシュからも排出される必要があり、この場合、必要なラインだけが排出されます。

デコード済み命令キャッシュ全体がフラッシュされることがあります。この 1 つの理由として ITLB エントリーの排出が考えられます。それ以外の理由は通常アプリケーション・プログラマーからは見えず、例えば CR3 でのマッピングや CR0 および CR4 での機能やモードの有効化など、重要な制御が変更になったときに発生します。また、例えば CS ベースアドレスがゼロに設定されていないなど、デコード済み命令キャッシュが無効になっているケースもあります。

### 2.4.2.3 分岐予測

分岐予測は分岐ターゲットを予測し、実際に分岐の実行パスが判明する手前から命令の実行を開始できます。すべての分岐が予測に分岐予測ユニット (BPU) を利用します。このユニットは、分岐の EIP だけでなく、この EIP に実行が到達した実行パスに基づいて、ターゲットアドレスを予測します。BPU は以下の分岐タイプを効率良く予測できます。

- Y 条件分岐
- Y 直接コールおよびジャンプ
- Y 間接コールおよびジャンプ
- Y リターン

### 2.4.2.4 マイクロオペレーション(μop)キューおよびループストリーム検出器(LSD)

マイクロオペレーション (μop) キューはフロントエンドとアウトオブオーダー・エンジンを分離します。図 2-7 に示すように、マイクロオペレーション (μop) の生成とリネーマーの間にあります。このキューは、フロントエンドのマイクロオペレーション (μop) の各種のソースで発生するバブルを隠蔽するのに有効であり、各サイクルで実行に向けて 4 つのマイクロオペレーション (μop) を確実に供給します。

マイクロオペレーション (μop) キューは、特定の命令タイプに対してポストデコード機能を提供します。特に、演算処理とすべてのストアを組み合わせたロードでは、インデックス付きアドレス指定が使用されると、デコーダーやデコード済み命令キャッシュでは単一のマイクロオペレーション (μop) として示されます。マイクロオペレーション (μop) キューでは、これはアンラミネーションと呼ばれる過程で 2 つのマイクロオペレーション (μop) (1 つはロード、もう一方は演算) に細分化されます。一般的な例は、以下の「ロード + 演算」命令です。

```
ADD RAX, [RBP+RSI] ; rax := rax + LD( RBP+RSI )
```

同様に、以下のストア命令にはレジスターソースが 3 つあり、「ストアアドレス生成」と「ストアデータ生成」のサブコンポーネントに分割されます。

```
MOV [ESP+ECX*4+12345678], AL
```

アンラミネーションによって生成される追加のマイクロオペレーション (μop) は、リネーム帯域幅とリタイアメント帯域幅を消費します。ただし、これによって全体的な消費電力を削減できるという利点があります。インデックス付きアドレス指定 (一般に配列処理の場合に生じる) に影響されるコードでは、ベース (またはベース + ディスプレースメント) アドレス指定を使用するリコード・アルゴリズムで、ロード + 演算命令とストア命令をフュージョンさせることにより、パフォーマンスを向上させることができます。

#### ループストリーム検出器(LSD)

ループストリーム検出器は、インテル® Core™ マイクロアーキテクチャーで導入されました。LSD は、マイクロオペレーション (μop) キューにある小さなループを検出し、ロックします。分岐予測ミスで終了するまで、どのキャッシュ

からもマイクロオペレーション (μop) のフェッチ、デコード、または読み取りがない状態となり、マイクロオペレーション (μop) キューからループストリームが送出されます。

以下の属性のループが LSD/マイクロオペレーション (μop) キューの再実行の対象となります。

- Y 32 命令バイトの最大 8 つのチャンクフェッチ
- Y 最大 28 のマイクロオペレーション (μop) (28 以下の命令)
- Y すべてのマイクロオペレーション (μop) がデコード済み命令キャッシュにも存在する
- Y 実行される分岐が 8 つを超えてはならず、いずれも CALL や RET ではない場合
- Y 不一致のスタック操作は許可されません。例えば、POP 命令よりも PUSH 命令が多い場合などです

大量の計算を行うループ、検索、文字列の移動は、上記の特性に当てはまることが多く、状況に応じてループキャッシュ機能を使用します。パフォーマンスが高いコードを作成するには、LSD 機能をオーバーフローしても、通常はループアンロールを行う方がパフォーマンス的に望ましいでしょう。

## 2.4.3 アウトオブオーダー・エンジン

アウトオブオーダー・エンジンは、消費電力特性に優れ、以前の世代と比べ、パフォーマンスが向上しています。依存関係の連鎖を検出し、正しいデータフローを保持したまま、アウトオブオーダーで実行に送出します。依存関係の連鎖が 2 次データ・キャッシュラインなどのリソースを待機している場合、別の連鎖からマイクロオペレーション (μop) を実行コアに送ります。このため、1 サイクルあたりに実行される命令数 (IPC) の全体的なレートが増加します。

アウトオブオーダー・エンジンは、図 2-7 のコア機能図に示すように、リネーム/リタイアメント・ブロックとスケジューラーの 2 つのブロックから構成されます。

アウトオブオーダー・エンジンには、以下の主要構成要素が含まれます。

**リネーマー:** マイクロオペレーション (μop) をフロントエンドから実行コアに移動します。マイクロオペレーション (μop) 間の不正な依存関係を排除し、マイクロオペレーション (μop) のアウトオブオーダー実行を可能にします。

**スケジューラー:** すべてのソースオペランドの準備が整うまで、マイクロオペレーション (μop) をキューに格納します。できる限り先入れ先出し (FIFO) 順序に従い、準備のできたマイクロオペレーション (μop) をスケジュールし、利用可能な実行ユニットにディスパッチします。

**リタイアメント:** 命令およびマイクロオペレーション (μop) を順番にリタイアさせ、フォルトおよび例外を処理します。

### 2.4.3.1 リネーマー

リネーマーは、図 2-5 のインオーダー部分とスケジューラーのデータフローの橋渡しを行います。サイクルごとに最大 4 つのマイクロオペレーション (μop) をマイクロオペレーション (μop) キューからアウトオブオーダー・エンジンに移動します。リネーマーはサイクルごとに最大 4 つのマイクロオペレーション (μop) (マイクロおよびマクロフュージョンされていない状態、マイクロフュージョンされた状態、またはマクロフュージョンされた状態) を送出できますが、これは発行ポートがサイクルごとに 6 つのマイクロオペレーション (μop) をディスパッチするのに相当します。この過程では、アウトオブオーダー・コアは以下のステップを実行します。

- Y マイクロオペレーション (μop) のソースとデスティネーションを、マイクロアーキテクチャー上のソースとデスティネーションにリネームします。
- Y マイクロオペレーション (μop) にリソースを割り当てます。例えば、ロードバッファやストアバッファなど。
- Y マイクロオペレーション (μop) を適切なディスパッチ・ポートにバインドします。

マイクロオペレーション (μop) によってはリネーム中に実行が完了するものがあり、その場合、実行が完了した時点でパイプラインから削除され、実行帯域幅に影響しません。例えば、以下のようなものがあります。

- ÿ ゼロイディオム (依存関係解消イディオム)
- ÿ NOP
- ÿ VZEROUPPER
- ÿ FXCHG

以前のマイクロアーキテクチャーでは各サイクルで 1 つの分岐しか割り当てることができなかったのに対し、リネーマーは各サイクルで 2 つの分岐を割り当てることができます。これによって、実行時の一部のバブルを解消できます。

インデックス・レジスターを使用する、マイクロフュージョンされたロード操作とストア操作は 2 つのマイクロオペレーション (μop) に分解され、リネーマーが各サイクルで使用可能な 4 つのスロットのうち 2 つが消費されます。

### ゼロイディオム(依存関係解消イディオム)

通常の命令を使用してレジスターの内容をゼロにクリアすることにより、命令の並列性を向上できます。リネーマーは、デスティネーション・レジスターのゼロ評価時にこれを検出します。

可能な場合は以下のいずれか 1 つの依存関係解消イディオムを使用して、レジスターをクリアします。

- ÿ XOR REG,REG
- ÿ SUB REG,REG
- ÿ PXOR/VPXOR XMMREG,XMMREG
- ÿ PSUBB/W/D/Q XMMREG,XMMREG
- ÿ VPSUBB/W/D/Q XMMREG,XMMREG
- ÿ XORPS/PD XMMREG,XMMREG
- ÿ VXORPS/PD YMMREG, YMMREG

ゼロイディオムはリネーマーによって検出および解決されるため、実行レイテンシーはありません。

もう 1 つ別に「1 イディオム」という依存関係解消イディオムがあります。

- ÿ CMPEQ XMM1, XMM1; "1 イディオム" はすべての要素をすべて「1」に設定します

この場合、マイクロオペレーション (μop) の実行が必要ですが、入力データに関係なく、出力データは常に「すべて 1」であることがわかっているため、ゼロイディオムの場合と同様、マイクロオペレーション (μop) のソースへの依存関係は存在せず、空いている実行ポートが見つかり次第実行できます。

## 2.4.3.2 スケジューラー

スケジューラーは、実行ポートへのマイクロオペレーション (μop) のディスパッチを制御します。そのため、どのマイクロオペレーション (μop) が準備でき、そのソースが何であるか (レジスター・ファイル・エントリーなのか、それとも実行ユニットから直接のバイパスなのか) を特定する必要があります。ディスパッチ・ポートおよびライトバック・バスの利用状況、準備ができたマイクロオペレーション (μop) の優先度に応じて、スケジューラーはサイクルごとにどのマイクロオペレーション (μop) をディスパッチするのかが選択します。

## 2.4.4 実行コア

実行コアはスーパースケイラーであり、命令をアウトオブオーダーで処理できます。潜在的な遅延を抑えながら、最も一般的な操作を効率良く処理することで、全体的なパフォーマンスを最適化します。

アウトオブオーダー実行コアでは、以前の世代と比べ、以下の点で、実行ユニットの編成が改善されています。

- ÿ 読み出しポートのストールの軽減

- Y ライトバックの競合および遅延の軽減
- Y 消費電力の軽減
- Y デノーマル入力とアンダーフロー出力処理の SIMD FP アシストの軽減

FTZ=0 および DAZ=0 における操作では、ある種の高精度 FP アルゴリズムが必要です。つまり、SIMD FP アシストによる以前の世代のマイクロアーキテクチャーのパフォーマンスの低下を犠牲にして、より高い数値精度を実現するために、アンダーフローの即値結果とデノーマル入力を許容します。インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> では、次のインテル® SSE 命令（および派生するインテル® AVX）における SIMD FP アシストを削減します: ADDPD/ADDPS、MULPD/MULPS、DIVPD/DIVPS、および CVTPD2PS。

アウトオブオーダー・コアは 3 つの実行スタックから構成され、それぞれのスタックに特定のタイプのデータがカプセル化されます。実行コアは、以下の実行スタックを備えています。

- Y 汎用整数
- Y SIMD 整数および浮動小数点
- Y x87

実行コアは、キャッシュ階層との接続も備えています。ロードされたデータはキャッシュからフェッチされ、いずれかのスタックに書き戻されます。

スケジューラーは各ポートで 1 つずつ、サイクルごとに最大 6 つのマイクロオペレーション (μop) をディスパッチできます。表 2-15 は、どの操作をどのポートにディスパッチできるかをまとめたものです。

表 2-15 ディスパッチ・ポートと実行スタック

	ポート 0	ポート 1	ポート 2	ポート 3	ポート 4	ポート 5
<b>整数</b>	ALU、 Shift	ALU、 Fast LEA、 Slow LEA、 MUL	Load_ Addr、 Store_ addr	Load_Adr Store_addr	Store_data	ALU、 Shift、 Branch、 Fast LEA
<b>SSE-Int、 AVX-Int、 MMX</b>	Mul、 Shift、 STTNI、 Int- Div、 128b-Mov	ALU、 Shuf、 Blend、 128b-Mov			Store_data	ALU、 Shuf、 Shift、 Blend、 128b-Mov
<b>SSE-FP、 AVX- FP_low</b>	Mul、 Div、 Blend、 256b- Mov	Add、 CVT			Store_data	Shuf、 Blend、 256b-Mov
<b>X87、 AVX- FP_High</b>	Mul、 Div、 Blend、 256b- Mov	Add、 CVT			Store_data	Shuf、 Blend、 256b-Mov

実行後、ディスパッチ・ポートと結果のデータ型に応じて、ライトバック・バスに書き戻されます。同じポートでディスパッチされ、レイテンシーが異なるマイクロオペレーション (μop) は同じサイクルでライトバック・バスが必要となることがあります。このような場合、ライトバック・バスが使用可能になるまで、いずれか 1 つのマイクロオペレーション (μop) の実行が遅延されます。例えば、MULPS (5 サイクル) と BLENDPS (1 サイクル) の両方がポート 0 で実行準備が整った場合、衝突が発生し、最初に MULPS が実行され、4 サイクル後に BLENDPS が実行されます。インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> では、マイクロオペレーション (μop) が結果を異なるスタックに書き出す限り、このような衝突は解消されます。例えば、整数 ADD (1 サイクル) は整数スタックを使用し、MULPS (5 サイクル) は FP スタックを使用するため、MULPS の 4 サイクル後に整数 ADD をディスパッチできます。

あるスタックで実行されるマイクロオペレーション (μop) のソースが、別のスタックで実行されるマイクロオペレーション (μop) からのものである場合、1 サイクルまたは 2 サイクルの遅延が生じる可能性があります。インテル® SSE 整数操作とインテル® SSE 浮動小数点操作の間の遷移でも遅延が発生します。このような場合、命令フローに

追加されるマイクロオペレーション (μop) によって、データ遷移が行われることがあります。以下の表 2-16 に、実行後にライトバックされるデータが以下のサイクルでどのようにマイクロオペレーション (μop) 実行にバイパスできるかを示します。

表 2-16 実行コアのライトバック・レイテンシー (サイクル数)

	整数	SSE-Int、AVX-Int、MMX	SSE-FP、AVX-FP_low	X87、AVX-FP_High
整数	0	micro-op (ポート 0)	micro-op (ポート 0)	micro-op (ポート 0) +1 サイクル
SSE-Int、AVX-Int、MMX	micro-op (ポート 5) または micro-op (ポート 5) +1 サイクル	0	1 サイクル遅延	0
SSE-FP、AVX-FP_low	micro-op (ポート 5) または micro-op (ポート 5) +1 サイクル	1 サイクル遅延	0	micro-op (ポート 5) +1 サイクル
X87、AVX-FP_High	micro-op (ポート 5) +1 サイクル	0	micro-op (ポート 5) +1 サイクル	0
ロード	0	1 サイクル遅延	1 サイクル遅延	2 サイクル遅延

## 2.4.5 キャッシュ階層

キャッシュ階層には、1 次命令キャッシュ、1 次データキャッシュ (L1D キャッシュ)、2 次(L2) キャッシュがそれぞれのコアの中に含まれます。L1D キャッシュは、インテル® ハイパースレッディング・テクノロジーをサポートしていれば、2 つの論理プロセッサで共有されます。L2 キャッシュは命令とデータで共有されます。物理プロセッサ・パッケージ内のすべてのコアは、リング接続により、共有されるラスト・レベル・キャッシュ (LLC) と接続します。

キャッシュは、命令トランслーション・ルックアサイド・バッファ (ITLB)、データトランслーション・ルックアサイド・バッファ (DTLB)、共有トランслーション・ルックアサイド・バッファ (STLB) の各種機能を使用して、リニアアドレスを物理アドレスに変換します。すべてのキャッシュレベルのデータ・コヒーレンシーは MESI プロトコルを使用して維持されます。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3C』を参照してください。キャッシュ階層の詳細については、実行時に CPUID 命令を使用して取得できます。『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』を参照してください。

表 2-17 キャッシュ・パラメーター

Level	Capacity	Associativity (ways)	Line Size (bytes)	Write Update Policy	Inclusive
L1 Data	32 KB	8	64	Writeback	-
Instruction	32 KB	8	N/A	N/A	-
L2 (Unified)	256 KB	8	64	Writeback	No
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback	Yes

### 2.4.5.1 ロード操作とストア操作の概要

この節では、ロード操作とストア操作の概要を示します。

#### ロード

ライトバック (WB) 方式のメモリー・ロケーションから命令がデータを読み出す場合、プロセッサはキャッシュおよびメモリー内を検索します。表 2-18 に、アクセス・ルックアップ順序およびベストケースのレイテンシーを示します。実際のレイテンシーはキャッシュキューの空き具合、LLC リングの空き具合、メモリー・コンポーネント、そのパラメーターによって変わることがあります。

表 2-18 ルックアップ順序とロード・レイテンシー

レベル	レイテンシー (サイクル数)	帯域幅 (1 コアあたり、1 サイクルあたり)
L1 データ	4 <sup>1</sup>	2x16 バイト
L2 (ユニファイド)	12	1x32 バイト
3 次 (LLC)	26-31	1x32 バイト
その他のコア内の L2 キャッシュおよび L1 D キャッシュ (該当する場合)	43- クリーンヒット 60- ダーティーヒット	

**注意:**

1. 表 2-16 に示すように、実行コアのバイパス制約を受けます。
2. L3 のレイテンシーは製品セグメントと SKU によって異なります。値は、第 2 世代インテル® Core™ プロセッサ・ファミリーに適用されます。

LLC はそれよりも上位のすべてのキャッシュレベルを含みます。コアキャッシュに含まれるデータは LLC にもなければいけません。LLC の各キャッシュラインは、L2 キャッシュと L1 キャッシュにこのラインを含む可能性があるコアの存在を示します。他のコアに目的のラインが含まれることが LLC で示され、その状態を変更する必要がある場合、それらのコアの L1D キャッシュと L2 キャッシュのルックアップを行います。他のコアキャッシュからのデータのフェッチを必要としない場合、このルックアップは「クリーン」ルックアップと呼ばれます。変更されたデータを他のコアキャッシュからフェッチし、ローディング・コアに転送する必要がある場合は、このルックアップを「ダーティー」ルックアップと呼びます。

上記に示すレイテンシーはベストケースのシナリオです。変更されたキャッシュラインを排出して、新しいキャッシュライン用にスペースを確保しなければならないことがあります。変更されたキャッシュラインは新しいデータの取得と並行して排出されるので、追加のレイテンシーは必要としません。ただし、データがメモリーにライトバックされる際は、キャッシュ帯域幅のほか、メモリー帯域幅も排出に使用される可能性があります。したがって、変更されたキャッシュラインの排出を伴うキャッシュミスが短時間に複数発生した場合、キャッシュ応答時間が全体的に低下します。メモリー・アクセス・レイテンシーは、メモリー・コントローラー・キューの空き具合、DRAM コンフィグレーション、DDR パラメーター、DDR ページング動作 (要求されたページがページヒット、ページミス、ページ・エンプティのいずれかの場合) によって異なります。

**ストア**

命令がライトバック方式のメモリー・ロケーションにデータを書き込む際、プロセッサはまず L1D キャッシュに排他または変更 MESI 状態でこのメモリー・ロケーションを含むラインがあることを確認します。キャッシュラインがそこに正しい状態で存在しない場合、プロセッサは所有権読み出し要求を使用して、次のメモリー階層レベルからフェッチします。プロセッサは、指定された順序で以下の場所からキャッシュラインを検索します。

1. L1D キャッシュ
2. L2
3. ラスト・レベル・キャッシュ
4. その他のコア内の L2 キャッシュおよび L1D キャッシュ (存在する場合)
5. メモリー

キャッシュラインが L1D キャッシュにあると、そこに新しいデータが書き込まれ、そのラインが変更済みとしてマークされます。

所有権読み出しとデータのストアは、命令のリタイアメントの後に発生し、ストア命令のリタイアメントの順序に従います。したがって、ストア・レイテンシーは、通常、ストア命令自体には影響を与えることはありません。ただし、L1D

キャッシュをミスする一部の連続したストアは、レイテンシーを累積してパフォーマンスに影響を与えることがあります。ストアが完了しない間は、ストアバッファ内のエントリが消費されます。ストアバッファがいっぱいになると、新しいマイクロオペレーション (μop) が実行パイプに入ることができず、実行がストールする可能性があります。

## 2.4.5.2 L1D キャッシュ

L1D キャッシュは、1 次データキャッシュです。内部データ構造を通過するすべてのタイプからのすべてのロード要求とストア要求を管理します。

L1D キャッシュ:

- Y ロードとストアを投機的にアウトオブオーダー発行することができます。
- Y リタイアしたロードとストアがリタイア時に正確なデータを得られます。
- Y ロードとストアが IA-32 アーキテクチャーとインテル® 64 命令セット・アーキテクチャーのメモリの順序付け規則に従うことを確実にします。

表 2-19 L1 データキャッシュの構成要素

構成要素	インテル® マイクロアーキテクチャー Sandy Bridge	インテル® マイクロアーキテクチャー Nehalem
データ・キャッシュ・ユニット (DCU)	32KB、8 ウェイ	32KB、8 ウェイ
ロードバッファ	64 エントリ	48 エントリ
ストアバッファ	36 エントリ	32 エントリ
ライン・フィル・バッファ (LFB)	10 エントリ	10 エントリ

DCU は 32KB の 8 ウェイ・セット・アソシアティブとして編成されます。キャッシュライン・サイズは 8 つのバンクに配置された 64 バイトです。

内部的には、アクセスは最大 16 バイトで、256 ビットのインテル® AVX 命令が 2 つの 16 バイト・アクセスを使用します。各サイクルで、2 つのロード操作と 1 つのストア操作を処理できます。

L1D キャッシュは、すぐに処理できない要求を完了するまで保持します。要求を遅延する理由としては、キャッシュミス、キャッシュライン間で分割されるアライメントの合っていないアクセス、先行するストアからフォワードされる準備が整っていないデータ、バンクの衝突が発生するロード、キャッシュラインの置換によるロードブロックなどがあります。

L1D キャッシュは、割り当てからリタイアメントまで最大 64 のロード・マイクロオペレーション (μop) を保持できます。割り当てからストア値がキャッシュにコミットされるまで、または非テンポラルなストアの場合はライン・フィル・バッファ (LFB) に書き込まれるまで、最大 36 のストア操作を保持できます。

L1D キャッシュは、複数の未処理のキャッシュミス処理し、後続のストアとロードの処理を続けることができます。LFB を利用することにより、最大 10 のキャッシュライン・ミスと同時に管理できます。

L1D キャッシュはライトバック、ライトアロケート・キャッシュです。DCU にヒットするストアは、より下位のメモリ階層を更新しません。DCU が見つからないストアはキャッシュラインを割り当てます。

### ロード

L1D キャッシュ・アーキテクチャーは、1 サイクルで 2 つのロードを処理でき、それぞれのロードは最大 16 バイトまで可能です。アウトオブオーダー・エンジンでの割り当てから、ロードされた値が実行コアに戻されるまで、さまざまな進捗段階で最大 32 のロードを保持できます。ロードでは以下を実行できます。



- Y ロードアドレスとストアアドレスの範囲が競合していないことが明らかな場合は、先行するストアの前にデータを読み取ることができます。
- Y 先行する分岐が解決される前に、投機実行できます。
- Y キャッシュミスにアウトオブオーダーでオーバーラップして対応します。

ロードでは以下を実行できません。

- Y あらゆるフォルトやトラップに対し投機的に対応すること。
- Y キャッシュできないメモリーに投機的にアクセスすること。

一般的なロード・レイテンシーは 5 サイクルです。シンプルなアドレス指定モードを使用している場合、2048 以下のベース + オフセットでは、4 サイクルのロード・レイテンシーが可能です。この手法は特にポインター追跡コードに便利です。ただし、スタックバイパスにより、ターゲットレジスターのデータ型に応じて、全体的なレイテンシーが変わります。詳細については、2.4.4 節を参照してください。

表 2-20 は、全体的なロード・レイテンシーの一覧です。これらのレイテンシーではフラットセグメントの一般的なケース（つまり、セグメントのベースアドレスがゼロであること）を前提としています。セグメントベースがゼロでない場合、ロード・レイテンシーは増大します。

表 2-20 ロード・レイテンシーに対するアドレス指定モードの影響

データタイプ/アドレス指定モード	ベース + オフセット > 2048; ベース + インデックス [+ オフセット]	ベース + オフセット < 2048
整数	5	4
MMX、SSE、128 ビット AVX	6	5
X87	7	6
256 ビット AVX	7	7

## ストア

メモリーへのストアは、2 つのフェーズで実行されます。

- Y 実行フェーズ: ストアバッファをリニアアドレス、物理アドレス、およびデータでいっぱいにします。ストアアドレスとデータが判明していれば、そのストアデータを必要とする以降のロード操作に転送できます。
- Y 完了フェーズ: ストアのリタイア後、L1D キャッシュはストアバッファから DCU に、1 サイクルあたり最大 16 バイトのデータを移動できます。

## アドレス変換

DTLB はサイクルごとに、ロードアドレスに 2 つ、ストアアドレスに 1 つの合計 3 つのリニアアドレスから物理アドレスへの変換を行うことができます。DTLB でアドレスが見つからない場合、プロセッサはデータおよび命令アドレス変換を保持する STLB を検索します。STLB にヒットする DTLB ミスのペナルティーは 7 サイクルです。ラージ・ページ・サポートには、4K ページと 2M/4M ページに加え、1G バイト・ページが含まれます。

DTLB および STLB は 4 ウェイ・セット・アソシアティブです。表 2-21 に、DTLB および STLB のエントリーの数を示します。

表 2-21 DTLB および STLB のパラメーター

TLB	ページサイズ	エントリー
DTLB	4KB	64
	2MB/4MB	32
	1GB	4
STLB	4KB	512

### ストア・フォワーディング

ストアに続いてロードを行い、ストアによってメモリーに書き込まれたデータを再読み込みする場合、ストア操作からロードにデータを直接転送できます。これはストア・ロード・フォワーディングと呼ばれ、ロードがメモリーを介さずにストア操作から直接データを取得できるので、サイクル数の節約になります。ストア・フォワーディングを利用すると、サブフィールドを転送する能力を犠牲にすることなく、複雑な構造を移動できます。メモリー制御ユニットは、以前のマイクロアーキテクチャーと比べ、より少ない制約でストア・フォワーディングを処理できます。

ストア・ロード・フォワーディングを行うには、以下の規則に従わなければなりません。

- Y ストアは、ロードに先行する、該当アドレスへの最後のストアでなければなりません。
- Y ストアに、ロードされているすべてのデータが含まれていなければなりません。
- Y ロードはライトバック・メモリーからで、ロードもストアも非テンポラルアクセスであってはなりません。

以下の場合、ストアはロードに転送できません。

- Y 先行する 16 バイトまたは 32 バイトのストアに対して、8 バイト境界をまたがる 4 バイトおよび 8 バイトのロード。
- Y 32 バイト・ストアの 16 バイト境界をまたがるロード。

表 2-22 から表 2-25 に、ストア・ロード・フォワーディング動作の詳細を示します。一定のストアサイズに対して、オーバーラップする可能性があるすべてのロードが示され、「F」記号で示されています。32 バイト・ストアからの転送は、半分の各 16 バイト・ストアからの転送と類似します。転送できないケースは「N」として示されます。

表 2-22 ストア・フォワーディング条件 (1 バイト・ストアおよび 2 バイト・ストア)

ストア サイズ	ロード サイズ	ロード・アライメント															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	F														
	2	F	N														

表 2-23 ストア・フォワーディング条件 (4 - 16バイト・ストア)

		ロード・アライメント															
ストア サイズ	ロード サイズ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	F	F	F												
	2	F	F	F	N												
	4	F	N	N	N												
8	1	F	F	F	F	F	F	F	F								
	2	F	F	F	F	F	F	F	N								
	4	F	F	F	F	F	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

表 2-24 ストア・フォワーディング条件 (0 - 15 バイト・ストア)

		ロード・アライメント															
ストア サイズ	ロード サイズ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

表 2-25 ストア・フォワーディング条件 (16 - 31 バイト・ストア)

		ロード・アライメント															
ストア サイズ	ロード サイズ	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

### メモリー・ディスアンビゲーション

ロード操作は先行するストアに依存する可能性があります。多くのマイクロアーキテクチャーでは、先行するストアのアドレスがすべて判明するまで、ロードがブロックされます。メモリー・ディスアンビゲーション機能は、どのロードが先行するストアに依存しないかを予測します。ロードにそのような依存関係がないことをディスアンビゲーション機能が予測した場合、たとえストアアドレスが判明していなくても、そのロードは 1 次データキャッシュからデータを取得します。これによって、ロード・レイテンシーが隠蔽されます。最終的に予測が検証され、実際に競合が検出されると、ロードとすべての後続命令が再実行されます。

以下のロードはディスアンピゲーションの対象外です。先行するすべてのストアのアドレスが判明するまで、これらのロードの実行はストールします。

- Y 16 バイト境界にまたがるロード。
- Y 32 バイトにアライメントされていない 32 バイトのインテル® AVX ロード。

メモリー・ディスアンピゲーション機能では、ロードと同じアドレスビット 0:11 を持つ先行するストアの間との依存関係を常に前提とします。

## バンクの競合

16 バイト・ロードは最大 3 つのバンクをカバーし、1 サイクルで 2 つのロードが可能であるため、ロードに対して各サイクルで 8 つのバンクのうち 6 つにアクセスできます。異なるセットで 2 つのロードアクセスが同じバンクを同時に必要とした (アドレスが同じ 2 ~ 4 ビット値を持つ) 場合にバンクの競合が発生します。バンクの競合が発生すると、ロードアクセスの一方が内部で再試行されます。

多くの場合、スタックからオペランドを取り出す場合やシーケンシャル・アクセスの場合に、2 つのロードは同じキャッシュラインの全く同じバンクにアクセスします。このような場合、競合は発生せず、同時にロードが処理されません。

### 2.4.5.3 リング・インターコネクトとラスト・レベル・キャッシュ

システムオンチップ設計により、IA コアとアンコアの各種サブシステム間を接続するための高帯域幅の双方向リングバスを提供します。第 2 世代のインテル® Core™ プロセッサ 2xxx 番台では、アンコア・サブシステムとして、システム・エージェント、グラフィックス・ユニット (GT)、ラスト・レベル・キャッシュ (LLC) が含まれます。

LLC は複数のキャッシュスライスで構成されます。スライスの数は IA コアの数と同じです。各スライスには、ロジック部分とデータ配列部分があります。ロジック部分は、データ・コヒーレンシー、メモリアクセス順序、データ配列部分へのアクセス、LLC ミス、メモリーへのライトバックなどを処理します。データ配列部分はキャッシュラインを格納します。各スライスには、32 バイト/サイクルを供給できるフル・キャッシュ・ポートが含まれます。

LLC データ配列に保持されるデータの物理アドレスは、アドレスが均一に分散されるように、ハッシュ関数によってキャッシュスライス間に分散されます。キャッシュブロック内のデータ配列は、0.5M/1M/1.5M/2M のブロックサイズに合わせ、4/8/12/16 のウェイの構成が可能です。ただし、キャッシュブロック間でのアドレス分散がソフトウェアの観点からであるため、これは通常の N ウェイキャッシュとして見えることはありません。

プロセッサ・コアおよび GT から見ると、LLC は複数のポートと、コアの数に応じてスケールする帯域幅を備えた 1 つの共有キャッシュとして機能します。LLC ヒット・レイテンシー (26 ~ 31 サイクル) は、LLC ブロックに対するコアの位置、およびその要求がリング上をどこまで伝わる必要があるかによって左右されます。

キャッシュブロックの数はコアの数に応じて増えるため、リングおよび LLC がコア操作に対して帯域幅を制限する可能性はありません。

GT も同じリング・インターコネクト上にあり、データ操作に LLC を使用します。この点では、IA コアと非常に類似しています。そのため、キャッシュ帯域幅と大きなキャッシュ領域を使用する高帯域幅のグラフィックス・アプリケーションの場合、多少コア操作に干渉するおそれがあります。

LLC ミス、ダーティーなライン・ライトバック、キャッシュ不能な操作、MMIO/IO 操作など、LLC で満足させることができないすべてのトラフィックが、キャッシュブロックのロジック部分およびリングを通り、システム・エージェントに伝わります。

インテル® Xeon® プロセッサ E5 ファミリーのアンコア・サブシステムは、グラフィックス・ユニット (GT) を含みません。代わりに、アンコア・サブシステムは、大きな容量と複数のプロセッサをサポートするスヌーピング機能を備え

た LCC、複数ソケットのプラットフォームをサポートするインテル® QuickPath インターコネクト・ファブリック、電力管理制御ハードウェア、およびメモリーと I/O デバイスからの高帯域幅のトラフィックをサポートするシステム・エージェント機能など、多くのコンポーネントを備えています。

インテル® Xeon® プロセッサー E5 2xxx または 4xxx 製品ファミリーでは、LLC の容量は 1 コアあたり 2.5M バイトでプロセッサーのコア数によって異なります。

#### 2.4.5.4 データ・プリフェッチ

ソフトウェア・プリフェッチ、ハードウェア・プリフェッチ、またはその 2 つの組み合わせを使用して、L1D キャッシュに投機的にデータをロードできます。

4 つのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) プリフェッチ命令を使用して、ソフトウェア制御プリフェッチを行うことができます。このような命令は、データを含むキャッシュラインを要求されるレベルのキャッシュ階層に移動する際のヒントとして機能します。ソフトウェア制御プリフェッチは、コードのプリフェッチではなく、データのプリフェッチを対象とします。

この節の残りでは、インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> によって提供される各種ハードウェア・プリフェッチ・メカニズム、およびこれまでのプロセッサーに対する改善点について説明します。プリフェッチの目的は、プログラムが消費しそうなデータを事前に予測することです。このデータが実行コアまたは内部キャッシュの近くにならない場合、プリフェッチでは、次のレベルのキャッシュ階層およびメモリーからデータを取り込みます。プリフェッチは以下の影響を及ぼします。

- Y データがプログラムで使用される順序で連続的に配置されている場合、パフォーマンスが向上します。
- Y アクセスパターンが局所的でなくまばらな場合、帯域幅の問題によりわずかにパフォーマンスが低下することがあります。
- Y まれに、アルゴリズムのワーキングセットがキャッシュの大部分を占めるようにチューニングされ、プログラムから要求されたキャッシュラインが不要なプリフェッチによって排出されている場合、1 次キャッシュの容量が原因でハードウェア・プリフェッチが重大なパフォーマンス低下をもたらすことがあります。

##### L1 データキャッシュに対するデータ・プリフェッチ

データ・プリフェッチは、以下の条件が満たされた場合にロード操作によりトリガーされます。

- Y ライトバック・メモリーからのロードである。
- Y プリフェッチされるデータが、トリガーしたロード命令と同じ 4K バイト・ページ内にある。
- Y パイプライン内でフェンスが発生していない。
- Y その他のロードミスがあまり発生していない。
- Y 連続的なストアのストリームが発生していない。

L1D キャッシュにデータをロードするハードウェア・プリフェッチは 2 つあります。

- Y **データ・キャッシュ・ユニット(DCU)プリフェッチャー:** このプリフェッチはストリーミング・プリフェッチとも呼ばれ、最後にロードされたデータへのユニット・ストライド・アクセスによってトリガーされます。プロセッサーは、このアクセスがストリーミング・アルゴリズムの一部であるとみなし、次のラインを自動的にフェッチします。
- Y **間隔を空けた命令ポインター(IP)ベースのプリフェッチャー:** このプリフェッチは、個々のロード命令を追跡します。ロード命令に規則的なアドレススキップがあることが検出された場合、現在のアドレスとこの間隔の合計である次のアドレスがプリフェッチされます。前方または後方へのプリフェッチが可能であり、最大で 2K バイトの間隔に対処できます。

## L2 キャッシュおよびラスト・レベル・キャッシュに対するデータ・プリフェッチャー:

メモリーから L2 キャッシュおよびラスト・レベル・キャッシュへデータをプリフェッチするハードウェア・プリフェッチは以下の 2 つです。

**空間的プリフェッチャー:** L2 キャッシュにフェッチされたすべてのキャッシュラインを、128 バイトにアライメントされたチャンクとなるペアラインにしようとします。

**ストリーマー:** 昇順および降順のアドレスシーケンスに対して、L1 キャッシュからの読み込み要求を監視します。監視される読み込み要求には、ロード操作とストア操作およびハードウェア・プリフェッチによって開始された L1D キャッシュ要求、およびコードフェッチに対する L1 命令キャッシュ要求が含まれます。前方または後方の要求ストリームが検出されると、予想されるキャッシュラインがプリフェッチされます。プリフェッチされるキャッシュラインは同じ 4K ページになければいけません。

ストリーマーおよび空間的プリフェッチでは、ラスト・レベル・キャッシュにデータをプリフェッチします。見つからない要求が大量に L2 キャッシュにロードされていない限り、通常、L2 にもデータが取り込まれます。

ストリーマーの改善点として、以下の機能が挙げられます。

- Y ストリーマーは、L2 ルックアップごとに 2 つのプリフェッチ要求を発行できます。ストリーマーはロード要求に先立ち、最大 20 のラインを処理できます。
- Y 1 コアあたりの未処理の要求の数に合わせて動的に調整を行います。未処理の要求がそれほど多くない場合、ストリーマーはさらに先立ってプリフェッチを行います。未処理の要求が多い場合は、LLC にのみ低頻度のプリフェッチを行います。
- Y キャッシュラインがはるか先にある場合、ラスト・レベル・キャッシュにのみプリフェッチを行い、L2 に対してはプリフェッチを行いません。この方法により、L2 キャッシュ内の有用なキャッシュラインが置換されなくなります。
- Y 最大 32 のデータ・アクセス・ストリームを検出および保持できます。4K バイト・ページごとに、1 つの前方ストリームと 1 つの後方ストリームを保持します。

## 2.4.6 システム・エージェント

第 2 世代のインテル® Core™ プロセッサ・ファミリーのシステム・エージェントには、以下の構成要素が含まれます。

- Y アービター: リングドメインおよび I/O (PCIe\* および DMI) からのすべてのアクセスを処理し、所定の場所にルーティングします。
- Y PCIe コントローラー: 外部 PCIe\* デバイスに接続します。PCIe\* コントローラーは可能なコンフィギュレーションがさまざまで、製品セグメントの詳細に応じて異なります (x16+x4, x8+x8+x4, x8+x4+x4+x4)。
- Y DMI コントローラー: PCH チップセットに接続します。
- Y 内部のグラフィック操作のための統合ディスプレイ・エンジン、フレキシブル・ディスプレイ・インターコネクト、およびディスプレイ・ポート。
- Y メモリー・コントローラー

メイン・メモリー・トラフィックは、すべてアービターからメモリー・コントローラーにルーティングされます。第 2 世代インテル® Core™ プロセッサ 2xxx 番台のメモリー・コントローラーは、2 つの DDR チャンネルをサポートし、ユニットタイプ、システム構成、DRAM に応じて、データレートは 1066MHz、1333MHz、1600MHz、および 1 サイクルあたり 8 バイトです。最高の帯域幅、最小の Hotspot 衝突を実現するために、チャンネル間のロードのバランスを取るローカルなハッシュ関数に基づき、メモリー・チャンネル間でアドレスが分散されます。

最高のパフォーマンスを実現するために、両方のチャンネルに同じメモリー量、できれば全く同じタイプの DIMM を設定することが推奨されます。さらに、同じメモリー量にさらに多くのランクを使用すると、さらに多くの DRAM ページを同時に開くことができるため、メモリー帯域幅が多少向上します。最高のパフォーマンスを実現するために、最高

の DRAM タイミングを備えたサポート速度が最高の DRAM (最高のサポート周波数に応じて、1333MHz または 1600MHz のデータレート)をシステムに設定します。

2 つのチャンネルは固有のリソースを持ち、メモリー要求を個別に処理します。メモリー・コントローラーには、レイテンシーを最低限に抑えながら最高のメモリー帯域幅を実現する高性能のアウトオブオーダー・スケジューラーが内蔵されています。各メモリーチャンネルには、32 キャッシュラインのライトデータバッファーがあります。メモリー・コントローラーへの書き込みは、ライトデータバッファーへ書き込まれた時点で完了とみなされます。ライトデータバッファーは、その後ライト・レイテンシーに影響を及ぼさなくなった時点で、メインメモリーにフラッシュされます。

パーシャル書き込みはメモリー・コントローラーでは効率的に処理されず、パーシャル書き込みが時間内にキャッシュライン全体を完了できない場合、DDR チャンネルは [読み出し-変更-書き込み] 操作となります。ソフトウェアではできる限りパーシャル書き込みトランザクションを行わないようにし、パーシャル書き込みをフル・キャッシュライン書き込みにバッファリングするなど、別の方法を考慮する必要があります。

メモリー・コントローラーは優先度の高い等時性要求 (USB 等時性要求、ディスプレイ等時性要求など) もサポートします。統合ディスプレイ・エンジンからの高帯域幅メモリー要求は、メモリー帯域幅の一部を占有し、コア・アクセス・レイテンシーに多少影響が生じます。

## 2.4.7 インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge

インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge をベースとした第 3 世代のインテル® Core™ プロセッサです。2.4.1 節 から 2.4.6 節で示される多くの機能は、Ivy Bridge† マイクロアーキテクチャーにも適用されます。この節ではコーディングとパフォーマンスに影響するマイクロアーキテクチャーの違いについて説明します。

新しく追加された命令には次のサポートが含まれます。

- Y 半精度浮動小数値との間の数値変換
- Y NIST SP 800-90A 準拠のハードウェア・ベースの乱数生成器
- Y ユーザーモードのスレッド化のサポートを強化するため、すべての特権リングレイヤーで FS/GS ベースレジスターの読み書きが可能になりました。

ハードウェア・ベースの乱数生成器命令 RDRAND の使い方の詳細は、インテル® デベロッパー・ゾーンで公開されている記事をご覧ください:

<http://software.intel.com/en-us/articles/download-the-latest-bull-mountain-software-implementation-guide/?wapkw=bull+mountain> (英語)

ソフトウェアに利点がある若干のマイクロアーキテクチャーの拡張が行われています。

- Y ハードウェア・プリフェッチャーの拡張: インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge では、ネクスト・ページ・プリフェッチャー (NPP) が追加されました。NPP は、ページ境界へ近づく上方または下方の連続したキャッシュライン・アクセスでトリガーされます。
- Y ゼロレイテンシーのレジスター移動操作: レジスター-レジスター間の MOV 命令のサブセットはフロントエンドで実行され、アウトオブオーダー・エンジンのスケジュールと実行リソースを使用しません。
- Y フロントエンドの拡張: インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge では、マイクロオペレーション (μop) キューは、ソフトウェアがシングルスレッドまたはマルチスレッドで実行されるのにかかわらずなく、それぞれの論理プロセッサへ 28 個のエントリーを提供するため静的に分割されていました。インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge では、一方の論理プロセッサがアクティブでない場合、プロセッサ上でシングルスレッドが実行されるとマイクロオペレーション・キューを 56 エントリー使用することができます。この場合、LSD は 28 エントリー以上を必要とする大きなループ構造を処理できます。
- Y いくつかの命令のレイテンシーとスループットが、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge に比べ改善されています。例えば、256 ビットのパックド浮動小数点除算と平方根命令が高速化されています。また、ROL と ROR 命令も改善されています。

## 2.5 インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャー

インテル® Core™ マイクロアーキテクチャーでは、シングルスレッドのワークロードだけでなくマルチスレッドにおけるワークロードでも優れたパフォーマンスと電力効率を実現する以下の機能が採用されています。

- Y **インテル® ワイド・ダイナミック・エグゼキューション:** 各プロセッサ・コアが 1 サイクルあたり最大 4 命令のフェッチ、ディスパッチ、高帯域幅での実行、リタイアを行います。これには以下の機能が含まれます。
  - 14 ステージの効率的なパイプライン
  - 3 つの演算論理ユニット
  - 1 サイクルあたり最大 5 命令をデコード可能な 4 つのデコーダー
  - フロントエンドのスループットを高めるマクロフュージョンとマイクロフュージョン
  - 1 サイクルあたり最大 6 マイクロオペレーション (μop) をディスパッチ可能な発行能力
  - 1 サイクルあたり最大 4 マイクロオペレーション (μop) のリタイアメント帯域幅
  - 高度な分岐予測
  - 関数やプロシーチャーの開始/終了を効率化するスタックポインター追跡機能
- Y **インテル® アドバンスド・スマート・キャッシュ:** シングルスレッド・アプリケーションとマルチスレッド・アプリケーションのいずれの場合でも、2 次キャッシュからコアへの帯域幅を拡大し、最適なパフォーマンスと柔軟性を提供します。これには以下の機能が含まれます。
  - マルチコアおよびシングルスレッドの実行環境向けに最適化
  - 2 次キャッシュから 1 次データキャッシュへの帯域幅を拡大する 256 ビットの内部データパス
  - 4MB、16 ウェイ (または 2MB、8 ウェイ) のユニファイド型共有 2 次キャッシュ
- Y **インテル® スマート・メモリー・アクセス:** データ・アクセス・パターンに応じてメモリーからデータをプリフェッチし、アウトオブオーダー実行によりキャッシュミス発生を軽減します。これには以下の機能が含まれます。
  - ハードウェア・プリフェッチによって 2 次キャッシュミスの実効レイテンシーを軽減
  - ハードウェア・プリフェッチによって 1 次データ・キャッシュ・ミスの実効レイテンシーを軽減
  - メモリー・ディスアンビゲーションによってスペキュレーティブ・エグゼキューション・エンジンを効率化
- Y **インテル® アドバンスド・デジタル・メディア・ブースト:** ほとんどの 128 ビット SIMD 命令のパフォーマンスを 1 サイクルのスループットと浮動小数点演算により改善します。これには以下の機能が含まれます。
  - ほとんどの 128 ビット SIMD 命令を 1 サイクルのスループットで実行 (128 ビットのシャッフル/パック/アンパック操作を除く)
  - 1 サイクルあたり最大 8 個の浮動小数点演算を実行可能
  - 3 つの発行ポートを利用して SIMD 命令をディスパッチし、実行することが可能

拡張版インテル® Core™ マイクロアーキテクチャーは、インテル® Core™ マイクロアーキテクチャーの機能をすべてサポートしているほか、以下のような包括的な拡張機能を提供します。

- Y **インテル® ワイド・ダイナミック・エグゼキューション:** 以下の拡張機能が追加されています。
  - 従来の基数 4 の除算器に代わり基数 16 の除算器を採用することで、除算や平方根などレイテンシーが大きい演算を高速化しています。
  - 内部構造の改善によって、RDTSC、STI、CLI、VM 終了の遷移など長いレイテンシーの操作を高速化しました。
- Y **インテル® アドバンスド・スマート・キャッシュ:** 2 つのプロセッサ・コア間で共有される最大 6MB の 2 次キャッシュ (クアドコア・プロセッサは最大 12MB の 2 次キャッシュを搭載)、最大 24 ウェイのセット・アソシアティビティーで構成されます。
- Y **インテル® スマート・メモリー・アクセス:** 最大 1600MHz の高速システムバスをサポートし、キャッシュライン分割されたロードやストアおよびロード・フォワードイングなどのメモリー操作の処理を効率化します。
- Y **インテル® アドバンスド・デジタル・メディア・ブースト:** シャッフル/パック/アンパック操作を高速化する 128 ビットのシャッフルユニットを提供します。これに伴い 47 個のインテル® SSE4.1 命令が追加されました。2.5



節のインテル® Core™ マイクロアーキテクチャーに関する説明の大部分は、拡張版インテル® Core™ マイクロアーキテクチャーにも適用されます。両アーキテクチャーの違いについては、明確に記載しています。

## 2.5.1 インテル® Core™ マイクロアーキテクチャーのパイプラインの概要

インテル® Core™ マイクロアーキテクチャーのパイプラインは以下で構成されます。

- Y インオーダー発行フロントエンド: メモリーから命令ストリームをフェッチし、4 つの命令デコーダーによってデコードされた命令 (マイクロオペレーション:  $\mu\text{op}$ ) をアウトオーダー実行コアに供給します。
- Y アウトオーダー・スーパースケーラー実行コア: 1 サイクルあたり最大 6 マイクロオペレーション ( $\mu\text{op}$ ) を発行できます (表 2-27 を参照)。入力ソースが準備でき実行リソースが利用可能になり次第、実行できるようにマイクロオペレーション ( $\mu\text{op}$ ) をリオーダーします。
- Y インオーダー・リタイアメント・ユニット: マイクロオペレーション ( $\mu\text{op}$ ) の実行結果が反映され、プログラム順序に従ってアーキテクチャー・ステートが更新されるようにします。

インテル® Core™2 Extreme プロセッサ X6800、インテル® Core™2 Duo プロセッサ、インテル® Xeon® プロセッサ 3000/5100 番台には、インテル® Core™ マイクロアーキテクチャー・ベースの 2 つのプロセッサ・コアが実装されています。また、インテル® Core™2 Extreme クアッドコア・プロセッサ、インテル® Core™2 Quad プロセッサ、インテル® Xeon® プロセッサ 3200/5300 番台には、4 つのプロセッサ・コアが実装されています。これらのクアッドコア・プロセッサの各物理パッケージには 2 つのプロセッサ・ダイが搭載され、各ダイには 2 つのプロセッサ・コアが搭載されています。図 2-8 に、各コア内のサブシステムの機能を示します。

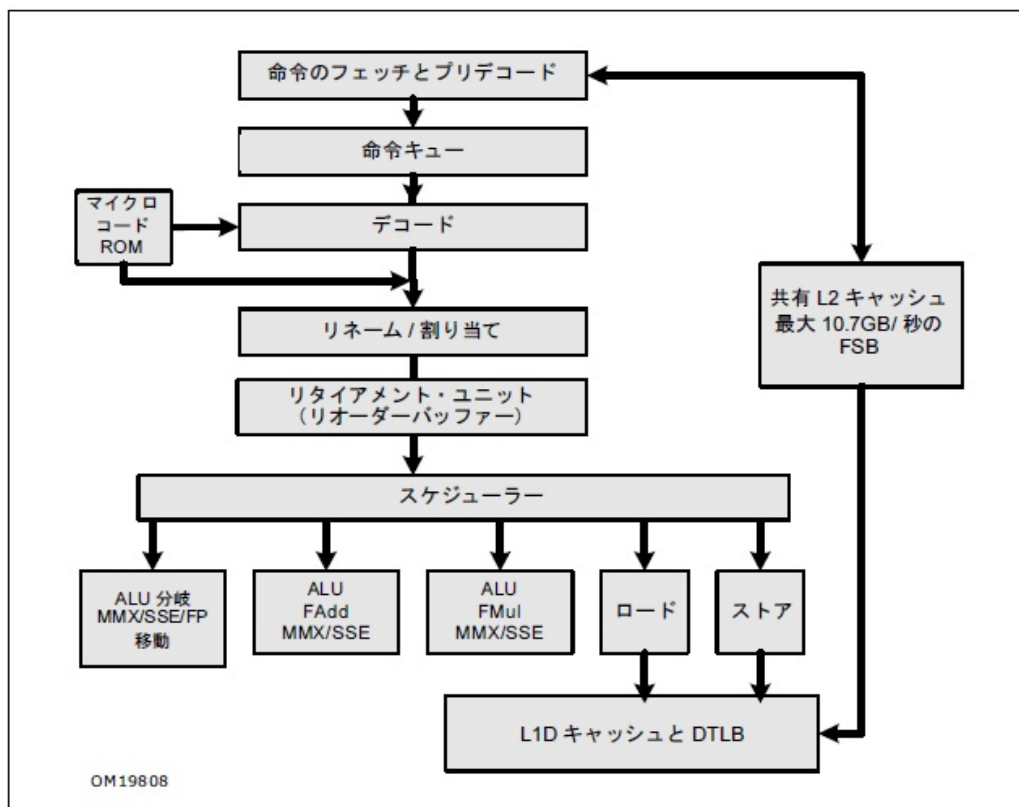


図 2-8 インテル® Core™ マイクロアーキテクチャーのパイプラインの構造

## 2.5.2 フロントエンド

フロントエンドは、デコードされた命令 (マイクロオペレーション:  $\mu\text{op}$ ) ストリームの供給を、6 マイクロオペレーション ( $\mu\text{op}$ ) の発行が可能なアウトオーダー・エンジンへ維持する必要があります。表 2-26 に、フロントエンドの構成要素、それぞれの機能、マイクロアーキテクチャー設計上のパフォーマンスの課題を示します。

表 2-26 フロントエンドの構成要素

構成要	機能	パフォーマンスの課題
分岐予測ユニット (BPU)	<ul style="list-style-type: none"> <li>Y 各種の分岐タイプ (条件付き、間接、直接、コール、リターン) を予測することによって、最も実行される可能性の高い命令を命令フェッチユニットがフェッチできるようにします。分岐タイプごとに専用のハードウェアを使用します。</li> </ul>	<ul style="list-style-type: none"> <li>Y 投機実行を有効にします。</li> <li>Y パイプラインにフェッチされる「非アーキテクチャー・パス」<sup>1</sup> 内のコード量を削減することによって、投機実行を効率化します。</li> </ul>
命令フェッチユニット	<ul style="list-style-type: none"> <li>Y 実行される可能性の高い命令をプリフェッチします。</li> <li>Y 実行頻度の高い命令をキャッシュします。</li> <li>Y 命令をプリデコードおよびバッファリングして、命令ストリームの不規則さにかかわらず一定の帯域幅を維持します。</li> </ul>	<ul style="list-style-type: none"> <li>Y 可変長命令フォーマットは、デコード帯域幅のばらつき (バブル) の原因となります。</li> <li>Y 実行される分岐とアライメントが合っていない分岐先は、フェッチユニットの帯域幅全体に悪影響を及ぼします。</li> </ul>
命令キューとデコードユニット	<ul style="list-style-type: none"> <li>Y 最大 4 命令 (マクロフュージョンの場合は最大 5 命令) をデコードできます。</li> <li>Y スタックポインター追跡アルゴリズムによって、プロシージャーの開始/終了処理を効率化します。</li> <li>Y パフォーマンスと効率の向上を図る、マクロフュージョン機能を提供します。</li> <li>Y 命令キューはループキャッシュとしても使用され、一部のループを高帯域幅および低消費電力で実行できます。</li> </ul>	<ul style="list-style-type: none"> <li>Y 命令あたりの処理量が変化する場合、不定数のマイクロオペレーション (μop) への展開が必要となります。</li> <li>Y プリフィクス付きの命令はデコードが複雑になります。</li> <li>Y レングス変更プリフィクス (LCP) はフロントエンド・バブルの原因となります。</li> </ul>

**注意:**

1. 実行されるとプロセッサが判断しましたが、別のパスの方が適切であることが判明したため、予測からそれたコードパスを指します。

### 2.5.2.1 分岐予測ユニット

分岐予測によって、プロセッサは、分岐の方向が判明する前に、以前に実行した命令を開始できます。すべての分岐は、分岐予測ユニット (BPU) によって予測されます。BPU には以下の機能があります。

- Y 16 エントリーのリターン・スタック・バッファ (RSB): BPU は RET 命令を正確に予測できます。
- Y BPU ルックアップのフロントエンド・キューイング: BPU は、一度に 32 バイト (フェッチエンジンの幅の 2 倍) の分岐予測を行うことができます。これにより、実行される分岐をペナルティーなしで予測できます。
- Y この BPU のメカニズムでは一般的に、実行される分岐のペナルティーを排除できますが、ソフトウェアは、実行されない分岐よりも実行される分岐の方が多くのリソースを消費するとみなすべきです。

BPU は以下のタイプの予測を行います。

- Y 直接コールおよびジャンプ: 実行されるか、されないか、いずれの予測にかかわらず分岐先はターゲット配列として読み出されます。
- Y 間接コールおよびジャンプ: 変化しない分岐先を持つか、プログラム動作に応じて変化する分岐先を持つかの、いずれかとして予測されます。
- Y 条件分岐: 分岐先と、分岐が実行されるかどうかを予測します。
- Y ソフトウェアを BPU 向けに最適化する際の詳細については、3.4 節「フロントエンドの最適化」を参照してください。

## 2.5.2.2 命令フェッチユニット

命令フェッチユニットは、命令トランслーション・ルックアサイド・バッファ (ITLB)、命令プリフェッチャー、命令キャッシュ、命令キュー (IQ) のプリデコード・ロジックで構成されます。

### 命令キャッシュと ITLB

命令フェッチは、ITLB 介して命令キャッシュと命令プリフェッチ・バッファへの 16 バイト・アライメントの参照を行います。命令キャッシュでヒットすると、16 バイト分の命令がプリデコーダーに供給されます。実行されるコードによりませんが、通常のプログラムでは命令長は平均 1 命令あたり 4 バイトをわずかに下回ります。ほとんどの命令は、すべてのデコーダーでデコード可能であり、多くの場合、デコーダーは 1 サイクルでフェッチした命令全体をデコードできます。

分岐先のアライメントが合っていないと、フェッチされた 16 バイトのオフセット分だけ命令バイト数が減少します。また、分岐が実行されると、分岐命令以降のバイトがデコードされないため、デコーダーに供給される命令バイト数が減少します。一般的な整数コードでは、平均 10 命令ごとに分岐が実行されます。つまり、3 ~ 4 サイクルごとに「部分的な (パーシャル)」命令フェッチが行われることとなります。

通常、パイプラインのほかの部分ストールすることが多いため、フロントエンド・スタベーションによりパフォーマンスが低下することはありません。例えば、より長い命令 (インテル® SSE2 整数メディアカーネルなど) で構成される極端に高速なコードの場合、分岐先をアライメントすることで命令のスタベーションを防止しメリットが得られる場合があります。

### 命令プリデコーダー

プリデコード・ユニットは、命令キャッシュやプリフェッチ・バッファから 16 バイトを受け入れて、以下のタスクを実行します。

- Y 命令長を判断します。
- Y 命令を修飾するすべてのプリフィクスをデコードします。
- Y デコーダーのために命令の属性ごとにマークを付けます (例えば、「分岐」)。

プリデコード・ユニットは、1 サイクルあたり最大 6 命令を命令キューに書き込むことができます。6 個以上の命令がフェッチに含まれていた場合、すべての命令が命令キューに書き込まれるまで、プリデコーダーは 1 サイクルあたり最大 6 命令の割合でプリデコードを続行します。以降のフェッチがプリデコードを開始できるのは、現在のフェッチが完了してからです。

7 命令がフェッチされた場合、プリデコーダーは最初の 6 命令を 1 サイクルでデコードし、次のサイクルで残りの 1 命令をデコードします。この場合、1 サイクルあたり 3.5 命令のデコードに対応することとなります。1 サイクルあたりの命令数 (IPC) のレートが十分に最適化されていない場合でも、これはほとんどのアプリケーションのパフォーマンスを上回ります。ソフトウェアは通常、命令スタベーションを防ぐため特別なことを行う必要はありません。

以下の命令プリフィクスを使用すると、命令のデコード中に問題が発生します。これらのプリフィクスは、レングス変更プリフィクス (LCP) と呼ばれ、命令長を動的に変更します。

- Y オペランド・サイズ・オーバーライド (66H): ワード即値データを持つ命令の先頭に付きます。
- Y アドレス・サイズ・オーバーライド (67H): リアルモード、16 ビット保護モード、または 32 ビット保護モードで、mod R/M を持つ命令の先頭に付きます。

プリデコーダーは、フェッチラインで LCP に遭遇すると、低速の命令長デコード・アルゴリズムを使用しなければなりません。低速の命令長デコード・アルゴリズムでは、6 サイクルでフェッチがデコードされます (通常は 1 サイクル)。

一般に、プロセッサ・パイプライン内の通常のキューイングでは、LCP のペナルティを隠蔽できません。

インテル® 64 アーキテクチャー命令セットの REX プリフィクス (4xh) は、2 つの命令 (MOV オフセットと MOV 即値) サイズを変更できます。ただし、LCP のペナルティは発生しないので、LCP とはみなされません。

### 2.5.2.3 命令キュー (IQ)

命令キュー (IQ) は、命令プリデコード・ユニットと命令デコーダーの間に存在し、18 命令の深さを持ちます。1 サイクルあたり最大 5 命令を送り出し、1 サイクルあたり 1 つのマクロフュージョンをサポートします。また、18 命令未満のループではループキャッシュとしても機能し、ループキャッシュは、以下のように動作します。

BPU 内にはループストリーム検出器 (LSD) があります。LSD は、IQ からのストリーミングの候補になるループを検出します。該当するループが見つかったら、命令バイトがロックダウンされ、ループは予測ミスによって終了するまで IQ からのストリーミングを許可されます。ループが IQ から実行されると、低消費電力で高い帯域幅を提供します (フロントエンド・パイプラインの残りの多くの機能が停止されるため)。

LSD には以下のメリットがあります。

- Y 実行される分岐による帯域幅の損失がない。
- Y アライメントの合っていない命令による帯域幅の損失がない。
- Y プリデコード・ステージを通過済みなので LCP のペナルティがない。
- Y 命令キャッシュ、BPU、プリデコード・ユニットをアイドル状態にできるので、フロントエンドの消費電力が減少する。

ソフトウェアは、状況に応じてループキャッシュ機能を利用すべきです。ループアンロールとその他のコード最適化を適用すると、ループが大きくなりすぎて、LSD に収まらないことがあります。パフォーマンスが高いコードを作成するには、ループキャッシュ機能をオーバーフローしても、通常はループアンロールを行う方がパフォーマンスの観点からは望まれます。

### 2.5.2.4 命令デコード

インテル® Core™ マイクロアーキテクチャーには、4 の命令デコーダーが備わっています。最初のデコーダー 0 は、最大で 4 マイクロオペレーション ( $\mu\text{op}$ ) で構成されるインテル® 64 命令と IA-32 命令をデコードできます。残りの 3 つのデコーダーは、単一マイクロオペレーション ( $\mu\text{op}$ ) の命令を処理します。マイクロシーケンサーは、1 サイクルあたり最大 3 マイクロオペレーション ( $\mu\text{op}$ ) に対応可能であり、5 マイクロオペレーション ( $\mu\text{op}$ ) 以上の命令のデコードを支援します。

すべてのデコーダーは、一般的なケースにおける単一マイクロオペレーション ( $\mu\text{op}$ ) のフロー (マイクロフュージョン、マクロフュージョン、スタックポインター追尾など) をサポートしています。したがって、3 つのシンプルなデコーダーは、単一マイクロオペレーション ( $\mu\text{op}$ ) の命令のデコードに限定されているわけではありません。命令を 4-1-1-1 のテンプレートにパックする必要はなく、また推奨されません。

マクロフュージョンでは、2 つの命令が単一のマイクロオペレーション ( $\mu\text{op}$ ) にマージされます。インテル® Core™ マイクロアーキテクチャーは、32 ビット操作 (インテル® 64 アーキテクチャーの互換サブモードを含む) では 1 サイクルあたり 1 つのマクロフュージョンを処理できます。ただし、バイト長が長い命令を頻繁に使用するコードほどマクロフュージョンのハードウェア・サポートを利用する可能性が少なく、64 ビット・モードでは対応していません。

### 2.5.2.5 スタックポインター追尾

インテル® 64 アーキテクチャーと IA-32 アーキテクチャーには、PUSH、POP、CALL、LEAVE、RET のように、パラメーターの受け渡しやプロシーチャーの開始/終了に共通して使用される命令があります。各命令は、スタック・ポイ

スタック・レジスター (RSP) を暗黙的に更新し、コントロールとパラメーターが組み合わされたスタック操作をソフトウェアの介入なしに行います。これらは、従来のマイクロアーキテクチャーでは複数のマイクロオペレーション (μop) によって実装されていた命令です。

スタックポインター追尾は、この暗黙的な RSP の更新をすべて、デコーダーに搭載されるロジックで行います。この機能には、以下の利点があります。

- Y インテル® Core™ マイクロアーキテクチャーでは PUSH、POP、RET は単一マイクロオペレーション (μop) の命令なので、デコード帯域幅が向上します。
- Y RSP の更新が実行リソースと競合しないので、実行帯域幅の節約になります。
- Y マイクロオペレーション (μop) 間の暗黙的なシリアル依存関係が排除されるので、アウトオブオーダー実行エンジンの並列処理が向上します。
- Y 小型の専用ハードウェアで RSP が更新されるので、電力効率が向上します。

### 2.5.2.6 マイクロフュージョン

マイクロフュージョンでは、同一命令の複数のマイクロオペレーション (μop) が単一の複雑なマイクロオペレーション (μop) に融合されます。複雑なマイクロオペレーション (μop) は、アウトオブオーダー実行コアにディスパッチされます。マイクロフュージョンには、次のようなパフォーマンス上の利点があります。

- Y デコードからリタイアメントへの命令帯域幅が向上します。
- Y 複雑なマイクロオペレーション (μop) では、短いフォーマット (ビット密度が低い形式) の処理が増加するので、一定処理量におけるマシン全体の「ビット・トグル」が減少し、アウトオブオーダー実行エンジンのストレージ量が事実上増加します。そのため消費電力が減少します。

多くの命令には、レジスター方式とメモリー方式があります。メモリーオペランドを含む方式は、レジスター方式よりも長いフローのマイクロオペレーション (μop) にデコードされます。マイクロフュージョンを利用すると、デコード帯域幅の損失を心配することなく、ソフトウェアがメモリーとレジスター間の操作によって、実際のプログラム動作を表現することを可能にします。

### 2.5.3 実行コア

インテル® Core™ マイクロアーキテクチャーの実行コアは、スーパースケalerであり、命令をアウトオブオーダーで処理できます。依存関係チェーンが原因でプロセッサがリソース (2 次データ・キャッシュラインなど) を待機している間、実行コアは別の命令を実行します。このため、1 サイクルあたりに実行される命令数 (IPC) の全体的なレートが増加します。

実行コアは、以下の 3 つのコンポーネントを備えています。

- Y **リネーマー**: マイクロオペレーション (μop) をフロントエンドから実行コアに移動します。アーキテクチャー・レジスターは、多数のセットを備えるマイクロアーキテクチャー・レジスターにリネームされます。リネーミングによって、リードアフターリード・ハザードやライトアフターリード・ハザードと呼ばれる依存関係を排除できます。
- Y **リオーダーバッファ (ROB)**: 各ステージで処理されたマイクロオペレーション (μop) を保持し、完了したマイクロオペレーション (μop) のバッファリング、インオーダーでのアーキテクチャー・ステートの更新、および例外の順序付けを管理します。ROB には、命令をインフライトで処理する 96 個のエントリーがあります。
- Y **リザベーション・ステーション (RS)**: すべてのソースオペランドが準備できるまでマイクロオペレーション (μop) のキューイングを行い、準備ができたマイクロオペレーション (μop) を利用可能な実行ユニットにスケジューリングおよびディスパッチします。RS には 32 個のエントリーがあります。

アウトオブオーダー・コアの初期ステージでは、マイクロオペレーション (μop) がフロントエンドから ROB と RS に移動されます。この過程では、アウトオブオーダー・コアは以下のステップを実行します。

- Y リソース (例えば、ロードバッファやストアバッファ) をマイクロオペレーション (μop) に割り当てます。

- Y マイクロオペレーション (μop) を適切な発行ポートにバインドします。
- Y マイクロオペレーション (μop) のソースとデスティネーションをリネームして、アウトオブオーダー実行を可能にします。
- Y データが即値または計算済みレジスター値の場合、データをマイクロオペレーション (μop) に供給します。

以下では、各種の一般的な操作をコアが効率良く実行する方法について説明します。

- Y **レイテンシーが単一のサイクルであるマイクロオペレーション(μop):** レイテンシーが単一のサイクルであるマイクロオペレーション (μop) のほとんどは、複数の実行ユニットによって実行できるため、複数の依存操作のストリームを迅速に実行できます。
- Y **レイテンシーが長く、頻繁に使用されるマイクロオペレーション(μop):** この種のマイクロオペレーション (μop) は、実行ユニットがパイプライン化されているので、パイプライン内の異なるステージで複数のマイクロオペレーション (μop) を同時に実行できます。
- Y **レイテンシーがデータに依存した操作:** 除算など一部の操作では、レイテンシーがデータに依存します。整数の除算では、オペランドを解析し、オペランドの有意部分のみの計算を実行するため、一般的な小さな数の除算が高速化されます。
- Y **オペランドが特定の制約に一致した場合、レイテンシーが固定された浮動小数点演算:** この制約に一致しないオペランドは例外的なケースとみなされ、長いレイテンシーと低いスループットで実行されます。低スループットのケースは、一般的なケースのレイテンシーとスループットには影響しません。
- Y **1 次キャッシュにヒットしてもレイテンシーが可変のメモリーオペランド:** フォワーディングするかしないかが不明なロードは、ストアアドレスが解決されるまで待機してから実行されます。メモリー・オーダー・バッファ (MOB) は、すべてのメモリー操作を受け入れて処理します。MOB の詳細については、2.5.4 節を参照してください。

### 2.5.3.1 発行ポートと実行ユニット

スケジューラーは、1 サイクルあたり最大 6 つのマイクロオペレーション (μop) を各ポートにディスパッチできます。表 2-27 に、インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャーの発行ポートを示します。前者は CUID シグネチャーの DisplayFamily 値と DisplayModel 値が 06\_0FH によって表され、後者は 06\_17H によって表されます。この表には、一般的な整数演算および浮動小数点 (FP) 演算のレイテンシーとスループットのサイクル数が発行ポートごとに記載されています。

表 2-27 インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャーの発行ポート

実行可能な操作	レイテンシー、スループット		説明 <sup>1</sup>
	シグネチャー = 06_0FH	シグネチャー = 06_17H	
整数 ALU 整数 SIMD ALU FP/SIMD/インテル® SSE2 ムーブと論理演算	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	64 ビット・モードの整数 MUL を含む。 発行ポート 0、ライトバック・ポート 0
単精度 FP MUL 倍精度 FP MUL	4, 1 5, 1	4, 1 5, 1	発行ポート 0、ライトバック・ポート 0
FP MUL (X87) FP シャッフル DIV/SQRT	5, 2 1, 1	5, 2 1, 1	発行ポート 0、ライトバック・ポート 0 FP シャッフルは QW シャッフルを処理しない。
整数 ALU 整数 SIMD ALU FP/SIMD/インテル® SSE2 ムーブと論理演算	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	64 ビット・モードの整数 MUL を除く。 発行ポート 1、ライトバック・ポート 1
FP ADD QW シャッフル	3, 1 1, 1 <sup>2</sup>	3, 1 1, 1 <sup>3</sup>	発行ポート 1、ライトバック・ポート 1

整数ロード FP ロード	3, 1 4, 1	3, 1 4, 1	発行ポート 2、ライトバック・ポート 2
ストアアドレス <sup>4</sup>	3, 1	3, 1	発行ポート 3
ストアデータ <sup>5</sup>			発行ポート 4
整数 ALU 整数 SIMD ALU FP/SIMD/インテル® SSE2 ムーブと 論理演算	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	発行ポート 5、ライトバック・ポート 5
QW シャッフル 128 ビットのシャッフル/パック/ アンパック	1, 1 <sup>2</sup> 2-4, 2-4 <sup>6</sup>	1, 1 <sup>3</sup> 1-3, 1 <sup>7</sup>	発行ポート 5、ライトバック・ポート 5

**注意:**

1. 同じポートを使用するレイテンシーが異なる操作を混在すると、ライトバック・バスの競合が発生して、全体的なスループットが低下する可能性があります。
2. 128 ビット命令は、長いレイテンシーと低いスループットで実行されます。
3. 128 ビット・シャッフル・ユニットはポート 5 を使用します。
4. ストアされるデータのアドレスを使って、ストア・フォワーディングとストア・リタイアメントのロジックを準備します。
5. ストアされるデータを使って、ストア・フォワーディングとストア・リタイアメントのロジックを準備します。
6. 命令によって異なりますが、128 ビット命令は、QW シャッフルユニットを使って実行されます。
7. 命令によって異なりますが、インテル® Core™ マイクロアーキテクチャーでは、QW シャッフルユニットの代わりに 128 ビット・シャッフル・ユニットが使用されます。

RS は、1 サイクルごとに最大 6 マイクロオペレーション (μop) をディスパッチできます。各サイクルでは、最大で 4 つの結果が RS と ROB にライトバックされ、RS はその結果を次のサイクルですぐに使用できます。この高い実行帯域幅によって、デコードおよびリタイアされるマイクロフュージョン済みマイクロオペレーション (μop) の複雑な処理の実行バーストを維持します。

実行コアは、以下の 3 つの実行スタックを備えています。

- Y SIMD 整数
- Y 通常の整数
- Y x87/SIMD 浮動小数点

実行コアは、メモリークラスターとの接続も備えています。図 2-9 を参照してください。

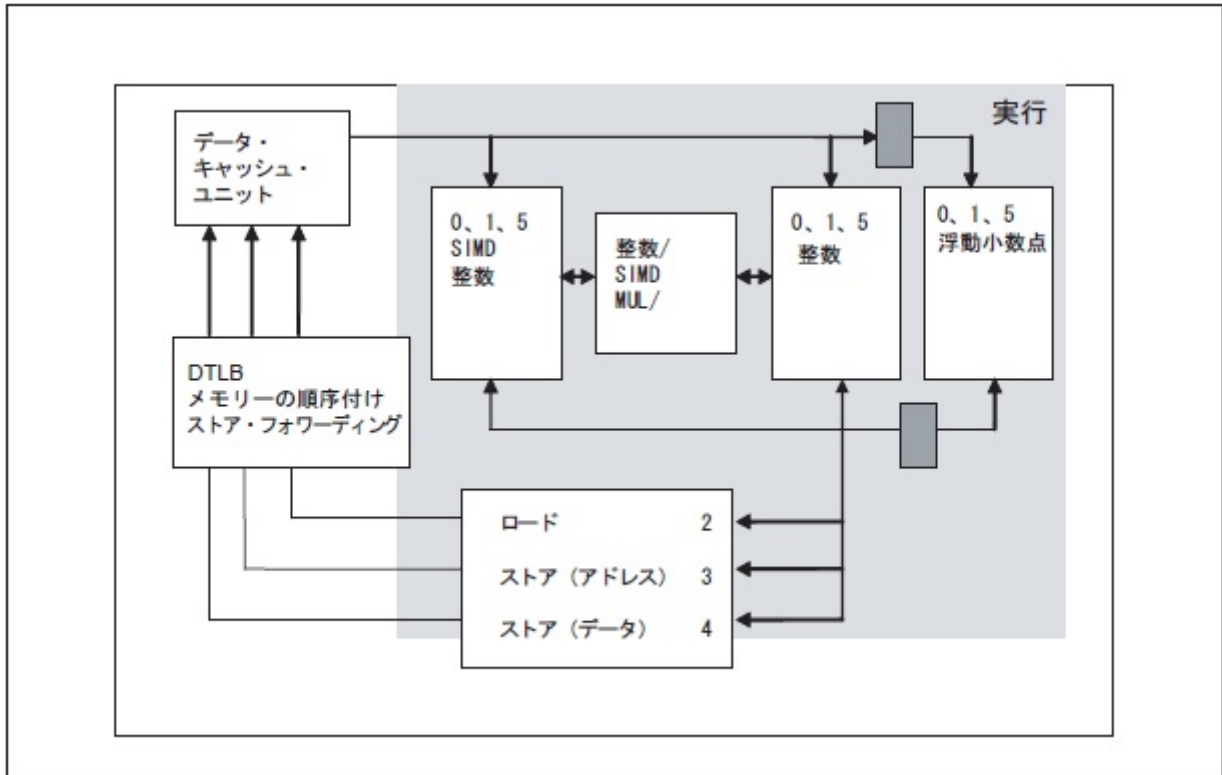


図 2-9 インテル® Core™ マイクロアーキテクチャーの実行コア

実行ブロック (グレー) 内の 2 つの濃い色の四角形に注目してください、これらの四角形は、整数スタックと SIMD 整数スタックを浮動小数点スタックに接続するパス上に存在しています。これは、バイパス遅延と呼ばれる余分なサイクルとなって表れます。1 次キャッシュから浮動小数点ユニットへのデータ転送には、1 サイクルのレイテンシーが追加されます。図 2-9 の濃い色の四角形は、この余分なレイテンシー・サイクルの発生を表しています。

## 2.5.4 インテル® アドバンスド・メモリー・アクセス

インテル® Core™ マイクロアーキテクチャーでは、各コアに 1 次命令キャッシュと 1 次データキャッシュが搭載されています。2 つのコアは 2MB または 4MB の 2 次キャッシュを共有します。すべてのキャッシュはライトバックであり、インクルーシブではありません。各コアは、以下の要素を備えています。

- Y **データ・キャッシュ・ユニット(DCU)と呼ばれる 1 次データキャッシュ** — DCU は、将来発生すると予測される複数のキャッシュミス処理して、後続のストアとロードを迅速に処理することができます。DCU はキャッシュ・コヒーレンシーの維持をサポートします。DCU の仕様は以下のとおりです。
  - 32KB のサイズ
  - 8 ウェイ・セット・アソシアティブ
  - 64 バイトのラインサイズ
- Y **データ・トランслेशन・ルックアサイド・バッファ(DTLB)** — インテル® Core™ マイクロアーキテクチャーの DTLB は、2 レベルの階層を実装しています。各レベルの DTLB には複数のエントリーがあり、4KB ページまたはラージページをサポートします。内側のレベル (DTLB0) のエントリーは、ロードに使用されます。外側のレベル (DTLB1) のエントリーは、ストア操作と、DTLB0 でミスをしたロードをサポートします。すべてのエントリーは、4 ウェイ・アソシアティブです。各 DTLB のエントリーのリストを以下に示します。
  - ラージページの DTLB1: 32 エントリー
  - 4KB ページの DTLB1: 256 エントリー
  - ラージページの DTLB0: 16 エントリー
  - 4KB ページの DTLB0: 16 エントリー



ロードが DTLB0 でミスし DTLB1 でヒットした場合、2 サイクルのペナルティーが発生します。一部のディスパッチで DTLB0 が使用される場合のみ、ソフトウェアはこのペナルティーを受けます。DTLB1 および PMH へのミスに関連した遅延は通常、スマート・メモリー・アクセスの設計によりノンブロッキングです。

Y ページ・ミス・ハンドラー(PMH)

Y メモリー・オーダー・バッファー(MOB) — 以下の機能を備えています。

- ロードとストアを投機的にアウトオブオーダー発行することができます。
- リタイアしたロードとストアがリタイア時に正確なデータを得られます。
- ロードとストアがインテル® 64 命令と IA-32 アーキテクチャーのメモリーの順序付け規則に従うことを確実にします。

インテル® Core™ マイクロアーキテクチャーのメモリークラスターは、以下を利用してメモリー操作の高速化を図ります。

- Y 128 ビットのロード操作とストア操作
- Y 1 次キャッシュからのデータ・プリフェッチ
- Y 2 次キャッシュへのプリフェッチで利用されるデータ・プリフェッチ・ロジック
- Y ストア・フォワーディング
- Y メモリー・ディスアンビゲーション (明確化)
- Y 8 個のフィル・バッファー・エントリー
- Y 20 個のストア・バッファー・エントリー
- Y メモリー操作のアウトオブオーダー実行
- Y パイプライン化された所有権読み出し (RFO) 操作

ソフトウェアをメモリークラスター向けに最適化する詳細については、3.6 節の「メモリーアクセスの最適化」を参照してください。

### 2.5.4.1 ロードとストア

インテル® Core™ マイクロアーキテクチャーでは、1 サイクルあたり 1 つの 128 ビット・ロードと 1 つの 128 ビット・ストアを、それぞれ異なるメモリー・ロケーションに対して実行できます。他の命令やメモリー操作は、アウトオブオーダーでメモリー操作を実行できます。

ロードは以下を実行できます。

- Y ロードアドレスとストアアドレスの範囲が競合していないことが明らかな場合は、先行するストアの前にロードを発行することができます。
- Y 先行する分岐が解決される前に、投機実行できます。
- Y キャッシュミスにアウトオブオーダーでオーバーラップして対応します。
- Y 競合するアドレスにストアが行われないことを見込んで、先行するストアの前にロードを発行できます。

ロードは以下を実行できません。

- Y あらゆるフォルトやトラップに対し投機的に対応すること。
- Y キャッシュできないメモリータイプに投機的にアクセスすること。

フォルトの発生したロードやキャッシュできないロードが検出されると、リタイアメントまで待機した上で、プログラマーから可視な状態に更新されます。x87 ロードと浮動小数点 SIMD ロードでは、1 クロックのレイテンシーが追加されます。

メモリーへのストアは、2 つのフェーズで実行されます。

- Y **実行フェーズ** — ストア・フォワードイングに対処するためアドレスとデータを使ってストアバッファを準備します。ディスパッチ・ポート (ポート 3 および 4) が使用されます。
- Y **完了フェーズ** — ストアがリタイアし、プログラマーがメモリーを参照できるようになります。実行中のロードとキャッシュバンクが競合する場合があります。ストアのリタイアメントはメモリー・オーダー・バッファによってバックグラウンド・タスクとして処理され、データはストアバッファから 1 次キャッシュに移動されます。

### 2.5.4.2.1 次キャッシュからのデータ・プリフェッチ

インテル® Core™ マイクロアーキテクチャーでは、2 つのハードウェア・プリフェッチャーを提供して、1 次データキャッシュへのプリフェッチを行うことにより、プログラムが高速にデータをアクセスすることを可能にしています。

- Y **データ・キャッシュ・ユニット (DCU) プリフェッチャー:** このプリフェッチャーはストリーミング・プリフェッチャーとも呼ばれ、最後にロードされたデータへのユニット・ストライド・アクセスによって開始されます。プロセッサは、このアクセスをストリーミング・アルゴリズムの一部であるとみなし、次のラインを自動的にフェッチします。
- Y **命令ポインター (IP) ベースのストライド・プリフェッチャー:** このプリフェッチャーは、個々のロード命令を追尾します。ロード命令に規則的なアドレススキップがあることが検出された場合、現在のアドレスとこの間隔の合計である次のアドレスがプリフェッチされます。前方または後方へのプリフェッチが可能であり、最大で 4KB ページの半分または 2K バイトのストライドを識別できます。

データ・プリフェッチは、以下の条件が満たされた場合にロード操作により起動されます。

- Y ライトバック・メモリーからのロード。
- Y 4K バイトのページ境界内でのプリフェッチ要求である。
- Y パイプライン内にフェンスやロックがない。
- Y その他のロードミスがあまり発生していない。
- Y バスがそれほどビジー状態ではない。
- Y 連続的なストアのストリームが発生していない。

DCU プリフェッチは以下の影響を及ぼします。

- Y 大きな構造体のデータが、プログラムで使用される順序で連続して配置されている場合、パフォーマンスが向上します。
- Y アクセスパターンに局所性がなくまばらな場合、帯域幅の問題によりわずかにパフォーマンスが低下することがあります。
- Y まれに、アルゴリズムのワーキングセットがキャッシュの大部分を占めるようにチューニングされ、プログラムから要求されたキャッシュラインが不要なプリフェッチによって排出されている場合、1 次キャッシュの容量が原因でハードウェア・プリフェッチが重大なパフォーマンス低下をもたらすことがあります。

ハードウェアに依存してデータ・トラフィックを予想するハードウェア・プリフェッチャーとは異なり、ソフトウェア・プリフェッチ命令は、プログラマーに依存してキャッシュミスの頻度を予想します。ソフトウェア・プリフェッチは、データを含むキャッシュラインを要求されるレベルのキャッシュ階層に移動するヒントとして機能します。ソフトウェア制御プリフェッチは、コードのプリフェッチではなく、データのプリフェッチを対象とします。

### 2.5.4.3 データ・プリフェッチ・ロジック

データ・プリフェッチ・ロジック (DPL) は、2 次 (L2) キャッシュへの DCU の過去の要求パターンに基づいて、2 次キャッシュに対しデータのプリフェッチを行います。DPL は、DCU からのアドレスを保持するため 2 つの独立した配列を備えており、一方はアップストリーム用 (12 エントリー)、他方はダウンストリーム用 (4 エントリー) です。DPL は、エントリーごとに 1 つの 4KB ページへのアクセスを追尾します。アクセスしたページがこの配列のどちらにも存在しない場合、新たに配列のエントリーが割り当てられます。

DPL は、インクリメントする要求シーケンス (ストリームとも呼ばれる) があるかどうか DCU の読み出しを監視します。ストリームの 2 回目のアクセスを検出すると、DPL は次のキャッシュラインをプリフェッチします。例えば、DCU がキャッシュライン A および A + 1 を要求した場合、DPL は、近い将来に DCU によってキャッシュライン A + 2 が要求されると仮定します。次に DCU が A + 2 を読み出すと、DPL はキャッシュライン A + 3 をプリフェッチします。DPL は、「後方参照」ループでも同様に機能します。

DPL は、インテル® Pentium® M プロセッサで導入されました。インテル® Core™ マイクロアーキテクチャーでは、以下の機能が DPL に追加されています。

- Y ストリームがキャッシュラインをスキップする場合など、DPL がより複雑なストリームを検出できるようになりました。DPL は、L2 ルックアップごとに 2 つのプリフェッチ要求を発行できます。インテル® Core™ マイクロアーキテクチャーでは、DPL はロード要求に先立ち、最大 8 のラインを処理できます。
- Y インテル® Core™ マイクロアーキテクチャーの DPL は、バス帯域幅と要求数に応じて動的に調整を行います。DPL プリフェッチャーはバスがビジーでない場合は最大限先までプリフェッチし、バスがビジーな場合はあまり先までプリフェッチしません。
- Y DPL は、各種のアプリケーションやシステム構成に応じて調整を行います。2 つのコアのエンタリーは、別々に処理されます。

#### 2.5.4.4 ストア・フォワードリング

インテル® Core™ マイクロアーキテクチャーでは、ストアに続くロードがストアによってメモリーに書き込まれたデータを再読み込みする場合、ストア操作からロードにデータを直接転送できます。これはストア・ロード・フォワードリングと呼ばれ、ロードがメモリーを介さずにストア操作から直接データを取得できるので、サイクル数の節約になります。

ストア・ロード・フォワードリングを行うには、以下の規則に従わなければなりません。

- Y ストアは、ロードに先行する該当アドレスへの最後のストアでなければなりません。
- Y ストアされるデータは、ロードされるデータとサイズが同じか、それよりも大きくなければなりません。
- Y ロードは、キャッシュライン境界をまたがることはできません。
- Y ロードは、8 バイト境界をまたがることはできません。ただし、16 バイト・ロードは例外です。
- Y ロードは、以下の例外を除いて、ストアアドレスの先頭にアライメントされていなければなりません。
  - アライメントされた 64 ビット・ストアでは、その半分である 32 ビットのいずれにも転送できます。
  - アライメントされた 128 ビット・ストアでは、その 4 分の 1 である 32 ビットのいずれにも転送できます。
  - アライメントされた 128 ビット・ストアでは、その半分である 64 ビットのいずれにも転送できます。

ソフトウェアは、最後の規則に対する例外を適用して、サブフィールドを転送する能力を犠牲にせず複雑な構造を移動できます。

拡張版インテル® Core™ マイクロアーキテクチャーでは、ストア・フォワードリングに関するアライメントの制限が緩和されています。拡張版インテル® Core™ マイクロアーキテクチャーは、後続のロードのアライメントが先行するストアに合っていない複数の状況でストア・フォワードリングを実行できます。拡張版インテル® Core™ マイクロアーキテクチャーでは許可されていますが、インテル® Core™ マイクロアーキテクチャーでは許可されていない 6 つのケース (背景がグラデーシヨンのもの) を図 2-10 に示します。背景が網かけのものは、両者で実行可能なストア・フォワードリングです。

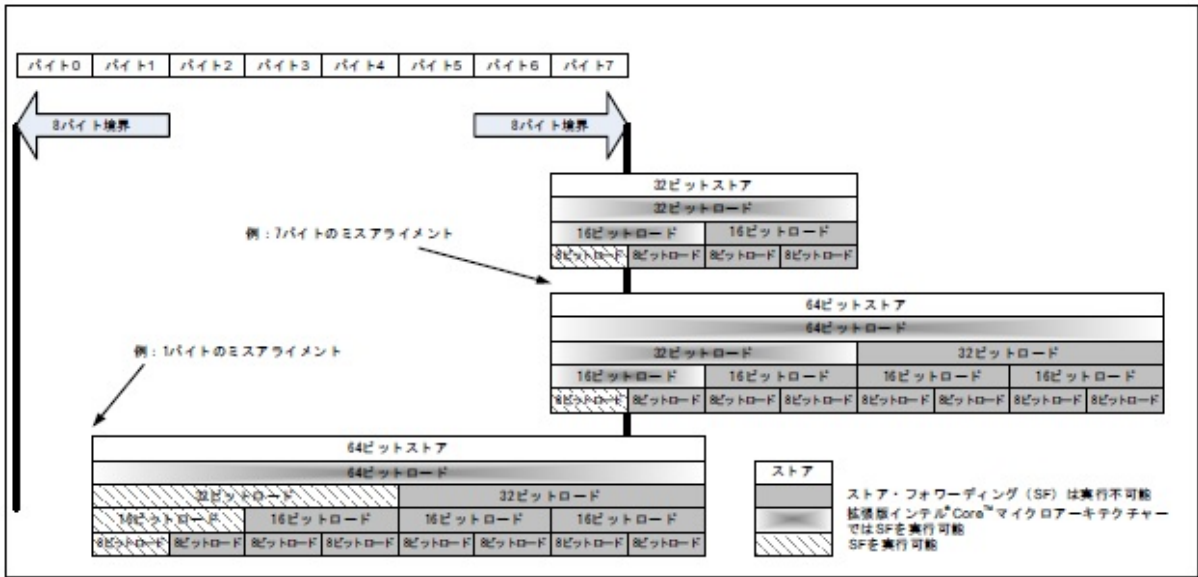


図 2-10 拡張版インテル® Core™ マイクロアーキテクチャーにおけるストア・フォワーディングの強化

### 2.5.4.5 メモリー・ディスアンビゲーション

ロード命令のマイクロオペレーション ( $\mu\text{op}$ ) は、先行するストアに依存する可能性があります。多くのマイクロアーキテクチャーでは、先行するストアアドレスがすべて判明するまで、ロードはブロックされます。

メモリー・ディスアンビゲーション機能は、どのロードが先行するストアに依存しないかを予測します。ロードにそのような依存関係がないことをディスアンビゲーション機能が予測した場合、たとえストアアドレスが判明していなくても、そのロードは 1 次データキャッシュからデータを取得します。

最終的に予測が検証され、実際に競合が検出されると、ロードとすべての後続命令が再実行されます。メモリー・ディスアンビゲーションは、メモリーアクセスの明確化とも呼ばれます。

## 2.5.5 インテル® アドバンスト・スマート・キャッシュ

インテル® Core™ マイクロアーキテクチャーでは、多くの機能が単一ダイ上の 2 つのプロセッサ・コアに最適化されています。2 つのコアは、インテル® アドバンスト・スマート・キャッシュと呼ばれる 2 次キャッシュおよびバス・インターフェイス・ユニットを共有します。ここではインテル® アドバンスト・スマート・キャッシュの構成要素について説明します。図 2-11 に、インテル® アドバンスト・スマート・キャッシュのアーキテクチャーを示します。

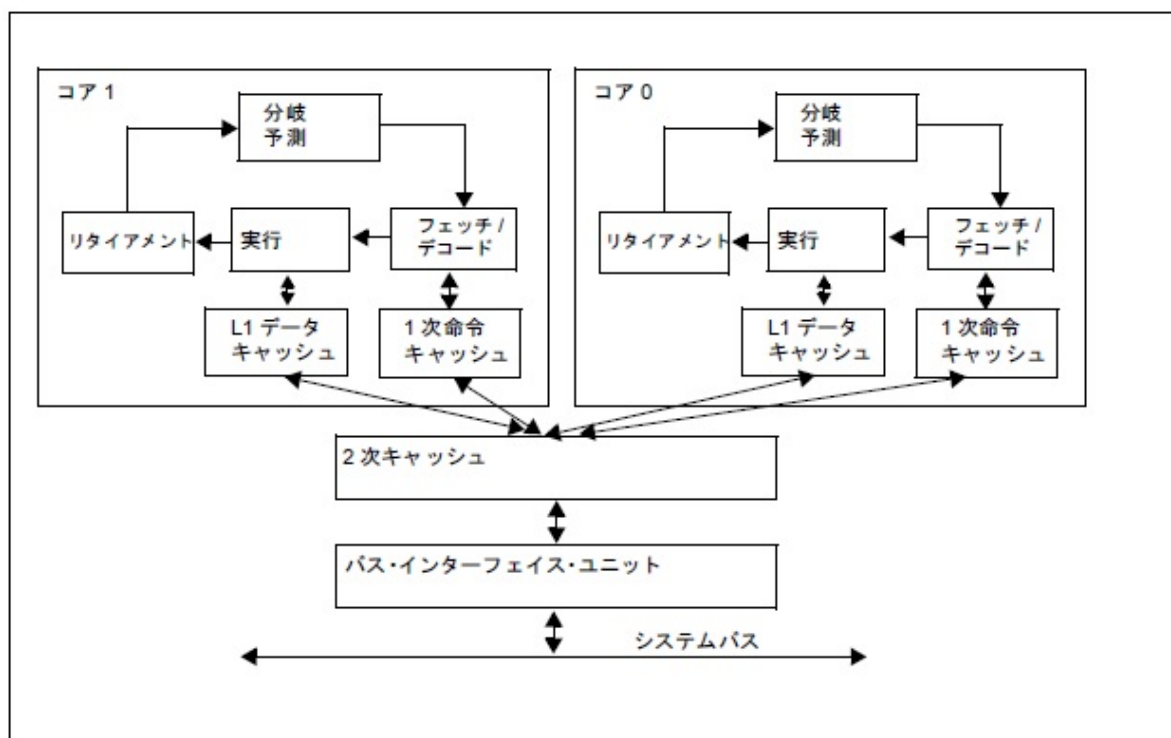


図 2-11 インテル® アドバンスト・スマート・キャッシュのアーキテクチャー

表 2-28 に、インテル® Core™ マイクロアーキテクチャーにおけるキャッシュのパラメーターの詳細を示します。CPUID 命令のキャッシュ・パラメーターを使用してキャッシュ階層 ID を列挙する際の詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』を参照してください。

表 2-28 インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサのキャッシュ・パラメーター

レベル	容量	アソシアティブ (ウェイ)	ラインサイズ (バイト)	アクセス・レイテンシー (クロック)	アクセス・スループット (クロック)	書き込みアップデート方式
1 次	32 KB	8	64	3	1	ライトバック
命令	32 KB	8	なし	なし	なし	なし
2 次 (共有 L2) <sup>1</sup>	2, 4 MB	8 または 16	64	14 <sup>2</sup>	2	ライトバック
2 次 (共有 L2) <sup>3</sup>	3, 6MB	12 または 24	64	15 <sup>2</sup>	2	ライトバック
3 次 <sup>4</sup>	8, 12, 16 MB	16	64	110	12	ライトバック

**注意:**

1. インテル® Core™ マイクロアーキテクチャー (CPUID シグネチャー DisplayFamily = 06H, DisplayModel = 0FH)。
2. ソフトウェアから見えるレイテンシーは、アクセスパターンとその他の要因に依存するため異なります。

3. 拡張版インテル® Core™ マイクロアーキテクチャー (CUID シグネチャー DisplayFamily = 06H, DisplayModel = 17H または 1DH),
4. 拡張版インテル® Core™ マイクロアーキテクチャー (CUID シグネチャー DisplayFamily = 06H, DisplayModel = 1DH),

### 2.5.5.1 ロード

ライトバック (WB) 方式のメモリー・ロケーションから命令がデータを読み出す場合、プロセッサはそのデータが含まれるキャッシュラインを以下の順序でキャッシュとメモリーから検索します。

1. 自コアの DCU
2. 他コアの DCU と 2 次キャッシュ
3. システムメモリー

キャッシュラインは、変更された場合のみ他方のコアの DCU から取得され、キャッシュラインの可用性や 2 次キャッシュの状態は無視されます。

表 2-29 に、局所性が異なる最初の 4 バイトをメモリー・クラスターからフェッチする際の特性を示します。レイテンシーの列には、アクセス・レイテンシーの概算値が記載されています。ただし、実際のレイテンシーは、キャッシュのロード、メモリー・コンポーネント、そのパラメーターによって異なります。

表 2-29 インテル® Core™ マイクロアーキテクチャーでのロード操作とストア操作の特性

データ局所性	ロード		ストア	
	レイテンシー	スループット	レイテンシー	スループット
DCU	3	1	2	1
変更状態にある他方のコアの DCU	14 + 5.5 バスサイクル	14 + 5.5 バスサイクル	14 + 5.5 バスサイクル	
2 次キャッシュ	14	3	14	3
メモリー	14 + 5.5 バスサイクル + メモリー	バス読み出しプロトコルによって異なります	14 + 5.5 バスサイクル + メモリー	バス書き込みプロトコルによって異なります

変更されたキャッシュラインを排出して、新しいキャッシュライン用にスペースを確保しなければならないことがあります。変更されたキャッシュラインは新しいデータの取得と並行して排出されるので、レイテンシーは追加されません。ただし、データがメモリーにライトバックされる際は、キャッシュ帯域幅のほか、バス帯域幅も排出に使用される可能性があります。したがって、変更されたキャッシュラインの排出を伴うキャッシュミスが短時間に複数発生した場合、キャッシュ応答時間が全体的に低下します。

### 2.5.5.2 ストア

ライトバック方式のメモリー・ロケーションに命令がデータを書き込む際、プロセッサはまず、自身の DCU においてキャッシュラインが排他状態 (Exclusive) または変更状態 (Modified) であることを確認します。プロセッサは、指定された順序で以下の場所からキャッシュラインを検索します。

1. 自コアの DCU
2. 他コアの DCU と 2 次キャッシュ
3. システムメモリー

キャッシュラインは、変更された場合のみ他方のコアの DCU から取得され、キャッシュラインの可用性や 2 次キャッシュの状態は無視されます。所有権読み出しが完了した後、データが 1 次データキャッシュに書き込まれ、キャッシュラインが変更済みとしてマークされます。

所有権読み出しとデータのストアは、命令のリタイアメントの後に発生し、リタイアメントの順序に従います。したがって、ストア・レイテンシーは、通常、ストア命令自体には影響を与えることはありません。ただし、一部の連続したストアは、レイテンシーを累積してパフォーマンスに影響を与えることがあります。表 2-28 に、キャッシュラインの階層に

よって異なるストア・レイテンシーを示します。

## 2.6 インテル® マイクロアーキテクチャー開発コード名 Nehalem

インテル® マイクロアーキテクチャー開発コード名 Nehalem は、インテル® Core™ i7 プロセッサおよびインテル® Xeon® プロセッサ 3400/5500/7500 番台が持つ数多くの革新的な機能の基盤となっています。このアーキテクチャーは、45nm の拡張版インテル® Core™ マイクロアーキテクチャーの成功を受けて開発され、以下の拡張機能を提供しています。

### Y 強化されたプロセッサ・コア

- 分岐予測と予測ミスからの回復を改善
- ループ・ストリーミングの強化により、フロントエンドのパフォーマンスを高め、消費電力を軽減
- アウトオブオーダー・エンジンでのバッファリングを深くすることにより、並列性を向上
- 実行ユニットの強化により、CRC、文字列/テキスト処理、データ・シャッフリングを高速化

### Y ハイパースレディング・テクノロジー

- 1 コアあたり 2 つのハードウェア・スレッド (論理プロセッサ) を提供
- 4 マイクロオペレーション (μop) の発行が可能な実行エンジン、大容量 L3 キャッシュ、広いメモリー帯域幅の利点を活用

### Y インテル® スマート・メモリー・アクセス

- システムメモリーへの低レイテンシー・アクセスとスケラブルなメモリー帯域幅を実現する統合メモリー・コントローラー
- インクルーシブ (包括的) な共有 L3 キャッシュを持つ新しいキャッシュ階層構造によって、スヌープ・トラフィックを軽減
- 2 レベルの TLB と、TLB サイズを拡大
- アライメントされないメモリーの高速アクセス

### Y 革新的な専用パワー・マネジメント機能

- 最適化された組込みファームウェアと統合マイクロコントローラーによって消費電力を管理
- 温度、電流、消費電力を測定する組込みリアルタイム・センサー
- コアごとに消費電力をオン/オフできる統合パワーゲート
- メモリーやリンク・サブシステムの消費電力を削減する汎用性

インテル® マイクロアーキテクチャー開発コード名 Westmere は、インテル® マイクロアーキテクチャー開発コード名 Nehalem の 32nm バージョンです。後者の機能はすべて前者にも適用されます。

### 2.6.1 マイクロアーキテクチャー・パイプライン

インテル® マイクロアーキテクチャー開発コード名 Nehalem は、65nm のインテル® Core™ マイクロアーキテクチャーで導入された、4 マイクロオペレーション (μop) の発行が可能なマイクロアーキテクチャー・パイプラインを継承しています。図 2-12 に、インテル® Core™ i7 プロセッサで実装されているインテル® マイクロアーキテクチャー開発コード名 Nehalem のパイプラインの基本構成要素を示します。ただし、図 2-12 のパイプラインでは、4 つのコアのうち 2 つのみが描かれています。

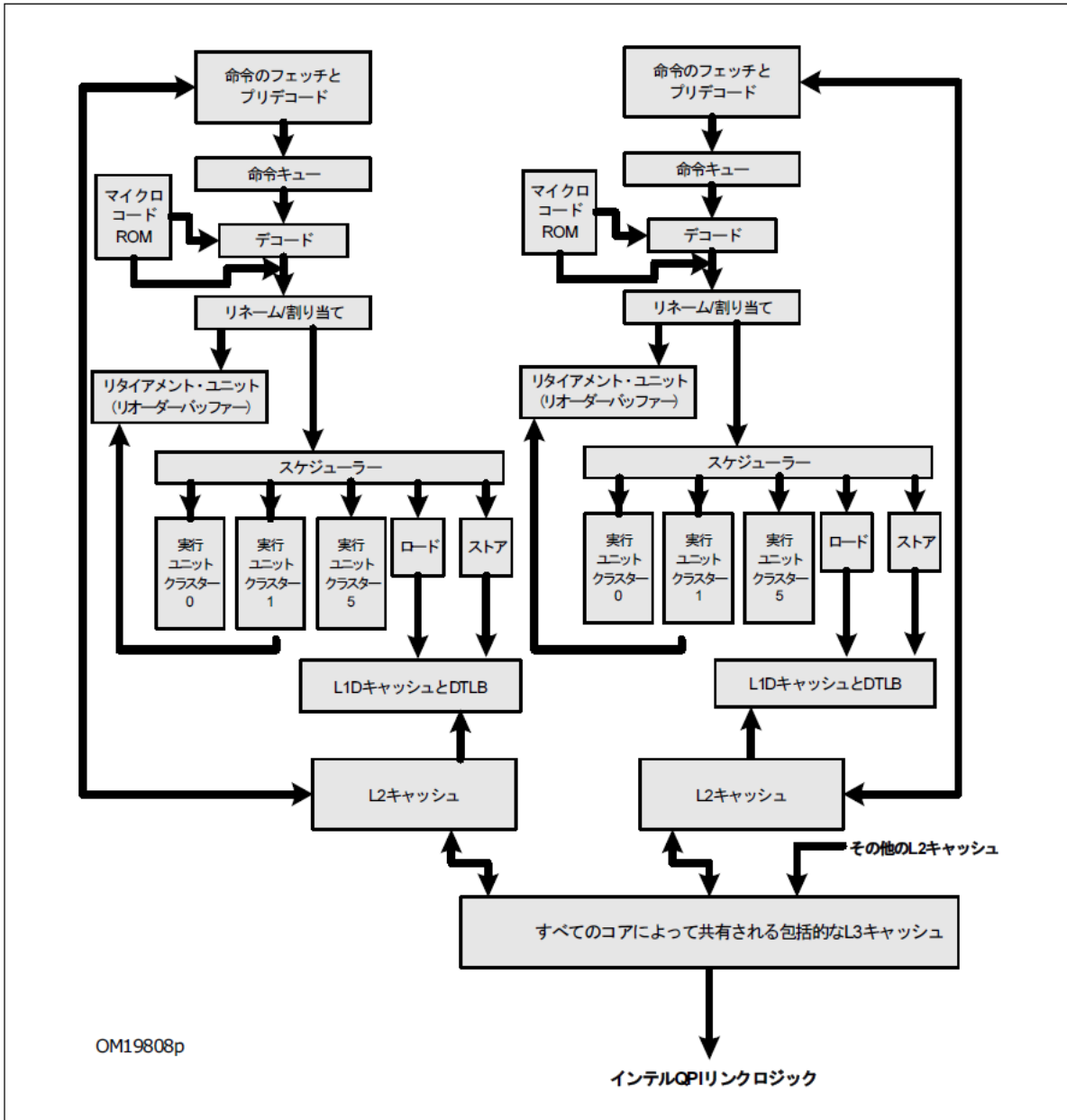


図 2-12 インテル® マイクロアーキテクチャー開発コード名 Nehalem のパイプラインの構造

インテル® マイクロアーキテクチャー開発コード名 Nehalem のパイプライン長は、分岐予測ミスの遅延を測定した場合、45nm のインテル® Core™2 プロセッサ・ファミリーにおける従来のパイプラインよりも 2 サイクルだけ長くなっています。フロントエンドは、1 サイクルあたり最大 4 命令をデコード可能であり、2 つの論理プロセッサが交互のサイクルで命令ストリームをデコードすることによって、2 つのハードウェア・スレッドをサポートします。フロントエンドには、分岐処理、ループ検出、MSROM スルーポイントなどを扱う拡張機能が備わっています。

スケジューラー (リザベーション・ステーション) は、1 サイクルあたり最大 6 マイクロオペレーション (μop) を 6 つの発行ポートにディスパッチできます (図 2-12 では 5 つの発行ポートのみが表示されています。ストア操作ではストアアドレスとストアデータに個別のポートが必要ですが、図では 1 つとして描かれています)。

アウトオブオーダー・エンジンには多くの実行ユニットがあり、各ユニットは図 2-12 に示す 3 つの実行クラスターに配置されています。アウトオブオーダー・エンジンは、従来と同様に、1 サイクルあたり 4 マイクロオペレーション (μop) をリタイアできます。



## 2.6.2 フロントエンドの概要

図 2-13 に、このマイクロアーキテクチャーのフロントエンドの主な構成要素を示します。命令フェッチユニット (IFU) は、1 サイクルあたり最大 16 バイトのアライメントされた命令バイトを、命令キャッシュから命令長デコーダー (ILD) にフェッチします。命令キュー (IQ) は、ILD で処理された命令をバッファリングして、1 サイクルあたり最大 4 命令を命令デコーダーに供給できます。

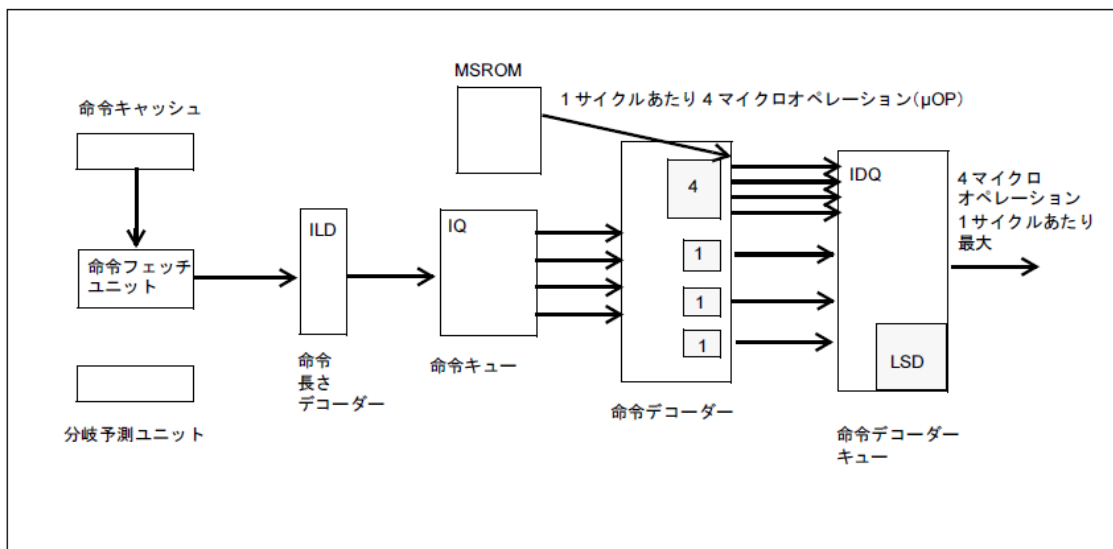


図 2-13 インテル® マイクロアーキテクチャー開発コード名 Nehalem のフロントエンド

命令デコーダーは、1 サイクルあたり 1 つの単純な命令をデコードできる 3 つのデコーダーユニットを備えています。もう 1 つのデコーダーユニットは、単純な命令でも、複数のマイクロオペレーション (μop) で構成された複雑な命令でも、1 サイクルあたり 1 命令をデコードできます。4 つを超えるマイクロオペレーション (μop) で構成された命令は、MSROM から供給されます。1 サイクルあたり最大 4 マイクロオペレーション (μop) を命令デコーダーキュー (IDQ) に供給できます。

IDQ 内のループストリーム検出器は、短い命令シーケンスからなるループの消費電力を削減してフロントエンドの効率化を図ります。

命令デコーダーはマイクロフュージョンをサポートすることで、フロントエンドのスループットを高め、スケジューラーおよびリオーダーバッファー (ROB) 内の有効キューサイズを拡大します。マイクロフュージョンの規則は、インテル® Core™ マイクロアーキテクチャーと同様です。

命令キューはマクロフュージョンもサポートしており、可能な限り、隣接する命令を単一のマイクロオペレーション (μop) に組み合わせます。前世代のインテル® Core™ マイクロアーキテクチャーでは、CMP/Jcc シーケンスでのマクロフュージョンのサポートは CF および ZF フラグに限定されており、64 ビット・モードではマクロフュージョンがサポートされていませんでした。

インテル® マイクロアーキテクチャー開発コード名 Nehalem では、64 ビット・モードでもマクロフュージョンがサポートされるようになり、以下の命令シーケンスがサポートされています。

Y 以下を比較する場合に CMP または TEST をフュージョンできます (変更なし)。

レジスター-レジスター。次に例を示します: CMP EAX,ECX; JZ label

レジスター-即値。次に例を示します: CMP EAX,0x80; JZ label

レジスター-メモリー。次に例を示します: CMP EAX,[ECX]; JZ label

メモリー-レジスター。次に例を示します: CMP [EAX],ECX; JZ label

Y TEST はすべての条件分岐とフュージョンされます (変更なし)。

- Y CMP は以下の条件分岐とフュージョンできます。これらの条件分岐では、キャリーフラグ (CF) またはゼロフラグ (ZF) をチェックします。以下は、マクロフュージョン可能な条件分岐のリストです (変更なし)。
  - JA または JNBE
  - JAE または JNB または JNC
  - JE または JZ
  - JNA または JBE
  - JNAE または JC または JB
  - JNE または JNZ
  
- Y インテル® マイクロアーキテクチャー開発コード名 Nehalem では、CMP は以下の条件分岐とフュージョンできます (拡張機能)。
  - JL または JNGE
  - JGE または JNL
  - JLE または JNG
  - JG または JNLE

ハードウェアは、いくつかの方法によって分岐処理を改善しています。分岐ターゲットバッファが拡大し、分岐予測の精度が向上しました。リターン・スタック・バッファでのリネームがサポートされ、コード中のリターン命令の予測ミスが減少しました。さらに、ハードウェアの強化によりリソースの再利用を促進することで、分岐予測ミスの処理が向上しました。これにより、予測ミスしたコードパスの実行にリソースが割り当てられていても、フロントエンドは構成済みのコードパス (リタイアメントに到達すると予測されるコードパス) 中の命令デコードを待機しなくて済みます。構成済みのコードパスの命令をフロントエンドがデコードすると、すぐに新しいマイクロオペレーション ( $\mu\text{op}$ ) ストリームが順方向で処理を開始できます。

### 2.6.3 実行エンジン

IDQ (図 2-13) は、マイクロオペレーション ( $\mu\text{op}$ ) ストリームをパイプラインの割り当て/リネームステージ (図 2-12) に供給します。アウトオブオーダー・エンジンは、インフライト (パイプライン中で同時に進行する) のマイクロオペレーション ( $\mu\text{op}$ ) を最大 128 個サポートします。各マイクロオペレーション ( $\mu\text{op}$ ) は、リオーダーバッファ (ROB) 内のエンタリー、リザベーション・ステーション (RS) 内のエンタリー、およびロード/ストアバッファ (メモリアクセスが必要な場合) などのリソースとともに割り当てられる必要があります。

アロケータは、インフライトの各マイクロオペレーション ( $\mu\text{op}$ ) のレジスター・ファイル・エンタリーをリネームします。マイクロオペレーション ( $\mu\text{op}$ ) に関連する入力データは、通常 ROB またはリタイアしたレジスターファイルから読み出されます。

RS の深さが 36 エンタリーに拡大されました (前世代では 32 エンタリー)。マイクロオペレーション ( $\mu\text{op}$ ) の実行準備が整っている場合、RS は 1 サイクルあたり最大 6  $\mu\text{op}$  をディスパッチできます。RS は、発行ポートを通じてマイクロオペレーション ( $\mu\text{op}$ ) を個々の実行クラスターにディスパッチします。各クラスターは、整数/FP/SIMD 実行ユニットの集合で構成されます。

マイクロオペレーション ( $\mu\text{op}$ ) を実行した実行ユニットの結果は、レジスターファイルにライトバックされるか、結果を必要とするインフライトの  $\mu\text{op}$  にバイパス・ネットワーク経由で転送されます。マイクロアーキテクチャー開発コード名 Nehalem は、各ポートで 1 サイクルごとに 1 つのレジスターファイルに書き込むライトバック・スループットに対応しています。バイパス・ネットワークは、整数/FP/SIMD の 3 ドメインで構成されます。同じバイパスドメイン内で結果を生産 (producer) するマイクロオペレーション ( $\mu\text{op}$ ) から消費 (consumer) する  $\mu\text{op}$  にハードウェア上遅延なく効率良く転送できます。異なるバイパスドメイン間での結果の転送は、バイパス遅延が増加する可能性があります。バイパス遅延は、個々の実行ユニットのレイテンシーとスループットに加え、ソフトウェアにも表れる場合があります。表 2-30 に、異なるバイパスドメインにおける生産マイクロオペレーション ( $\mu\text{op}$ ) と消費  $\mu\text{op}$  の間のバイパス遅延を示します。

表 2-30 生産側と消費側のマイクロ ops 間のパイパス遅延 (サイクル)

	FP	整数	SIMD
FP	0	2	2
整数	2	0	1
SIMD	2	1	0

### 2.6.3.1 発行ポートと実行ユニット

表 2-31 に、マイクロアーキテクチャーで一般的な操作に関する、発行ポートおよび実行ユニットのレイテンシー/スループットの特性を示します。

表 2-31 インテル® マイクロアーキテクチャー開発コード名 Nehalem の発行ポート

ポート	実行可能な操作	レイテンシー	スループット	ドメイン	説明
ポート 0	整数 ALU	1	1	整数	
	整数シフト	1	1		
ポート 0	整数 SIMD ALU	1	1	SIMD	
	整数 SIMD シャッフ ル	1	1		
ポート 0	単精度 (SP) FP MUL	4	1	FP	
	倍精度 FP MUL	5	1		
	FP MUL (X87)	5	1		
	FP/SIMD/SSE2 ムー ブと論理演算	1	1		
	FP シャッフ DIV/SQRT	1	1		
ポート 1	整数 ALU	1	1	整数	
	整数 LEA	1	1		
	整数 MUL	3	1		
ポート 1	整数 SIMD MUL	1	1	SIMD	
	整数 SIMD シフト	1	1		
	PSAD	3	1		
	文字列比較				
ポート 1	FP ADD	3	1	FP	
ポート 2	整数ロード	4	1	整数	

ポート	実行可能な操作	レイテンシー	スループット	ドメイン	説明
ポート 3	ストアアドレス	5	1	整数	
ポート 4	ストアデータ			整数	
ポート 5	整数 ALU	1	1	整数	
	整数シフト	1	1		
	ジャンプ	1	1		
ポート 5	整数 SIMD ALU	1	1	SIMD	
	整数 SIMD シャッフ ル	1	1		
ポート 5	FP/SIMD/SSE2 ムー ブと論理演算	1	1	FP	

## 2.6.4 キャッシュとメモリー・サブシステム

インテル® マイクロアーキテクチャー開発コード名 Nehalem では、各コアに命令キャッシュ、L1 データキャッシュ、ユニファイド L2 キャッシュが搭載されています (図 2-12 を参照)。それぞれの物理プロセッサは、複数のプロセッサ・コアと、「アンコア」と呼ばれる共有サブシステムで構成されています。具体的には、インテル® Core™ i7 プロセッサのアンコアは、物理プロセッサ内のすべてのコアで共有されるユニファイド L3 キャッシュ、インテル® QuickPath インターコネクト・リンク、および関連ロジックを含みます。L1 キャッシュと L2 キャッシュはライトバックで、かつインクルーシブではありません。

共有 L3 キャッシュはライトバックで、かつインクルーシブです。つまり、L1 データキャッシュ、L1 命令キャッシュ、ユニファイド L2 キャッシュのいずれかに存在するキャッシュラインは、L3 キャッシュにも存在します。L3 キャッシュはインクルーシブである特性を利用して、プロセッサ・コア間のスヌープ・トラフィックを最小限に抑えるように設計されています。表 2-32 に、キャッシュ階層の特性を示します。L3 キャッシュアクセスのレイテンシーは、プロセッサとアンコア・サブシステムとの周波数比に応じて異なります。

表 2-32 インテル® Core™ i7 プロセッサのキャッシュ・パラメーター

レベル	容量	アソシアティブ (ウェイ)	ライン サイズ (バイト)	アクセス・ レイテンシー (クロック)	アクセス・ スループット (クロック)	書き込みアップ デート方式
L1 データ	32KB	8	64	4	1	ライトバック
命令	32KB	4	なし	なし	なし	なし
L2	256KB	8	64	10 <sup>1</sup>	状況により異なる	ライトバック
L3 (共有) <sup>2</sup>	8MB	16	64	35-40+ <sup>2</sup>	状況により異なる	ライトバック

### 注:

1. ソフトウェアから見えるレイテンシーは、アクセスパターンなどの要因によって異なります。
2. コアとアンコアとの周波数比が 1:1 である場合、L3 キャッシュの最小レイテンシーは 35 サイクルです。

インテル® マイクロアーキテクチャー開発コード名 Nehalem は、2 レベルのトランスレーション・ルックアサイド・バッファ (TLB) を実装しています。第 1 レベルは、データおよびコード向けの個別の TLB を構成します。DTLB0 はデータアクセスのアドレス変換を処理し、4KB ページをサポートする 64 個のエントリーと、ラージページをサポートする 32 個のエントリーを備えています。ITLB は、4KB ページ向けに 1 スレッドあたり 64 個のエントリーと、ラージページ向けに 1 スレッドあたり 7 個のエントリーを備えます。

第 2 レベルの TLB (STLB) は、4KB ページのコードアクセスとデータアクセスの両方を処理します。これは、DTLB0 や ITLB でミスをした 4KB ページの変換操作をサポートします。すべてのエントリーは、4 ウェイ・アソシアティブです。各 DTLB のエントリーのリストを以下に示します。

- Y 4KB ページの STLB:512 エントリー (データ・ルックアップと命令ルックアップの両方を処理)
- Y ラージページの DTLB0:32 エントリー
- Y 4KB ページの DTLB0:64 エントリー

DTLB0 でミスし STLB でヒットした場合、7 サイクルのペナルティーが発生します。一部のディスパッチで DTLB0 が使用される場合のみ、ソフトウェアはこのペナルティーを受けます。STLB および PMH へのミスに関連した遅延は通常大きくノンブロッキングです。

## 2.6.5 ロード操作とストア操作の強化

インテル® マイクロアーキテクチャー開発コード名 Nehalem のメモリークラスターは、以下の拡張機能によってメモリー操作の高速化を図ります。

- Y 1 サイクルあたり 1 つの 128 ビット・ロード操作と 1 つの 128 ビット・ストア操作のピーク発行率
- Y ロード操作とストア操作向けの深いバッファ: 48 個のロードバッファ、32 個のストアバッファ、10 個のフィルバッファ
- Y アライメントされていないメモリーの高速度アクセスと、堅牢なメモリー・アライメント・ハザード処理
- Y ストア・フォワーディングの改善により、アライメントされたシナリオにもアライメントされないシナリオにも対応
- Y ほとんどのアドレス・アライメントに対応したストア・フォワーディング

### 2.6.5.1 効率的なアライメント・ハザードの処理

あらゆるワークロードにおいて、キャッシュ・サブシステムとメモリー・サブシステムが大部分の命令を処理しています。アドレス・アライメントのシナリオが異なると、メモリー操作やキャッシュ操作におけるパフォーマンスへの影響も変化します。例えば、L1 キャッシュからの、自然境界にアライメントされたロードには通常、L1 キャッシュでの 1 サイクルのスループット (表 2-33 を参照) が適用されます。しかし、アライメントしないロード命令 (MOVUPS、MOVUPD、MOVDQU など) によって L1 キャッシュのデータにアクセスすると、マイクロアーキテクチャーやアライメントのシナリオに応じて遅延は異なります。

表 2-33 L1 キャッシュからの MOVDQU におけるアドレス・アライメントがパフォーマンスに与える影響

スループット (サイクル)	インテル® Core™ i7 プロセッサ	45nm のインテル® Core™ マイクロアーキ テクチャー	65nm のインテル® Core™ マイクロアーキ テクチャー
アライメント・シナリオ	06_1AH	06_17H	06_0FH
16 バイト境界にアライメント	1	2	2
16 バイト境界にアライメントされず、キャッシュ分割なし	1	~2	~2
キャッシュライン境界が分割	~4.5	~20	~20

表 2-33 に、各種のアドレス・アライメント・シナリオで MOVDQU 命令を発行し L1 キャッシュからデータをロードした場合のおよそのスループットを示します。16 バイト・ロードがキャッシュライン境界をまたいだ場合、前世代のマイクロアーキテクチャーでは、ソフトウェアから見ても大きな遅延が発生しました。

インテル® マイクロアーキテクチャー開発コード名 Nehalem では、ハードウェアの強化により、キャッシュライン分割など各種アドレス・アライメント・シナリオを処理する際の遅延が短縮されています。

### 2.6.5.2 ストア・フォワーディングの強化

マイクロアーキテクチャーでは、ストアに続くロードがストアによってメモリーに書き込まれたデータを再読み込みする場合、ストア操作からロードにデータを直接転送できます。これはストア・ロード・フォワーディングと呼ばれ、ロードがメモリーを介さずにストア操作から直接データを取得できるので、サイクル数の節約になります。

遅延なくストア・ロード・フォワーディングを行うには、以下の規則に従わなければなりません。

- Y ストアは、ロードに先行する該当アドレスへの最後のストアでなければなりません
- Y ストアされるデータは、ロードされるデータとサイズが同じか、それよりも大きくなければなりません
- Y ロードデータは、先行するストア内に完全に含まれていなければなりません

ストア操作とロード操作間のアドレス・アライメントとデータサイズによって、ストア・フォワーディングでデータが転送されるか、キャッシュ/メモリー・サブシステムによる遅延が生じるかが決まります。45nm の拡張版インテル® Core™ マイクロアーキテクチャーでは、アドレス・アライメントとデータサイズの要件が従来のマイクロアーキテクチャーよりも緩和されました。インテル® マイクロアーキテクチャー開発コード名 Nehalem ではさらに機能が強化され、さらに多くの状況で迅速にデータ転送できます。

図 2-14 に、16 バイト・ストア操作におけるストア・フォワーディング・シナリオを示します。

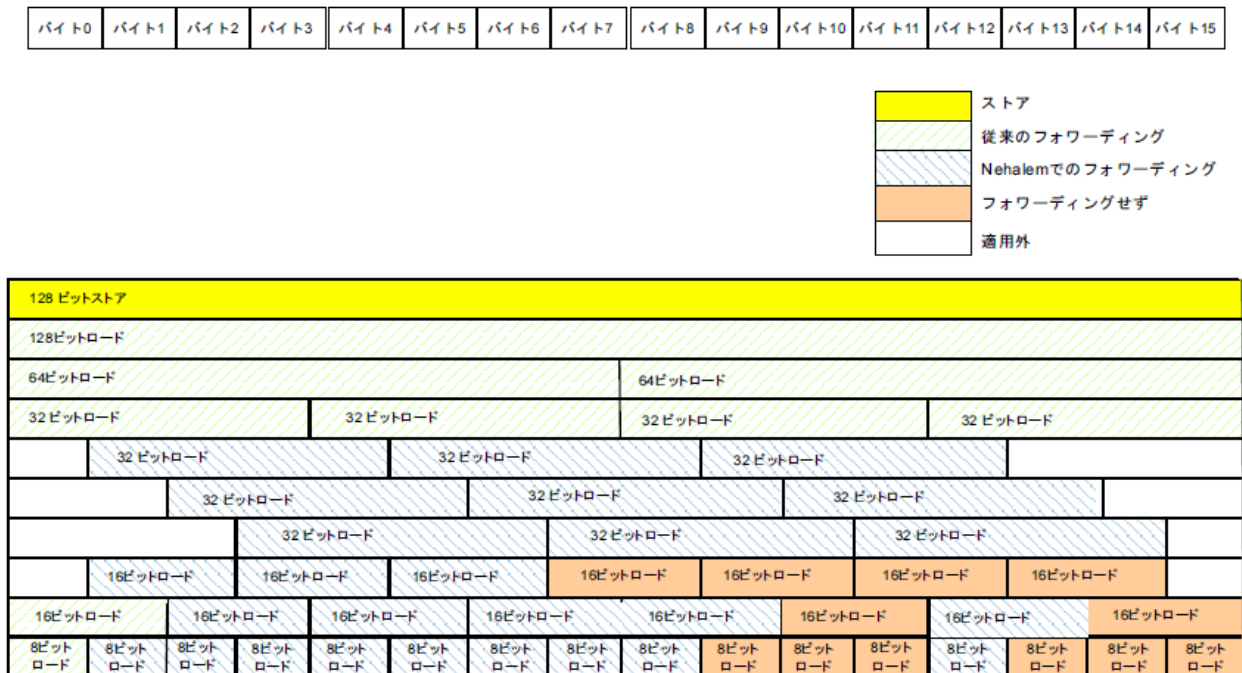


図 2-14 16 バイト・ストア操作のストア・フォワーディング・シナリオ

インテル® マイクロアーキテクチャー開発コード名 Nehalem では、ストアアドレスのアライメントにかかわらずストア・ロード・フォワーディングを実行できます (図中の空欄は該当ストア・ロードシナリオが適用されません)。図 2-15 に、8 バイト以下のストア操作のシナリオを示します。

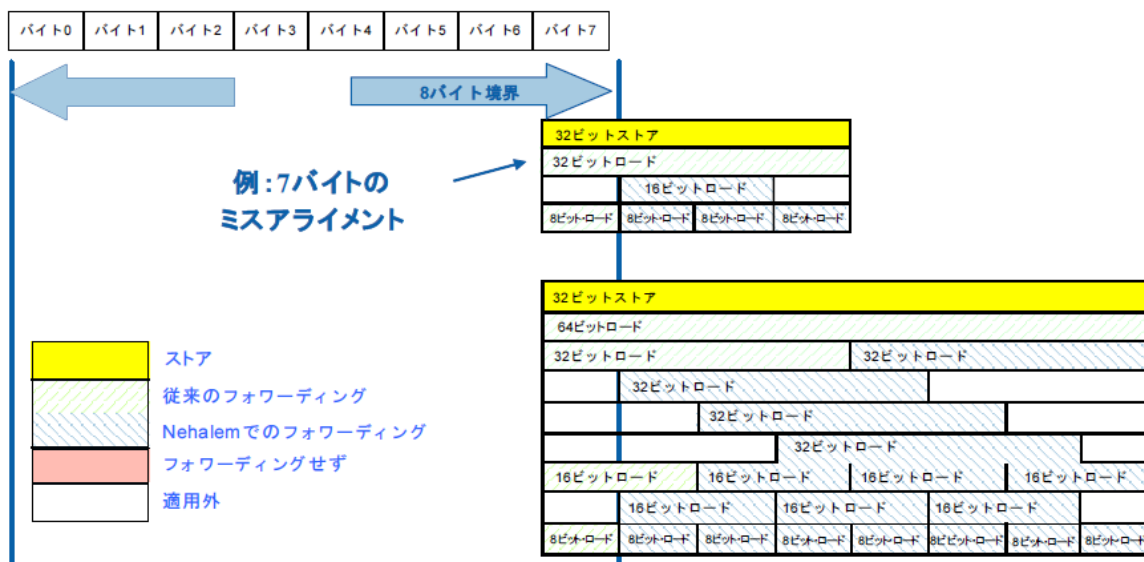


図 2-15 インテル® マイクロアーキテクチャー開発コード名 Nehalem におけるストア・フォワーディングの強化

## 2.6.6 REP (リピート) 文字列の強化

memcpy/memset などのライブラリー関数の実装では、REP プリフィクスを MOVSB/STOS 命令および ECX のカウント値と組み合わせることが頻繁に行われます。これは、「REP 文字列 (文字列リピート)」命令と呼ばれます。この命令では、反復ごとに一定の値をバイト/ワード/ダブルワード/クワッドワード単位でコピー/書き込みできます。文字列 REP を使用する上でのパフォーマンスは、開始オーバーヘッドとデータ転送スループットという 2 つの要素によって決まります。

REP 文字列命令のパフォーマンスを決める 2 つの要素は、さらに単位、アライメント、カウント値によって異なります。MOVSB は通常、極めて小さなデータチャンクの処理に使用されます。したがって、REP MOVSB を実装したプロセッサは、ECX < 4 の処理に最適化されており、ECX > 3 の REP MOVSB を使用すると、バイト単位のデータ転送に加えて開始オーバーヘッドの増加により、データ・スループットが低下します。ECX < 4 の場合、MOVSB のレイテンシーは 9 サイクルですが、ECX > 9 の REP MOVSB では、50 サイクルの開始コストが発生します。

さらに大きな単位でデータ転送が行われる REP 文字列の場合、ECX 値の増加に伴い、REP 文字列の開始オーバーヘッドは以下のようにステップ単位で増加します。

- Y 短い文字列 (ECX ≤ 12): REP MOVSB/MOVSQ のレイテンシーは約 20 サイクルです。
- Y 高速文字列 (ECX ≥ 76: REP MOVSB を除く): プロセッサは、できるだけ多くのデータを 16 バイト単位で移動することによりハードウェアの最適化を行います。REP 文字列のレイテンシーは、16 バイト・データ転送のいずれかがキャッシュライン境界をまたいでいるかどうかによって異なります。
  - 分割なし: レイテンシーは約 40 サイクルの開始コストからなり、64 バイトのデータごとに 4 サイクルが追加されます。
  - キャッシュが分割: レイテンシーは約 35 サイクルの開始コストからなり、64 バイトのデータごとに 6 サイクルが追加されます。
- Y 中間の文字列長: REP MOVSB/MOVSQ のレイテンシーは約 15 サイクルの開始コストからなり、ワード/ダブルワード/クワッドワード単位でのデータ移動の反復ごとに 1 サイクルが追加されます。

インテル® マイクロアーキテクチャー開発コード名 Nehalem では、REP 文字列のパフォーマンスは以下の面で従来のマイクロアーキテクチャーに比べ大幅に向上しています。

- Y ほとんどのケースで開始オーバーヘッドが従来のマイクロアーキテクチャーに比べて削減されました
- Y データ転送スループットが前世代よりも向上しました

- Y 従来のマイクロアーキテクチャーの場合、REP 文字列命令を「高速文字列」モードで動作させるには、アドレス・アライメントが必要でしたが、インテル® マイクロアーキテクチャー開発コード名 Nehalem では、アドレスが 16 バイト境界にアライメントされていなくても、REP 文字列命令を「高速文字列」モードで動作できます

## 2.6.7 システム・ソフトウェアの強化

インテル® マイクロアーキテクチャー開発コード名 Nehalem では、アプリケーション・レベル・ソフトウェアとシステム・レベル・ソフトウェアの両方に利点があるマイクロアーキテクチャーの強化に加えて、主にシステム・ソフトウェアに適した強化も行われています。

Lock プリミティブ: Lock プリフィクス (XCHG, CMPXCHG8B など) を使用して同期プリミティブを実行すると、従来のマイクロアーキテクチャーよりもレイテンシーが大幅に減少します。

VMM のオーバーヘッド改善: 従来のマイクロアーキテクチャーでは、仮想マシン (VM) とそのスーパーバイザー (VMM) との間の VMX 遷移は、一度に数千サイクルを要することがあります。インテル® マイクロアーキテクチャー開発コード名 Nehalem ベース・プロセッサでは、VMX 遷移のレイテンシーが軽減されています。

## 2.6.8 電力消費の効率化

インテル® マイクロアーキテクチャー開発コード名 Nehalem は、さまざまな負荷状況のもとで高いパフォーマンスと電力効率に優れたパフォーマンスを発揮できるように設計されているだけでなく、システムアイドル時の消費電力を軽減する拡張機能を備えています。インテル® マイクロアーキテクチャー開発コード名 Nehalem ではプロセッサ固有の C6 ステートがサポートされます。これは、OS が ACPI と OS のパワー・マネジメント機構によって管理できる中でリーク消費電力が最も少ないステートです。

## 2.6.9 インテル® マイクロアーキテクチャー開発コード名 Nehalem におけるハイパースレディング・テクノロジーのサポート

インテル® マイクロアーキテクチャー開発コード名 Nehalem は、ハイパースレディング (HT) テクノロジーをサポートしています。このテクノロジーの実装では、2 つの論理プロセッサが各コアの実行/キャッシュリソースの大半を共有します。インテル® マイクロアーキテクチャー開発コード名 Nehalem の HT テクノロジーは、Intel NetBurst® マイクロアーキテクチャー・ベースの旧世代の HT テクノロジーと以下の点が異なります。

- Y インテル® マイクロアーキテクチャー開発コード名 Nehalem では、4 マイクロオペレーション (μop) 幅の実行エンジンと、演算操作を発行可能な 3 つの発行ポートに組み合わされた実行ユニットが搭載されています。
- Y インテル® マイクロアーキテクチャー開発コード名 Nehalem でサポートされている統合メモリー・コントローラーにより、インテル® Core™ i7 プロセッサでは最大 25.6GB/秒のピークメモリー帯域幅を提供します。
- Y 以下のように、バッファリングが深くなり、リソース共有/分割ポリシーが強化されました。
  - HT テクノロジー操作向けに複製されるリソース: レジスター状態、リネームされたリターン・スタック・バッファ、ラージページ ITLB
  - HT テクノロジー操作向けに分割されるリソース: ロードバッファ、ストアバッファ、リオーダーバッファ、スモールページ ITLB は、2 つの論理プロセッサ間で静的に割り当てられます
  - HT テクノロジー操作中に共有されるリソース: リザベーション・ステーション、キャッシュ階層、フィルバッファ、DTLB0 と STLB の両方
  - HT テクノロジー操作中に交互に実行: フロントエンド操作は通常、公平に 2 つの論理プロセッサ間で交互に実行されます
  - HT テクノロジーで管理されないリソース: 実行ユニット

## 2.7 インテル® ハイパースレディング・テクノロジー

インテル® ハイパースレディング・テクノロジー (HT テクノロジー) は、物理プロセッサ・パッケージ内や物理プロセッサ・パッケージの各プロセッサ・コア内で複数の論理プロセッサを提供することで、ソフトウェアがタス



クやスレッドレベルの並列処理の利点を得られることを可能にします。インテル® Xeon® プロセッサにおけるハイパースレッディング・テクノロジーの最初の実装では、単一の物理プロセッサ（プロセッサ・コア）で 2 つの論理プロセッサを提供しました。Knights Landing<sup>†</sup> マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサは、各プロセッサ・コアで 4 つの論理プロセッサをサポートします。Knights Landing<sup>†</sup> マイクロアーキテクチャーにおけるハイパースレッディング・テクノロジーの実装の詳細については第 17 章を参照してください。

ほとんどのインテル® アーキテクチャー・ベースのプロセッサ・ファミリーは、各プロセッサ・コアまたは初期の実装では物理プロセッサで 2 つの論理プロセッサのハイパースレッディング・テクノロジーをサポートします。残りの節では、初期のハイパースレッディング・テクノロジー実装の機能について説明します。大部分の説明は、2 論理プロセッサをサポートする以降のハイパースレッディング・テクノロジーの実装にも適用できます。この節のマイクロアーキテクチャーに関する説明では、個々のマイクロアーキテクチャーへ追加された詳細とハイパースレッディング・テクノロジーの拡張を提供します。

2 つの論理プロセッサは、それぞれが一連のアーキテクチャー・レジスタをすべて所有し、1 つの物理プロセッサのリソースを共有します。HT テクノロジー対応のプロセッサは、2 つのプロセッサ・アーキテクチャー・ステートを保持しているため、オペレーティング・システムやアプリケーションなどのソフトウェアからは、2 つのプロセッサのように見えます。

ピーク時の要求を処理するために必要なリソースを 2 つの論理プロセッサ間で共有しているため、HT テクノロジーはマルチプロセッサ・システム適しており、従来の MP システムに比べてより高いスループット性能を発揮します。

図 2-16 は、HT テクノロジー対応のプロセッサを基本とした一般的なバス接続による SMP (symmetric multiprocessor) を示します。それぞれの論理プロセッサでソフトウェア・スレッドの実行が可能であり、1 つの物理プロセッサ上で最大 2 つのソフトウェア・スレッドを実行できます。2 つのソフトウェア・スレッドは同時に実行されますが、論理プロセッサ 0 からの "add" 演算、および論理プロセッサ 1 からの別の "add" 演算とロードを、同じクロックサイクルで実行エンジンによって同時に実行できます。

HT テクノロジーの最初の実装では、物理実行リソースが共有され論理プロセッサごとにアーキテクチャー・ステートの複製を持ちます。これにより、HT テクノロジーの実装コストが最小限に抑えられ、マルチスレッド・アプリケーションまたはマルチタスク・ワークロードのパフォーマンスが向上します。

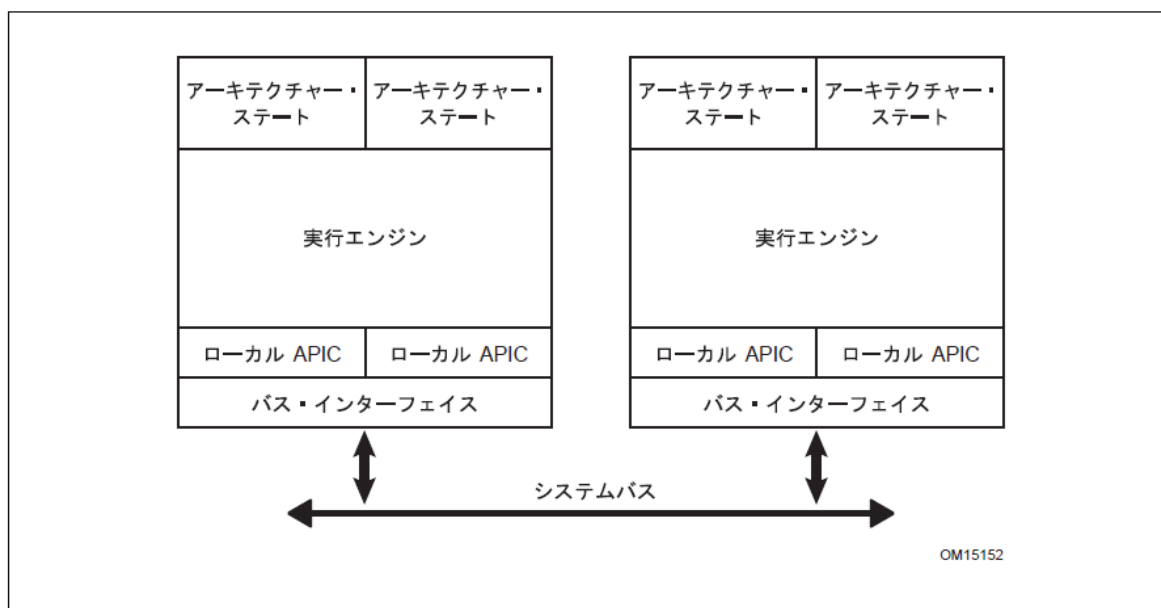


図 2-16 SMP 上のハイパースレッディング・テクノロジー

ハイパースレッディング・テクノロジーを実装すると、次の理由によりパフォーマンスが向上する可能性があります。

- Y オペレーティング・システムおよびユーザー・プログラムでプロセスまたはスレッドをスケジューリングして、各物理プロセッサ内の論理プロセッサ上で同時に実行できます
- Y シングルスレッドのみが実行リソースを消費しているときよりも、高いレベルでプロセッサの実行リソースを活用できます。リソースをより高いレベルで活用すると、システム・スループットが向上します

## 2.7.1 プロセッサ・リソースと HT テクノロジー

物理プロセッサのマイクロアーキテクチャー・リソースの大部分が、論理プロセッサ間で共有されます。論理プロセッサごとに複製されるデータ構造はごくわずかです。この節では、リソースの共有、分割、および複製について説明します。

### 2.7.1.1 リソースの複製

アーキテクチャー・ステートは、論理プロセッサごとに複製されます。アーキテクチャー・ステートは、プログラムの動作を制御したり、計算データを保存するレジスターから構成され、これらのレジスターは、オペレーティング・システムおよびアプリケーション・コードによって使用されます。アーキテクチャー・ステートには、8 つの汎用レジスター、制御レジスター、マシン・ステート・レジスター、デバッグレジスターなどが含まれます。メモリー・タイプ・レンジ・レジスター (MTRR) やパフォーマンス監視リソースなどいくつかの例外があります。アーキテクチャー・ステートと例外の詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A、3B、3C、3D』を参照してください。

命令ポインターやレジスター・リネーミング・テーブルなどのリソースが、2 つの論理プロセッサの実行およびステートの変化を同時に追跡するために複製されます。また、リターン命令の分岐予測を改善するため、リターンスタック分岐予測機構も複製されます。

そのほか、煩雑な操作を低減するため、いくつかのバッファー (2 エントリー命令ストリーミング・バッファーなど) が複製されます。

### 2.7.1.2 リソースの分割

いくつかのバッファーは、各論理プロセッサが使用するエンタリーを半分に制限することで共有されます。これらは分割リソースと呼ばれます。バッファーを分割する理由は、以下のとおりです。

- Y 公平な操作を行うため
- Y 一方の論理プロセッサからの操作が、ストールした可能性のあるもう一方の論理プロセッサの操作により妨げられないようにする

例えば、キャッシュミス、分岐の予測ミス、または命令に依存性があると、数サイクルの間、論理プロセッサは処理を進行できなくなる可能性があります。リソースを分割すると、ストールした論理プロセッサは、処理の進行をブロックしなくなります。

一般に、主要なパイプステージ間で命令をステージ化するためバッファーは分割されます。こうしたバッファーの例として、実行トレースキャッシュ後のマイクロオペレーション ( $\mu\text{op}$ ) キュー、レジスター・リネーム・ステージ後のキュー、リタイアメント用の命令をステージ化するリオーダーバッファー、およびロードバッファーやストアバッファーなどがあります。

ロードバッファーとストアバッファーは、各論理プロセッサのメモリアクセス順序を維持し、メモリアクセス順序の違反を検出するために、すでに分割が実装されています。

### 2.7.1.3 リソースの共有

物理プロセッサ内のリソースの大部分は、リソース (キャッシュやすべての実行ユニットなど) の動的な利用を改善するため完全に共有されています。DTLB など、リニアに処理される共有リソースの中には、エントリーがどちらの論理プロセッサに属しているのかを識別するため論理プロセッサ ID ビットを持つものがあります。

1 次キャッシュは、コンテキスト ID ビットに応じて 2 つのモードで動作します。

- Y 共有モード: L1 データキャッシュが 2 つの論理プロセッサによって完全に共有されます。
- Y 適応モード: 適応モードでは、ページ・ディレクトリを参照するメモリアクセスは、L1 データキャッシュを共有する論理プロセッサ間で等しくマッピングされます。

その他のリソースは、完全に共有されています。

### 2.7.2 マイクロアーキテクチャー・パイプラインと HT テクノロジー

この節では、HT テクノロジーのマイクロアーキテクチャーについて説明します。また、2 つの論理プロセッサからの命令が、パイプラインのフロントエンドとバックエンドでどのように処理されるのか説明します。

2 つのプログラムまたは 2 つのスレッドで実行される命令は同時に処理されますが (実行コアやメモリー階層内のプログラム順序に必ずしも従うわけではありません)、フロントエンドおよびバックエンドには、2 つの論理プロセッサからの命令を選択するいくつかのポイントがあります。すべてのポイントでは、一方の論理プロセッサがパイプライン・ステージを利用できない場合に、2 つの論理プロセッサ間で切り替わります。この場合、もう一方の論理プロセッサは、パイプライン・ステージの全サイクルをすべて利用できます。論理プロセッサがパイプライン・ステージを利用しない場合もありますが、その場合、キャッシュミス、分岐の予測ミス、命令依存性などによりストールしていることが考えられます。

### 2.7.3 フロントエンドのパイプライン

2 つの論理プロセッサによって、実行トレースキャッシュが共有されます。実行トレースキャッシュへのアクセスは、2 つの論理プロセッサによってクロックごとに調整されます。両方の論理プロセッサがトレースキャッシュへのアクセスを要求する場合、あるクロックサイクルで一方の論理プロセッサ向けにキャッシュラインがフェッチされると、次のクロックサイクルでもう一方の論理プロセッサ向けにキャッシュラインがフェッチされます。

一方の論理プロセッサがストールした、または実行トレースキャッシュに命令がなくなった場合、もう一方の論理プロセッサは、最初の論理プロセッサが実行する命令を 2 次キャッシュからフェッチする間、トレースキャッシュの帯域幅をすべて占有できます。

命令がフェッチされ、マイクロオペレーション ( $\mu\text{op}$ ) のトレースが構成されると、マイクロオペレーション ( $\mu\text{op}$ ) はキューに格納されます。このキューは、レジスター・リネーム・パイプライン・ステージから実行トレースキャッシュを分離します。前述したように、どちらの論理プロセッサもアクティブである場合、両方の論理プロセッサが独立して処理を進行できるようにこのキューが分割されています。

### 2.7.4 実行コア

マイクロオペレーション ( $\mu\text{op}$ ) の実行準備が整っている場合、実行コアは 1 サイクルあたり最大 6  $\mu\text{op}$  をディスプレイできます。実行待ちのキューにマイクロオペレーション ( $\mu\text{op}$ ) が配置されると、2 つの論理プロセッサからの命令は区別されなくなります。実行コアとメモリー階層も、どの命令がどの論理プロセッサのものであるかを記憶していません。

実行が完了すると、命令はリオーダーバッファーに格納されます。リオーダーバッファーは、リタイアメント・ステージから実行ステージを分離します。リオーダーバッファーは、各プロセッサがエンタリーの半分ずつを使用できるように分割されます。

## 2.7.5 リタイア

リタイアメント・ロジックは、2 つの論理プロセッサからの命令がリタイアする準備が完了するのを追跡します。リタイアの準備が完了すると、2 つの論理プロセッサ間で切り替えを行うことにより、命令を論理プロセッサごとにプログラムの順序でリタイアします。一方の論理プロセッサで命令をリタイアする準備ができていない場合、もう一方の論理プロセッサがすべてのリタイアメント帯域幅を使用できます。

ストアがリタイアした場合、プロセッサは、そのストアデータを L1 データキャッシュに書き込む必要があります。

選択ロジックは、ストアデータをキャッシュにコミットするため、2 つの論理プロセッサ間で切り替えを行います。

## 2.8 インテル® 64 アーキテクチャー

インテル® 64 アーキテクチャーは、IA-32 インテル® アーキテクチャーの機能をほぼすべてサポートしており、64 ビットのリニアアドレス空間で 64 ビット OS と 64 ビット・アプリケーションを実行できるように拡張されています。インテル® 64 アーキテクチャーには、IA-32e モードと呼ばれる新しい動作モードも導入されました。このモードを利用すると、ソフトウェアのリニアアドレス空間を 64 ビットに広げ、最大で 40 ビットの物理アドレス空間を利用できます。

IA-32e モードは次の 2 つのサブモードから成ります。(1) 互換モードでは、64 ビット OS がほとんどのレガシー 32 ビット・ソフトウェアを無修正で実行できます。(2) 64 ビット・モードでは、64 ビット OS が、64 ビットのリニアアドレス空間にアクセスするように作成されたアプリケーションを実行できます。インテル® 64 アーキテクチャーの 64 ビット・モードでは、ソフトウェアは以下の機能を使用できます。

- Y 64 ビットのフラットなリニアアドレス
- Y 追加された 8 つの汎用レジスター (GPR)
- Y インテル® ストリーミング SIMD 拡張命令 (インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI、PCLMULQDQ) 向けの 8 つの追加レジスター
  - インテル® AVX F16C、インテル® AVX2 または FMA がサポートされる場合、16 個の 256 ビット・レジスター (下位 128 ビットは対応する XMM レジスターに上書きされる)
- Y 64 ビット幅の GPR および命令ポインター
- Y 一律なバイト・レジスター・アドレス
- Y 高速割り込み優先機構
- Y 新しい命令ポインターの相対アドレスモード

## 2.9 SIMD 技術

SIMD 計算 (図 2-17 を参照) は、インテル® MMX® テクノロジーでアーキテクチャーに導入されました。インテル® MMX® テクノロジーは、バイト、ワード、ダブルワードのパックド整数データ型の SIMD 操作を可能にします。この整数は、インテル® MMX® テクノロジー・レジスターと呼ばれる 8 つの 64 ビット・レジスターに格納されます (図 2-18 を参照)。

この SIMD 計算モデルは、インテル® Pentium® III プロセッサでインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) が導入されたことによって拡張されました。インテル® SSE では、4 つのパックド単精度浮動小数点値データ要素を含むオペランドに対し、SIMD 計算を実行できます。オペランドは、メモリーまたは 8 つの 128 ビット

ト XMM レジスター内に置かれます (図 2-18 を参照)。インテル® SSE では、SIMD 計算機能を拡張するために、64 ビットのインテル® MMX® 命令も追加されました。

図 2-17 に一般的な SIMD 計算を示します。4 つのパックドデータ要素 2 つ (X1、X2、X3、X4 と Y1、Y2、Y3、Y4) が並列処理され、対応するデータ要素ペア (X1 と Y1、X2 と Y3、X3 と Y3、X4 と Y4) ごとに同じ操作が行われます。4 つの並列処理の結果は、4 つのパックドデータ要素として格納されます。

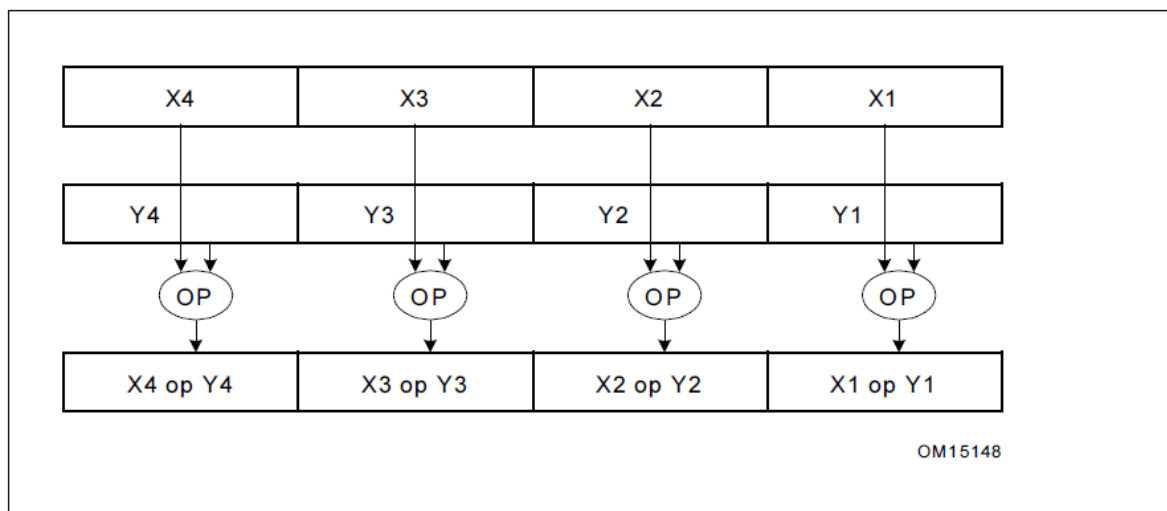


図 2-17 典型的な SIMD 操作

インテル® Pentium® 4 プロセッサでは、さらにインテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3) の導入によって、SIMD 計算モデルが拡張されました。さらに、インテル® Xeon® プロセッサ 5100 番台では、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3) が導入されました。

インテル® SSE2 は、メモリーまたは XMM レジスター内のオペランドを処理します。この技術により SIMD 計算が拡張され、パックド倍精度浮動小数点データ要素と 128 ビット・パックド整数を処理できるようになりました。インテル® SSE2 には 144 の命令が追加され、これらの命令により、2 つのパックド倍精度浮動小数点データ要素、または 16 のパックドバイト整数、8 つのパックドワード整数、4 つのダブルワード整数、2 つのクワッドワード整数を操作できます。

インテル® SSE3 は、特定分野のアプリケーション・パフォーマンスを高める 13 の命令を提供することによって、x87、インテル® SSE、インテル® SSE2 を強化します。これらの分野には、ビデオ処理、複素数演算、スレッド同期などが含まれます。インテル® SSE3 は、SIMD データの非対称処理、水平計算の簡易化、キャッシュライン分割のロード防止を実行する命令によって、インテル® SSE とインテル® SSE2 を補完します。図 2-18 を参照してください。

インテル® SSSE3 では、デジタルビデオと信号処理に関連する 32 の命令によって SIMD 計算がさらに強化されています。

インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI は、メディア処理、テキスト/字句処理、ブロック暗号化/復号のアプリケーションの高速化を図る追加の SIMD 拡張命令です。

SIMD 拡張は、次の点を除いて、IA-32 アーキテクチャーとインテル® 64 アーキテクチャーで同じように動作します。

- Y 64 ビット・モードでは、XMM レジスターを参照する 128 ビット SIMD 命令が 16 個の XMM レジスターにアクセスできます。
- Y 64 ビット・モードでは、32 ビット汎用レジスターを参照する命令が 16 個の汎用レジスターにアクセスできません。

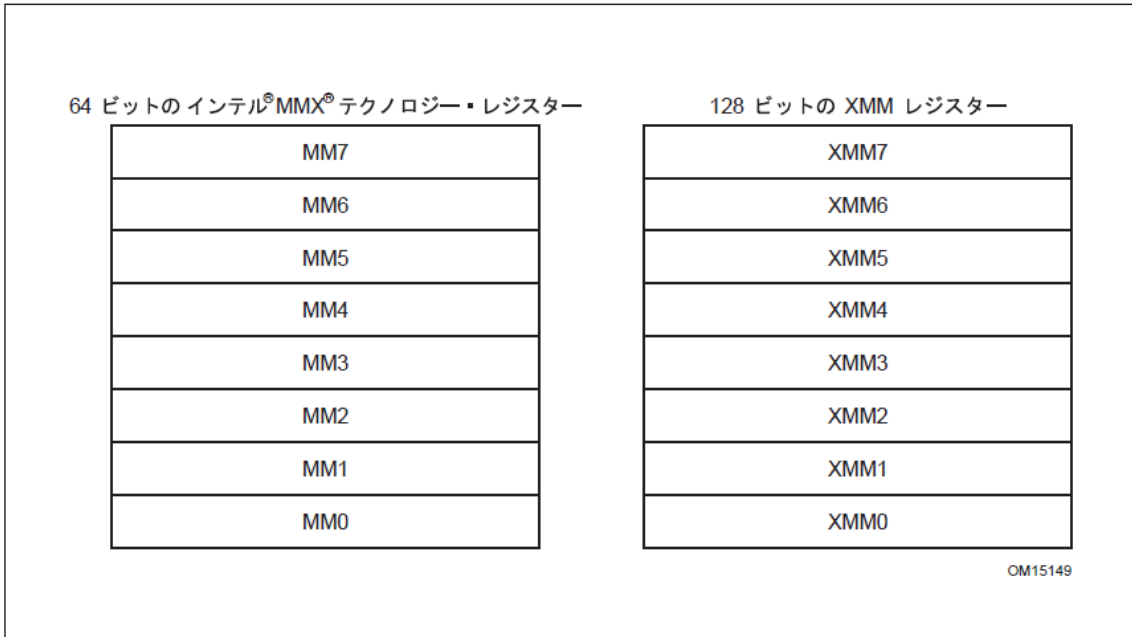


図 2-18 SIMD 命令のレジスターの利用

SIMD を活用して、3D グラフィックス、音声認識、画像処理、科学計算、および以下のような特性を持つアプリケーションのパフォーマンスを向上できます。

- Y 本質的にパラレルである処理
- Y 反復的なメモリー・アクセス・パターン
- Y データに対して局所的な反復操作が行われる
- Y 制御フローがデータに依存しない

## 2.10 SIMD 技術とアプリケーション・レベル拡張のまとめ

SIMD 浮動小数点命令は、IEEE 規格 754 のバイナリー浮動小数点算術演算を完全にサポートしています。SIMD 命令は、すべての IA-32 実行モード (プロテクトモード、実アドレスモード、仮想 8086 モード) で利用できます。

インテル® SSE、インテル® SSE2、インテル® MMX® テクノロジーはアーキテクチャーの拡張技術です。既存のソフトウェアは、修正することなく、これらの技術をサポートするインテル® マイクロプロセッサ上で正しく動作します。また、既存のソフトウェアは、SIMD 技術を組込んだアプリケーションと併用しても正常に動作します。

インテル® SSE 命令とインテル® SSE2 命令には、キャッシュ制御命令とメモリーアクセス順序命令も導入されており、これらの命令によってキャッシュ利用とアプリケーションのパフォーマンスを改善できます。

インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® MMX® テクノロジーに関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の以下の章を参照してください。

- 第 9 章 「Programming with Intel® MMX™ Technology」
- 第 10 章 「Programming with Streaming SIMD Extensions (SSE)」
- 第 11 章 「Programming with Streaming SIMD Extensions 2 (SSE2)」
- 第 12 章 「Programming with SSE3, SSSE3 and SSE4」
- 第 14 章 「Programming with AVX, FMA and AVX2」
- 第 15 章 「Programming with Intel® AVX-512」
- 第 16 章 「Programming with Intel® Transactional Synchronization Extensions」

## 2.10.1 インテル® MMX® テクノロジー

インテル® MMX® テクノロジーでは以下が導入されました。

- Y 64 ビット MMX レジスター
- Y パックドバイト、パックドワード、パックド・ダブルワード整数の SIMD 演算のサポート

インテル® MMX® 命令は、マルチメディアや通信ソフトウェアに効果的です。

## 2.10.2 インテル® ストリーミング SIMD 拡張命令

インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) では以下が導入されました。

- Y 128 ビット XMM レジスター
- Y 4 つのパックド単精度浮動小数点オペランドで構成される 128 ビット・データ型
- Y データ・プリフェッチ命令
- Y 非テンポラルなストア命令などのキャッシュ制御命令とメモリアクセス順序命令
- Y 追加の 64 ビット SIMD 整数のサポート

インテル® SSE 命令は、3D ジオメトリ、3D レンダリング、音声認識、ビデオ・エンコーディング/デコーディングに効果的です。

## 2.10.3 インテル® ストリーミング SIMD 拡張命令 2

インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) では以下が追加されました。

- Y 2 つのパックド倍精度浮動小数点オペランドで構成される 128 ビット・データ型
- Y 16 のバイト整数、8 つのワード整数、4 つのダブルワード整数、または 2 つのクワッドワード整数の SIMD 整数演算用の 128 ビット・データ型
- Y 64 ビット整数オペランドの SIMD 算術演算のサポート
- Y 新しいデータ型と既存のデータ型間での変換の命令
- Y データシャッフルのサポートを拡張
- Y キャッシュ制御操作とメモリアクセス順序操作のサポートを拡張

インテル® SSE2 命令は、3D グラフィックス、ビデオ・デコーディング/エンコーディング、暗号化処理に効果的です。

## 2.10.4 インテル® ストリーミング SIMD 拡張命令 3

インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3) では以下が追加されました。

- Y 非対称計算および水平計算向けの SIMD 浮動小数点命令
- Y キャッシュライン分割を防止する専用 128 ビット・ロード命令
- Y 浮動小数点制御ワード (FCW) に依存しない整数への変換を行う x87 FPU 命令
- Y スレッド同期をサポートする命令

インテル® SSE3 命令は、科学、ビデオ、マルチスレッドのアプリケーションに効果的です。

## 2.10.5 インテル® ストリーミング SIMD 拡張命令 3 補足命令

インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3) では、8 種類のパックド整数演算を高速化する 32 の新しい命令が導入されました。これには以下のものがあります。

- Y 水平加算または減算を実行する 12 の命令
- Y 絶対値を評価する 6 つの命令
- Y 乗算と加算を実行して、ドット積の評価を高速化する 2 つの命令
- Y パックド整数の乗算を高速化して、スケーリングで整数値を生成する 2 つの命令
- Y 2 番目のシャッフル制御オペランドに応じてバイトごとのインプレース・シャッフルを実行する 2 つの命令
- Y ソースオペランド中の対応する要素の符号がマイナスの場合にデスティネーション・オペランド中のパックド整数を否定する 6 つの命令
- Y 2 つのオペランドの合成から得たデータをアライメントする 2 つの命令

## 2.10.6 インテル® ストリーミング SIMD 拡張命令 4.1

インテル® ストリーミング SIMD 拡張命令 4.1 (インテル® SSE4.1) では、ビデオ、画像処理、3D の各アプリケーションを高速化する 47 個の新しい命令が導入されました。また、インテル® SSE4.1 ではコンパイラーのベクトル化機能が改善され、パックド・ダブルワード演算のサポートが大幅に向上しました。これには以下のものがあります。

- Y パックド・ダブルワード乗算を実行する 2 つの命令
- Y 入出力を選択して浮動小数点ドット積を計算する 2 つの命令
- Y ストリーミング・ヒントを WC ロードに提供する 1 つの命令
- Y パックドブレンドを簡素化する 6 つの命令
- Y パックド整数の MIN/MAX のサポートを拡張する 8 つの命令
- Y 選択可能な丸めモードと精度例外のオーバーライドによって浮動小数点の丸めをサポートする 4 つの命令
- Y XMM レジスターからのデータ挿入/抽出を改善する 7 つの命令
- Y パックド整数の形式変換 (符号拡張とゼロ拡張) を改善する 12 の命令
- Y 小さなブロックサイズにおける絶対差の和 (SAD) の生成を改善する 1 つの命令
- Y ワード整数の水平検索操作を支援する 1 つの命令
- Y マスクされた比較を改善する 1 つの命令
- Y パックド・クワッドワードが等しいかどうかの比較を追加する 1 つの命令
- Y 符号なし飽和演算によるダブルワードのパックを追加する 1 つの命令

## 2.10.7 インテル® ストリーミング SIMD 拡張命令 4.2

インテル® ストリーミング SIMD 拡張命令 4.2 (インテル® SSE4.2) では、7 つの新しい命令が導入されました。これには以下のものがあります。

- Y 64 ビットの整数データ要素を比較する 1 個の 128 ビット SIMD 整数命令
- Y 豊富なプリミティブを提供する 4 つの文字列/テキスト処理命令。これらのプリミティブを使用すると、以下のものを高速化できます
  - strlen から strcmp、strcspn に至るまでの基本的小および高度な文字列ライブラリー関数
  - テキストストリームを字句解析するための区切り文字の処理やトークンの抽出
  - XML 処理を含むパーサーやスキーマ検証
- Y 巡回冗長チェックサム・シグネチャーの計算を高速化する 1 個の汎用命令
- Y 整数値のビット・カウントを計算する 1 個の汎用命令



## 2.10.8 インテル® AES New Instructions と PCLMULQDQ

インテル® AES New Instructions (インテル® AES-NI) では、7 個の新しい命令が導入されました。そのうちの 6 つは AES 暗号化/復号規格 (AESN) に基づくアルゴリズムを高速化するためのプリミティブです。

PCLMULQDQ 命令は、汎用ブロックの暗号化を高速化し、最大 64 ビットの 2 進数のキャリーなし乗算を実行できます。

一般的に、AES 規格に基づくアルゴリズムでは、いくつかのプリミティブを介して、複数の反復でブロックデータを変換します。AES 規格では、128、192、および 256 ビットの暗号鍵をサポートしています。暗号鍵のサイズは、それぞれ 10、12、および 14 ラウンドの反復に対応しています。

AES 暗号化には、128 ビットの入力データ (プレーンテキスト) を 128 ビットの暗号化されたブロック (暗号文) にする、有限数の反復処理が含まれています。この処理を AES ラウンドと呼びます。暗号解読は、“逆暗号 (inverse cipher)” の代わりに “等価逆暗号 (Equivalent Inverse Cipher)” を使用して、この反復処理を逆方向に行います。

各ラウンドの暗号処理には、“状態” と “ラウンドキー” という 2 つの入力データがあります。“ラウンドキー” はラウンドごとに異なります。ラウンドキーは、“キー・スケジュール” アルゴリズムを使用して暗号鍵から生成されます。“キー・スケジュール” アルゴリズムは、暗号/復号のデータ処理から独立しており、暗号/復号フェーズとは別に実行することができます。

AES 拡張命令には、暗号化の AES ラウンドを高速化するため 2 つのプリミティブ、“等価逆暗号 (equivalent inverse cipher)” を使用して復号を行う AES ラウンド用の 2 つのプリミティブ、AES のキー・スケジュール生成プロシージャをサポートするための 2 つの命令が含まれます。

## 2.10.9 インテル® アドバンスド・ベクトル・エクステンション

インテル® アドバンスド・ベクトル・エクステンション (インテル® AVX) は、これまでのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) よりもアーキテクチャー面で広範な拡張機能を提供します。インテル® AVX で導入されているアーキテクチャー上の拡張機能は以下のとおりです。

- ✓ 256 ビットのベクトルと SIMD レジスターセットのサポート
- ✓ 128 ビットのインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) が 256 ビットの浮動小数点の命令セットに拡張され、最大 2 倍のパフォーマンスを実現
- ✓ 一般的な 3 オペランド構文の命令構文をサポートすることで、命令のプログラミングの柔軟性を向上させ、新しい拡張命令セットを効率良くエンコード
- ✓ 既存の 128 ビットのインテル® SIMD 拡張命令の拡張により、3 オペランド構文をサポートし、コンパイラーによる高級言語レベルでのベクトル化を簡略化します
- ✓ 256 ビットのインテル® AVX コード、128 ビットのインテル® AVX コード、128 ビットのレガシーコードおよびスカラーコード間の柔軟な導入をサポート

インテル® AVX 命令セットおよび 256 ビット・レジスターの状態管理の詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A、2B、2C、および 3D』を参照してください。インテル® AVX の最適化手法については、本書の第 11 章、「インテル® アドバンスド・ベクトル・エクステンションの最適化」で説明します。

## 2.10.10 半精度浮動小数点変換 (F16C)

VCVTPH2PS と VCVTPS2PH は、単精度浮動小数点データ型と半精度浮動小数点データ型間のデータ変換をサポートする命令です。これら 2 つの命令は、インテル® AVX プログラミング・モデルとともに拡張されました。

## 2.10.11 RDRAND

RDRAND 命令は、暗号化による安全で、決定論的な乱数ビット生成器 (DRBG) によって供給される乱数を取得します。DRBG は NIST SP800-90A 規格を満たすように設計されています。

## 2.10.12 乗算-加算の融合 (FMA) 拡張

FMA 拡張は、融合乗算加算、融合乗算減算、融合乗算加算/減算インターリーブ、および融合乗算加算と融合乗算減算操作における符号付き反転乗算をカバーする高スループット、算術演算機能を備えることでインテル® AVX を強化します。FMA 拡張では、256 ビット・ベクトルと追加の 128 ビットを計算する 36 の 256 ビット浮動小数点命令とスカラー FMA 命令が提供されます。

## 2.10.13 インテル® アドバンスド・ベクトル・エクステンション 2

インテル® アドバンスド・ベクトル・エクステンション 2 (インテル® AVX2) は、インテル® AVX の 128 ビット SIMD 整数命令のほとんどを 256 ビットの数値処理に対応するように拡張します。インテル® AVX2 命令のプログラミング・モデルは、インテル® AVX 命令と同様です。

さらに、インテル® AVX2 では、データ要素のブロードキャスト/置換操作、データ要素ごとにシフトカウントの異なるベクトルシフト命令、連続していないデータをメモリーからフェッチする命令拡張機能が提供されます。

## 2.10.14 汎用ビット処理命令

汎用レジスター上でビット処理を行う命令群は、第 4 世代インテル® Core™ プロセッサ・ファミリーで導入されました。これらの命令のほとんどは、非破壊ソースオペランド構文を提供するため VEX プリフィクス・エンコード体系を使用します。

これらの命令は、CPUID によって示される 3 つの異なる機能フラグで列挙されます。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の 5.1 節と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A、2B、2C、および 2D』の第 3、4 および 5 章をご覧ください。

## 2.10.15 インテル® トランザクショナル・シンクロナイゼーション・エクステンション

インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) は、第 4 世代インテル® Core™ プロセッサ・ファミリーで導入され、ロックベースのプログラミング・モデルを持つマルチスレッド・アプリケーションのロックで保護されたクリティカル・セクションのパフォーマンスを向上させます。

インテル® TSX の背景と詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 16 章「Programming with Intel® Transactional Synchronization Extensions」をご覧ください。

マルチスレッド・アプリケーションにおけるロックベースのクリティカル・セクションでインテル® TSX を使用するソフトウェア・チューニングの推奨事項は、同マニュアルの第 12 章「Intel® TSX Recommendations」を参照してください。

## 2.10.16 RDSEED

RDSEED、ADCX、および ADOX 命令は、インテル® Core™ M プロセッサ・ファミリーで導入されました。

RDSEED 命令は、暗号化による安全で、決定論的な乱数ビット生成器拡張 (NRBG) によって供給される乱数を取得します。NRBG は、NIST SP 800-90B と NIST SP 800-90C 規格を満たすように設計されています。

## 2.10.17 ADCX と ADOX 命令

ADCX と ADOX 命令は、MULX 命令を連携して大きな整数値の計算をスピードアップします。詳細は、<http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integersquaring-ia-aper.pdf> (英語) のドキュメントをご覧ください。

## 第 4 章 SIMD アーキテクチャー向けのコーディング

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3 をサポートします。拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1 をサポートします。インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートします。インテル® マイクロアーキテクチャー開発コード名 Westmere ベースのプロセッサは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI をサポートします。インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge ベースのプロセッサは、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、インテル® AES-NI、PCLMULQDQ、インテル® AVX をサポートします。

インテル® Pentium® 4 プロセッサ、インテル® Xeon® プロセッサ、インテル® Pentium® M プロセッサは、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジーをサポートしています。90nm テクノロジーを採用した、ハイパースレッディング・テクノロジー対応インテル® Pentium® 4 プロセッサでは、インテル® SSE3 が導入されました。インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサは、インテル® SSE3/インテル® SSE2/インテル® SSE、インテル® MMX® テクノロジーをサポートしています。

SIMD (Single Instruction, Multiple Data) 技術により、マルチメディア、信号処理、モデリングなどの高度なアプリケーションを開発できます。

SIMD 手法は、テキスト/文字列処理、字句解析、構文解析のアプリケーションにも適用できます。これについては、第 11 章「テキスト処理/字句解析/構文解析におけるインテル® SSE4.2 と SIMD プログラミング」で説明しています。インテル® AES-NI を最適化する手法については、5.10 節で説明します。

これらの機能によって可能となる高度なパフォーマンスを十分に活用するには、以下のことを行う必要があります。

- Y プロセッサが、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートしていることを確認します。
- Y オペレーティング・システムが、インテル® MMX® テクノロジーとインテル® SSE をサポートしていることを確認します (OS がインテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートする条件は、インテル® SSE をサポートする条件と同じです)。
- Y 本書で説明する最適化手法とスケジューリング手法を導入します。
- Y スタック・アライメントおよびデータ・アライメント手法を使用して、データの適切なアライメントを維持し、メモリの使用効率を高めます。
- Y 必要に応じて、インテル® SSE とインテル® SSE2 のキャッシュ制御命令を利用します。

### 4.1 プロセッサによる SIMD 技術のサポートをチェック

この節では、プロセッサがインテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 をサポートしているかを確認する方法について説明します。

次の 3 つの方法で、アプリケーションに SIMD 技術を実装できます。

1. アプリケーションのインストール時に、SIMD 技術のサポートをチェックします。希望の SIMD 技術が使用できる場合は、適切な DLL をインストールします。

- プログラムの実行時に SIMD 技術のサポートをチェックし、実行時に適切な DLL をインストールします。この方法は、異なるマシン上で実行されるプログラムに効果的です。
- 複数のバージョン (SIMD 技術を使用するバージョンと使用しないバージョン) のルーチンを含む「ファット」バイナリーコードを作成します。プログラムの実行時に SIMD 技術のサポートをチェックし、適合するバージョンのルーチンを実行します。この方法は、異なるマシン上で実行されるプログラムに特に効果的です。

#### 4.1.1 インテル® MMX® テクノロジーのサポートをチェック

インテル® MMX® テクノロジーが利用可能な場合、CPUID.01H:EDX[BIT 23] が 1 です。例 4-1 のコードを使用して、プロセッサがインテル® MMX® テクノロジーを利用できるかどうかをテストします。

例 4-1 CPUID によるインテル® MMX® テクノロジーの識別

```

; CPUID 命令の実装を確認
...
; シグネチャーが "GENUINE INTEL" であることを確認
...
;
mov eax, 1 ; 機能フラグを要求
cpuid ; 0FH, 0A2H CPUID 命令
test edx, 00800000h ; 機能フラグでインテル® MMX テクノロジー・ビット (ビット 23) が
; 1 であることを確認
jnz Found
    
```

CPUID 命令の詳細については、『インテル® プロセッサの識別と CPUID 命令』(資料番号 241618) を参照してください。

#### 4.1.2 インテル® ストリーミング SIMD 拡張命令のサポートをチェック

プロセッサがインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) をサポートしているかどうかをチェックする手順は、インテル® MMX® テクノロジーのチェックとよく似ています。しかし、インテル® SSE 命令を使用するアプリケーションの動作の一貫性を保証するには、オペレーティング・システム (OS) がコンテキスト・スイッチ時にインテル® SSE 状態の保存と復元をサポートする必要があります。

システムがインテル® SSE をサポートしているかをチェックするには、次の手順に従います。

- プロセッサが CPUID 命令をサポートしていることを確認します。
- CPUID の機能ビットをチェックして、プロセッサがインテル® SSE をサポートしていることを確認します。

例 4-2 に、CPUID 機能フラグのインテル® SSE 機能ビット (ビット 25) を確認する方法を示します。

例 4-2 CPUID によるインテル® SSE の識別

```

; CPUID 命令の実装を確認
; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1 ; 機能フラグを要求
cpuid ; 0FH, 0A2H cpuid 命令
test EDX, 002000000h ; 機能フラグのビット 25 が 1 であることを確認
jnz Found
    
```

#### 4.1.3 インテル® ストリーミング SIMD 拡張命令 2 のサポートをチェック

プロセッサがインテル® SSE2 をサポートしているかをチェックする手順は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE2 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE2 をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサがインテル® SSE2 をサポートしていることを確認します。

例 4-3 に、CPUID 機能フラグのインテル® SSE2 機能ビット (ビット 26) を確認する方法を示します。

例 4-3 CPUID によるインテル® SSE2 の識別

```

; CPUID 命令の実装を確認
mov eax, 1
cpuid
test EDX, 00400000h
jnz Found
; シグネチャーが "GENUINE INTEL" であることを確認
; 機能フラグを要求
; 0FH, 0A2H CPUID 命令
; 機能フラグのビット 26 が 1 であることを確認
    
```

### 4.1.4 インテル® ストリーミング SIMD 拡張命令 3 のサポートをチェック

インテル® SSE3 には 13 の命令が含まれ、そのうち 11 個は SIMD 形式または x87 形式のプログラミングに適しています。プロセッサがインテル® SSE3 をサポートしているかをチェックする手順は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE3 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE3 の x87 命令と SIMD 命令をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサが CPUID 命令をサポートしていることを確認します。
2. CPUID の ECX 機能ビット 0 をチェックして、プロセッサがインテル® SSE3 テクノロジーをサポートしていることを確認します。

例 4-4 に、CPUID 機能フラグのインテル® SSE3 機能ビット (ビット 0) を確認する方法を示します。

例 4-4 CPUID によるインテル® SSE3 の識別

```

; CPUID 命令の実装を確認
mov eax, 1
cpuid
test ECX, 000000001h
jnz Found
; シグネチャーが "GENUINE INTEL" であることを確認
; 機能フラグを要求
; 0FH, 0A2H CPUID 命令
; 機能フラグのビット 0 が 1 であることを確認
    
```

ソフトウェアは、MONITOR と MWAIT 命令を実行する前に、これらの命令がサポートされているかどうかをチェックする必要があります。MONITOR と MWAIT 命令が利用可能かどうかは、例 4-4 と同様のコードシーケンスによって確認できます。これらの命令が利用可能かどうかは、ecx に返される値のビット 3 によって示されます。

### 4.1.5 インテル® ストリーミング SIMD 拡張命令 3 補足命令のサポートをチェック

プロセッサがインテル® SSSE3 をサポートしているかをチェックする方法は、インテル® SSE のチェックとよく似ています。OS がインテル® SSSE3 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSSE3 をサポートしているかをチェックするには、次の手順に従います。

1. プロセッサが CPUID 命令をサポートしていることを確認します。

2. CPUID の機能ビットをチェックして、プロセッサがインテル® SSSE3 テクノロジーをサポートしていることを確認します。

例 4-5 に、CPUID 機能フラグのインテル® SSSE3 機能ビット (ビット 9) を確認する方法を示します。

#### 例 4-5 CPUID によるインテル® SSSE3 の識別

```

; CPUID 命令の実装を確認
; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1 ; 機能フラグを要求
cpuid ; 0FH, 0A2H CPUID 命令
test ECX, 000000200h ; 機能フラグのビット 9が 1 であることを確認
jnz Found
    
```

### 4.1.6 インテル® ストリーミング SIMD 拡張命令 4.1 のサポートをチェック

プロセッサがインテル® SSE4.1 をサポートしているかを確認する方法は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE4.1 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE4.1 をサポートしているかを確認するには、次の手順に従います。

1. プロセッサが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサがインテル® SSE4.1 をサポートしていることを確認します。

例 4-6 に、CPUID 機能フラグのインテル® SSE4.1 機能ビット (ビット 19) を確認する方法を示します。

#### 例 4-6 CPUID によるインテル® SSE4.1 の識別

```

; CPUID 命令の実装を確認
; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1 ; 機能フラグを要求
cpuid ; 0FH, 0A2H CPUID 命令
test ECX, 000080000h ; 機能フラグのビット 19が 1 であることを確認
jnz Found
    
```

### 4.1.7 インテル® ストリーミング SIMD 拡張命令 4.2 のサポートをチェック

プロセッサがインテル® SSE4.2 をサポートしているかを確認する方法は、インテル® SSE のチェックとよく似ています。OS がインテル® SSE4.2 をサポートする必要条件は、OS がインテル® SSE をサポートする必要条件と同じです。

システムがインテル® SSE4.2 をサポートしているかを確認するには、次の手順に従います。

1. プロセッサが CPUID 命令をサポートしていることを確認します。
2. CPUID の機能ビットをチェックして、プロセッサがインテル® SSE4.2 をサポートしていることを確認します。

例 4-7 に、CPUID 機能フラグのインテル® SSE4.2 機能ビット (ビット 20) を確認する方法を示します。

#### 例 4-7 CPUID によるインテル® SSE4.2 の識別

```

; CPUID 命令の実装を確認
; シグネチャーが "GENUINE INTEL" であることを確認
mov eax, 1 ; 機能フラグを要求
cpuid ; 0FH, 0A2H CPUID 命令
test ECX, 000100000h ; 機能フラグのビット 20が 1 であることを確認
jnz Found
    
```

## 4.1.8 PCLMULQDQ およびインテル® AES-NI 命令のサポートを検出

アプリケーションは、インテル® AES-NI 命令 (AESDEC/AESDECLAST/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST) を使用する前に、プロセッサがインテル® AES-NI 拡張命令をサポートしていることをチェックする必要があります。CPUID.01H:ECX.AESNI[bit 25] = 1 である場合、インテル® AES-NI 拡張命令がサポートされています。

また、PCLMULQDQ 命令を使用する前に、アプリケーションは CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1 であるかチェックする必要があります。

インテル® SSE ステートをサポートしているオペレーティング・システムは、インテル® AES-NI 拡張命令および PCLMULQDQ 命令を使用するアプリケーションもサポートします。これは、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4 の必要条件と同じです。

### 例 4-8 インテル® AES-NI 命令の検出

```

; CPUID 命令の実装を確認
mov eax, 1          ; シグネチャーが "GENUINE INTEL" であることを確認
cpuid              ; 機能フラグを要求
cpuid              ; 0FH, 0A2H CPUID 命令
test ECX, 00200000h ; 機能フラグのビット 25が 1 であることを確認
jnz Found
    
```

### 例 4-9 PCLMULQDQ 命令の検出

```

; CPUID 命令の実装を確認
mov eax, 1          ; シグネチャーが "GENUINE INTEL" であることを確認
cpuid              ; 機能フラグを要求
cpuid              ; 0FH, 0A2H CPUID 命令
test ECX, 000000002h ; 機能フラグの 1 ビットが 1 であることを確認
jnz Found
    
```

## 4.1.9 インテル® AVX 命令の検出

インテル® AVX は、256 ビットの YMM レジスタステートを処理します。アプリケーションは、図 4-1 に示す手順に従って、YMM ステートを処理する新しい拡張命令が利用できることを検出します。

アプリケーションは、インテル® AVX を使用する前に、オペレーティング・システムが XGETBV 命令と YMM レジスタステートをサポートすることに加え、プロセッサが XSAVE/XRSTOR およびインテル® AVX 命令を使用して YMM ステートを管理できることを確認する必要があります。以下の手順により両方をチェックします (強く推奨)。

1. CPUID.1:ECX.OSXSAVE[bit 27] = 1 を検出します (アプリケーションで使用できるように XGETBV が有効になっている<sup>3</sup>)。
2. XGETBV を発行し、XFEATURE\_ENABLED\_MASK[2:1] = '11b' であることを確認します (XMM ステートおよび YMM ステートが OS で有効になっている)。
3. CPUID.1:ECX.AVX[bit 28] = 1 を検出します (AVX 命令がサポートされている)。

注意: 手順 3 は手順 1 と 2 に対して任意の順序で実行できます。

<sup>3</sup> CPUID.01H:ECX.OSXSAVE が 1 を示す場合、プロセッサで XSAVE, XRSTOR, XGETBV、プロセッサによる拡張ステート・ビット・ベクトル XFEATURE\_ENALBED\_MASK レジスタがサポートされていることも間接的に示しています。そのため、アプリケーションは、XSAVE および OSXSAVE に対する CPUID 機能フラグのチェックを簡略化できます。XSETBV は特権命令です。



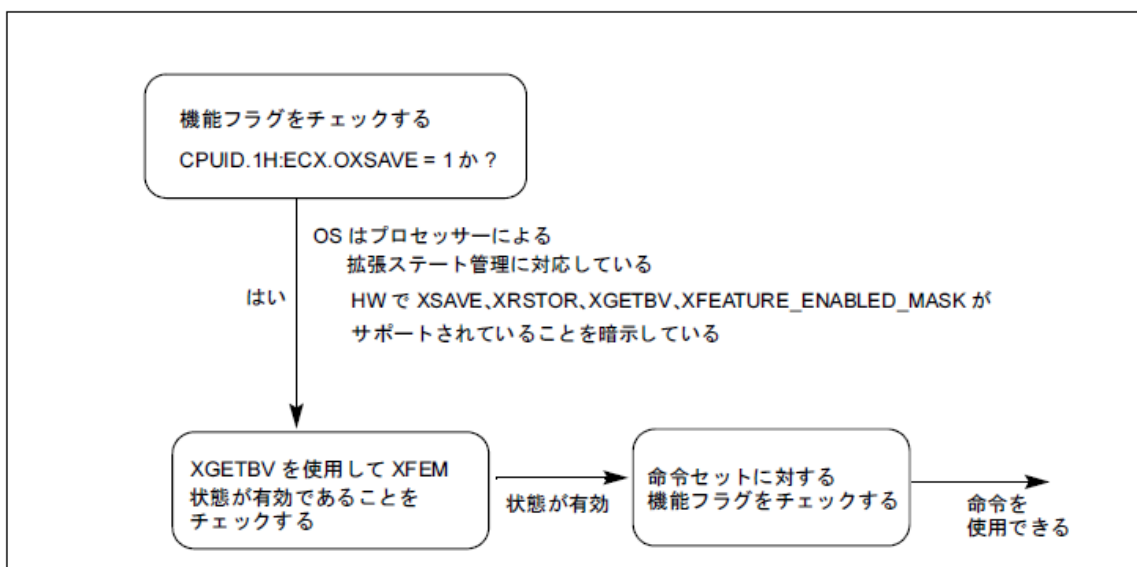


図 4-1 アプリケーションでインテル® AVX を検出する一般的な手順

以下の擬似コードは、アプリケーションでインテル® AVX を検出するときの推奨プロセスです。

#### 例 4-10 インテル® AVX 命令の検出

```

INT supports_AVX()
{
  mov eax, 1
  cpuid
  and ecx, 018000000H
  cmp ecx, 018000000H ; OSXSAVE とインテル® AVX 機能フラグの両方をチェック
  jne not_supported
  ; プロセッサはインテル® AVX 命令をサポートし OS で XGETBV が有効
  mov ecx, 0 ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
  XGETBV ; 結果は EDX:EAX
  and eax, 06H
  cmp eax, 06H ; OS が XMM と YMM ステートをサポートしているのをチェック
  jne not_supported
  mov eax, 1
  jmp done
NOT_SUPPORTED:
  mov eax, 0
done:

```

注意: アプリケーションが CPUID.1:ECX.AVX[bit 28] に排他的に依存したり、CPUID.1:ECX.XSAVE[bit 26] に全面的に依存することは賢明とは言えません。これらは、オペレーティング・システムのサポートではなく、ハードウェアのサポートを示すためです。YMM ステート管理がオペレーティング・システムで有効になっていない場合、インテル® AVX 命令は CPUID.1:ECX.AVX[bit 28] に関わりなく #UD を生成します。“CPUID.1:ECX.XSAVE[bit 26] = 1” は、OS がステート管理で実際に XSAVE プロセスを使用することを保証するものではありません。

#### 4.1.10 VEX エンコードされた AES および VPCLMULQDQ の検出

VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST 命令は YMM ステートを処理します。検出手順では、CPUID.1:ECX.AES[bit 25] = 1 のチェックと、インテル® AVX に対するアプリケーション・サポートの検出の手順を組み合わせる必要があります。

例 4-11 VEX エンコードされたインテル® AES-NI 命令の検出

```

INT supports_VAESNI()
{
    mov eax, 1
    cpuid
    and ecx, 01A000000H
    cmp ecx, 01A000000H ; OSXSAVE、インテル® AVX、インテル® AES-NI 機能フラグをチェック
    jne not_supported
    ; プロセッサはインテル® AVX と VEX エンコードのインテル® AES-NI 命令をサポートし
    ; OS で XGETBV が有効
    mov ecx, 0 ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
    XGETBV ; 結果は EDX:EAX
    and eax, 06H
    cmp eax, 06H ; OS が XMM と YMM ステートをサポートしているのをチェック
    jne not_supported
    mov eax, 1
    jmp done
NOT_SUPPORTED:
    mov eax, 0
done:

```

同様に、VPCLMULQDQ の検出手順では、CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 のチェックと、インテル® AVX に対するアプリケーション・サポートの検出手順を組み合わせる必要があります。

以下に擬似コードを示します。

例 4-12 VEX エンコードされたインテル® AES-NI 命令の検出

```

INT supports_VPCLMULQDQ()
{
    mov eax, 1
    cpuid
    and ecx, 018000002H
    cmp ecx, 018000002H ; OSXSAVE、インテル® AVX、PCLMULQDQ 機能フラグのチェック
    jne not_supported
    ; プロセッサはインテル® AVX と VEX エンコード PCLMULQDQ 命令をサポートし
    ; OS で XGETBV が有効
    mov ecx, 0 ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
    XGETBV ; 結果は EDX:EAX
    and eax, 06H
    cmp eax, 06H ; OS が XMM と YMM ステートをサポートしているのをチェック
    jne not_supported
    mov eax, 1
    jmp done
NOT_SUPPORTED:
    mov eax, 0
done:

```

### 4.1.11 F16C 命令の検出

float16 命令を使用するアプリケーションは、インテル® AVX と同様の検出手順に従って命令が利用できるか確認する必要があります。

- Y OS は YMM ステート管理のサポートを有効にします。
- Y インテル® AVX をサポートするプロセッサは、CPUID 機能フラグの CPUID.01H:ECX.AVX[bit 28] = 1 で示されます。
- Y 16 ビット浮動小数点変換命令をサポートするプロセッサは、CPUID 機能フラグ (CPUID.01H:ECX.F16C[bit 29] = 1) で確認できます。

アプリケーションは、図 4-2 の一般的な手順に従って float16 変換命令を検出します。

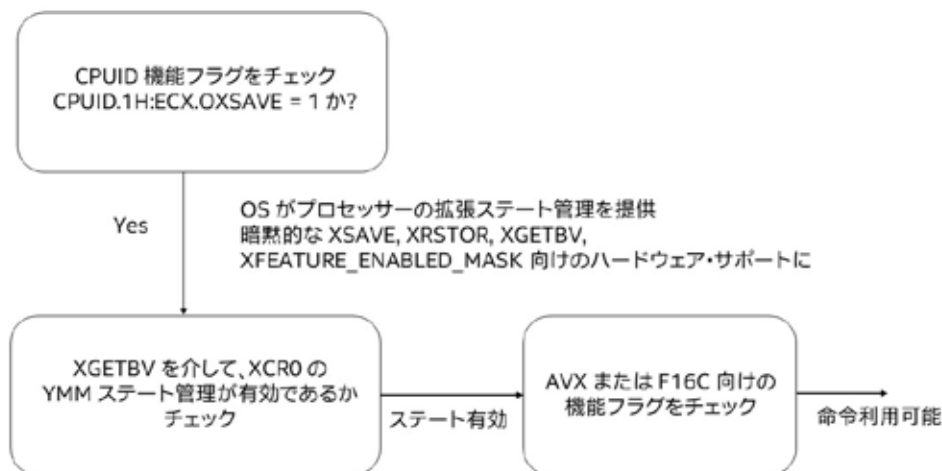


図 4-2 float 16 を検出する一般的な手順

```

INT supports_f16c()
{ ; 結果は eax に生成
  mov eax, 1
  cpuid
  and ecx, 038000000H
  cmp ecx, 038000000H ; OSXSAVE、インテル® AVX、F16C 機能フラグのチェック
  jne not_supported
  ; プロセッサはインテル® AVX、F16C 命令をサポートし OS で XGETBV が有効
  mov ecx, 0 ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
  XGETBV ; 結果は EDX:EAX
  and eax, 06H
  cmp eax, 06H ; OS が XMM と YMM ステートをサポートしているのをチェック
  jne not_supported
  mov eax, 1
  jmp done
NOT_SUPPORTED:
  mov eax, 0
done:
}
  
```

#### 4.1.12 FMA 命令の検出

FMA のハードウェア・サポートは、CPUID.1:ECX.FMA[bit 12]=1 によって示されます。

アプリケーション・ソフトウェアは、ハードウェアがインテル® AVX をサポートしていることを確認した後、CPUID 機能フラグ CPUID.1:ECX.FMA[bit 12] によって FMA のサポートを識別する必要があります。以下に、推奨される FMA の検出疑似シーケンスを示します。

```

INT supports_fma()
{ ; 結果は eax に生成
  mov eax, 1
  cpuid
  and ecx, 018001000H
  cmp ecx, 018001000H      ; OSXSAVE、インテル® AVX と FMA 機能フラグのチェック
  jne not_supported
  ; プロセッサは AVX、FMA 命令をサポートし OS で XGETBV が有効
  mov ecx, 0                ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
  XGETBV                    ; 結果は EDX:EAX
  and eax, 06H
  cmp eax, 06H              ; OS が XMM と YMM ステートをサポートしているのをチェック
  jne not_supported
  mov eax, 1
  jmp done
NOT_SUPPORTED:
  mov eax, 0
done:
}

```

### 4.1.13 インテル® AVX2 の検出

ハードウェアがインテル® AVX2 をサポートするかどうかは、CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5] = 1 によって確認できます。

アプリケーション・ソフトウェアは、ハードウェアがインテル® AVX をサポートしていることを確認した後、CPUID 機能フラグ CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5] によってインテル® AVX2 のサポートを検出する必要があります。以下に、推奨されるインテル® AVX2 の検出疑似シーケンスを示します。

```

INT supports_avx2()
{ ; 結果は eax に生成
  mov eax, 1
  cpuid
  and ecx, 018000000H
  cmp ecx, 018000000H      ; OSXSAVE とインテル® AVX 機能フラグの両方をチェック
  jne not_supported
  ; プロセッサはインテル® AVX 命令をサポートし OS で XGETBV が有効
  mov eax, 7
  mov ecx, 0
  cpuid
  and ebx, 20H
  cmp ebx, 20H              ; インテル® AVX2 機能フラグをチェック
  jne not_supported
  mov ecx, 0                ; XFEATURE_ENABLED_MASK レジスターに 0 を指定
  XGETBV                    ; 結果は EDX:EAX
  and eax, 06H
  cmp eax, 06H              ; OS が XMM と YMM ステートをサポートしているのをチェック
  jne not_supported
  mov eax, 1
  jmp done
NOT_SUPPORTED:
  mov eax, 0
done:
}

```

## 4.2 SIMD プログラミングにおけるコード変換に関する留意事項

インテル® VTune™ パフォーマンス拡張環境は、コードの評価とチューニングを支援するツールを提供します。これらのツールを使用する前に、次の点を確認する必要があります。

1. 現在のコードで、インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 を使用するメリットがあるか。
2. コードは整数コードか浮動小数点コードか。
3. 必要とする整数ワードサイズまたは浮動小数点精度はどれくらいか。
4. どのコーディング手法を使用すべきか。
5. いずれのガイドラインに従ったら良いか。
6. データ型をどのように配置し、アライメントするか。

図 4-3 に、既存のコードを、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2 コードに変換するプロセスのフローチャートを示します。

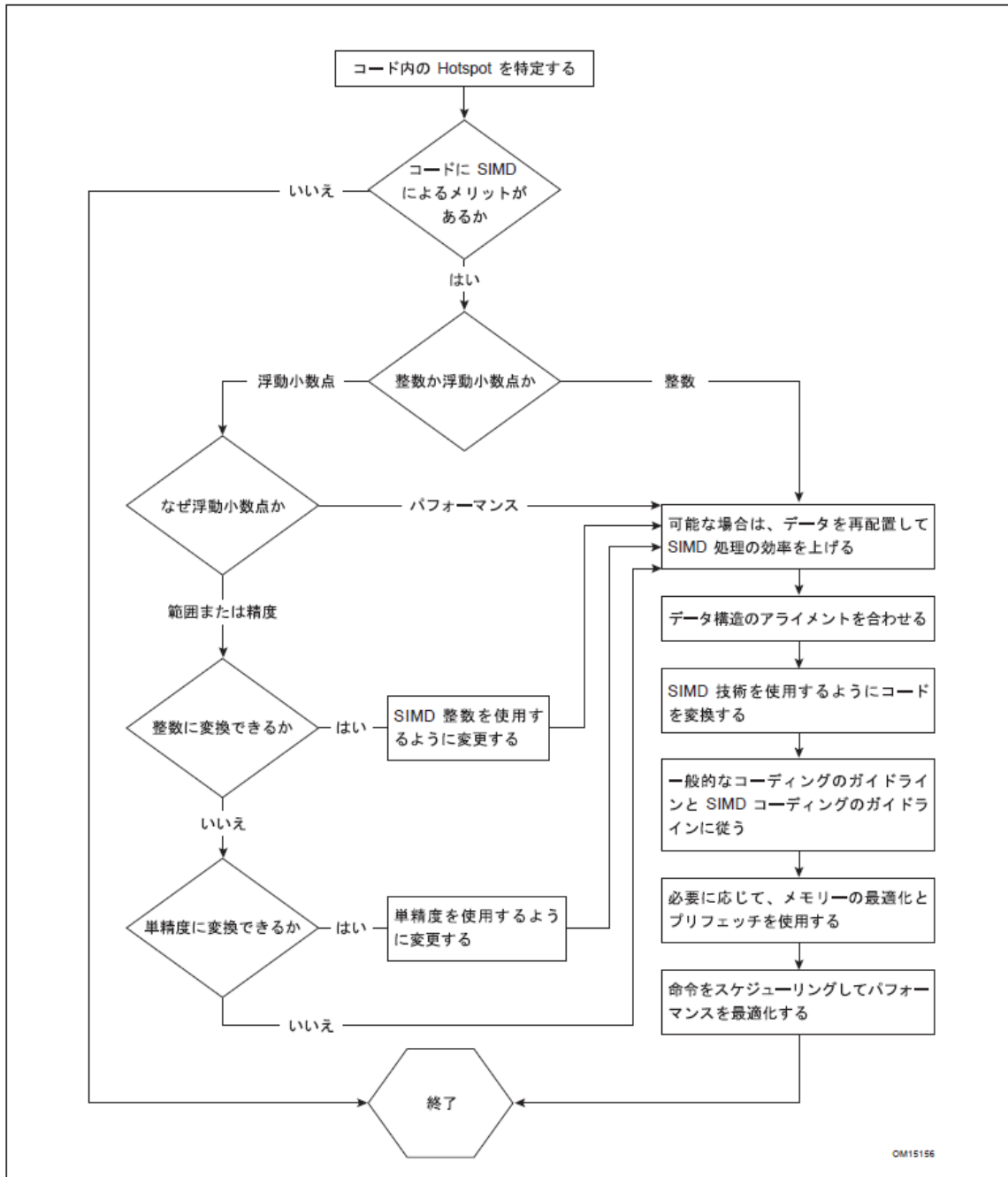


図 4-3 インテル® ストリーミング SIMD 拡張命令コードへの変換チャート

SIMD 技術の効果を最大限に発揮するには、コード内の以下の個所を特定する必要があります。

- Y 大量の計算を必要とする部分
- Y 頻繁に実行され、パフォーマンスに影響を与える部分
- Y ほとんどデータに依存しない制御フローを使用する部分
- Y 浮動小数点計算が必要な部分
- Y 一度に 16 バイトのデータを移動することでメリットを得られる部分
- Y 少ない命令を使ってコード化できる計算の部分
- Y キャッシュ階層を効率的に使用するために、プログラマーの指示を必要とする部分

## 4.2.1 ホットスポットの特定

パフォーマンスを最適化するには、インテル® VTune™ Amplifier を使用して、計算時間の大部分を占有するコードセクションを特定します。このセクションは、ホットスポットと呼ばれます。付録 A「アプリケーション・パフォーマンス・ツール」を参照してください。

インテル® VTune™ Amplifier は、特定のモジュールのホットスポットを検出して、大部分の CPU 時間を消費するパフォーマンス上問題があるコードセクションを識別します。ホットスポットを表示することで、大部分の CPU 時間を消費しているパフォーマンス上問題の可能性のあるコード内のセクションを識別できます。

インテル® VTune™ Amplifier は、メモリー・ロケーション、関数、クラス、またはソースファイル別にホットスポットを表示できます。ホットスポットをダブルクリックすると、そのホットスポットのソースまたはアセンブリー表示が開き、ホットスポット内の各命令のパフォーマンスに関する詳しい情報が表示されます。

インテル® VTune™ Amplifier によって、ソースコードのすべてのレベルで、詳しい分析データとパフォーマンス・データが得られ、アセンブリー言語レベルでの助言も得られます。コードコーチは、C/C++、Fortran、Java\* プログラムのパフォーマンスを向上できる個所を分析して特定し、具体的な最適化手法を提案します。必要に応じて、コードコーチは、(インテル® パフォーマンス・ライブラリーに含まれている) 高度に最適化された組み込み関数と関数の使用を提案する疑似コードを表示します。インテル® VTune™ Amplifier は、インテル® Pentium® 4 プロセッサを含むインテル® アーキテクチャー (IA) ベースのプロセッサ向けに設計されており、IA プロセッサ上のプログラムを最適化するための詳しい手法を提案できます。詳細と最新情報については、付録 A.1.1「インテル® 64 プロセッサと IA-32 プロセッサの推奨される最適化設定」をご覧ください。

## 4.2.2 SIMD 実行コードへの変換にメリットがあるかどうか判定

SIMD 技術を使用することでメリットが得られるコードを特定するのは、時間と困難を伴う作業です。計算集約型のアプリケーションは、SIMD コードに変換することでパフォーマンスが向上する可能性があり、これには次のものが考えられます。

- Y 音声圧縮アルゴリズム/フィルター
- Y 音声認識アルゴリズム
- Y ビデオ表示/キャプチャルーチン
- Y レンダリングルーチン
- Y 3D グラフィックス (ジオメトリー)
- Y 画像/動画処理アルゴリズム
- Y 空間的 (3D) オーディオ
- Y 物理的モデリング (グラフィックス、CAD)
- Y ワークステーション・アプリケーション
- Y 暗号化アルゴリズム
- Y 複素数演算

一般に、SIMD コードへの変換に適したコードとは、8 ビット、16 ビットまたは 32 ビット整数、単精度 32 ビット浮動小数点データ、または倍精度 64 ビット浮動小数点データ (インテル® SSE2) の連続した配列を操作する、小さいサイズの反復ループを含むコードです (整数および浮動小数点データアイテムは、メモリーに連続して配置されている必要があります)。これらのコードでは、ループが繰り返されるため、アプリケーションの処理時間が長くなります。しかし、これらのルーチンは、いずれかの SIMD 技術を採用すると、パフォーマンスが大幅に向上する可能性があります。

SIMD 技術を使用できる場所が特定できたら、現在のアルゴリズムと変更後のアルゴリズムのどちらが優れたパフォーマンスを得られるか評価する必要があります。

## 4.3 コーディング手法

インテル® SSE4.2、インテル® SSE4.1、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジーの SIMD 機能を利用するには、コーディング・アルゴリズムにおいて新しい手法が必要となります。その 1 つはベクトル化です。ベクトル化とは、シーケンシャルに実行される (スカラー) コードを、SIMD アーキテクチャーの並列処理を利用する、並列実行が可能なコードに変換するプロセスを指します。

この節では、アプリケーションに SIMD アーキテクチャー導入するコーディング手法について説明します。

コードをベクトル化して、SIMD アーキテクチャーを利用するには、以下の手順に従います。

- Y メモリアクセスにおいて並列実行の妨げとなる依存関係があるかどうかを判断します。
- Y 内部ループで「ストリップマイニング」を行い、ループの反復回数を、 $1/(\text{SIMD 操作の長さ})$  に減らします (例えば、単精度浮動小数点 SIMD 演算の場合は  $1/4$ 、16 ビット整数 SIMD 演算の場合は  $1/8$  にします)。
- Y SIMD 命令を使用してループを再構成します。

本章の各節では、上記の手順について詳しく述べていきます。また、インテル® C++ コンパイラーの自動ベクトル化機能についても説明します。インテル® SSE4.2 固有の手順については第 11 章を、インテル® AVX/インテル® AVX2 固有の手順は第 12 章を、インテル® AVX-512 固有の手順については第 13 章を参照してください。

### 4.3.1 各種コーディング手法

ソフトウェア開発者は、アセンブリ・コードから得られるパフォーマンスの向上と、改善されたコストを比較する必要があります。対象とするプラットフォーム向けにアセンブリ言語で直接プログラミングすれば、必要とするパフォーマンスが得られる可能性は高まりますが、アセンブリ・コードにはプロセッサ・アーキテクチャー間の移植性がなく、開発と保守に大きなコストがかかります。

各種の SIMD 技術を利用することで、アセンブリ言語の代わりに高水準言語を使用して、パフォーマンス目標を達成できる可能性があります。インテル® SSE4.2、インテル® SSE4.1、インテル® SSSE3、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジー向けに設計された、新しい C/C++ 言語拡張がこのようなコーディングを可能にします。

図 4-4 に、手作業でコーディングしたアセンブリ言語と各種のコーディング手法を比較した、パフォーマンスとプログラミングの容易さ/移植性のトレードオフを示します。

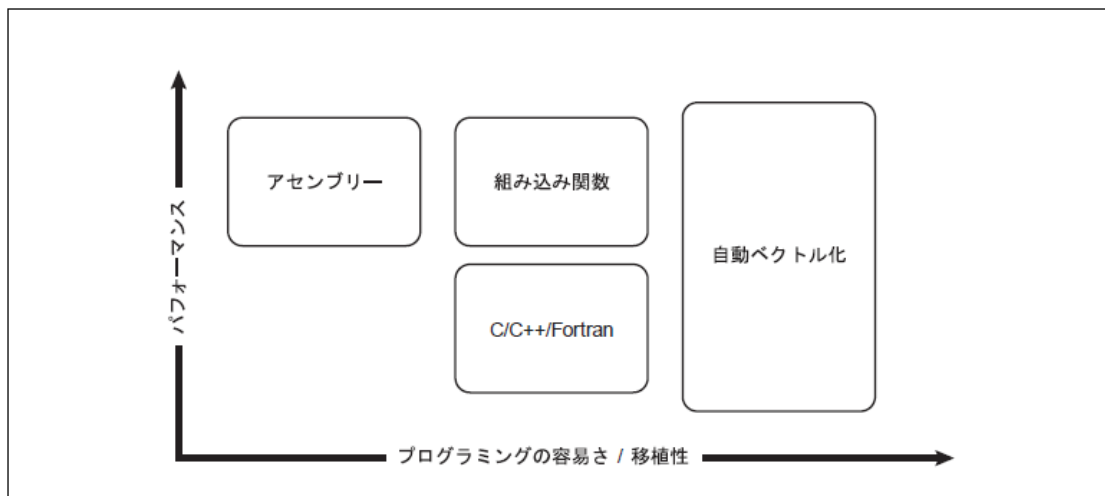


図 4-4 手作業でコーディングされたアセンブリと高水準コンパイラーのパフォーマンスのトレードオフ



以下の例では、各種のコーディング手法を使用して、インテル® SSE のメリットが得られるアルゴリズムを表現する方法を示します。インテル® SSE4.2、インテル® SSE4.1、インテル® SSSE3、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジー向けに、単精度浮動小数点データ、倍精度浮動小数点データ、整数データに対して、同じ手法を適用できます。

この節で説明する使用モデルの基礎として、例 4-13 に示す簡単なループを考えます。

#### 例 4-13 4 回反復される簡単なループ

```
void add(float *a, float *b, float *c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

このループは 4 回しか反復しません。そのため、このコードはインテル® ストリーミング SIMD 拡張命令で容易に置き換えられます。

16 バイト境界へのデータ・アライメントを必要とするインテル® ストリーミング SIMD 拡張命令を効果的に使用できるように、この節のすべての例では、ルーチンに渡される配列 a, b, c は、呼び出し元ルーチンで 16 バイト境界にアライメントされているものと仮定します。アライメントを保証する手法については、インテル® Pentium® 4 プロセッサのアプリケーション・ノートを参照してください。

以下の各項では、インライン・アセンブリー、組込み関数、C++ ベクトルクラス、自動ベクトル化の各コーディング手法について詳しく説明します。

### 4.3.1.1 アセンブリー

主要なループは、アセンブラーを使用するか、C/C++ コード内でインライン・アセンブリー (C-asm) を使用して、アセンブリー言語で直接コーディングできます。インテル® コンパイラーまたはアセンブラーは、新しい命令とレジスターを認識し、それに対応するコードを直接生成できます。このモデルを適切に導入すると、各種のコーディング手法の中で最も高い性能が得られる可能性があります。しかし、異なるプロセッサ・アーキテクチャー間での移植性はありません。

例 4-14 に、インライン・アセンブリーによるインテル® ストリーミング SIMD 拡張命令のコーディングを示します。

#### 例 4-14 インライン・アセンブリーによるインテル® ストリーミング SIMD 拡張命令のコーディング

```
void add(float *a, float *b, float *c)
{
    __asm {
        mov eax, a
        mov edx, b
        mov ecx, c
        movaps xmm0, XMMWORD PTR [eax]
        addps xmm0, XMMWORD PTR [edx]
        movaps XMMWORD PTR [ecx], xmm0
    }
}
```

### 4.3.1.2 組み込み関数

組み込み関数を使用すると、アセンブリ言語の代わりに C/C++ 形式のコーディング・スタイルで命令セットを利用できます。インテル® C++ コンパイラーでは、複数の組み込み関数セットを定義しており、これらの組み込み関数は、それぞれインテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令、インテル® ストリーミング SIMD 拡張命令 2 を含む最新の SIMD 拡張命令をサポートしています。組み込み関数のオペランドとして、64 ビットと 128 ビットのオブジェクトを表す 4 種類の新しい C データ型 (`__m64`、`__m128`、`__m128i`、`__m128d`) が定義されています。`__m64` はインテル® MMX 命令の整数 SIMD 演算に、`__m128` は単精度浮動小数点 SIMD 演算に、`__m128i` はインテル® ストリーミング SIMD 拡張命令 2 の整数 SIMD 演算に、`__m128d` は倍精度浮動小数点 SIMD 演算に、それぞれ使用されます。これらのデータ型によって、プログラマーはアルゴリズムのコーディングで命令セットを直接選択でき、コンパイラーはレジスターの割り当てと命令のスケジューリングを可能な限り最適化できます。これらの組み込み関数は、コンパイラーがサポートしているすべてのインテル® アーキテクチャー・ベースのプロセッサ間で移植性があります。

組み込み関数を使用したプログラムのパフォーマンスは、アセンブリを使用したプログラムに近いレベルに達します。また、組み込み関数を使用すると、アセンブリを直接記述する場合と比べて、プログラムの作成と保守のコストを大幅に軽減できます。組み込み関数とその使用方法に関する詳細は、インテル® C/C++ コンパイラーのドキュメントを参照してください。

例 4-15 に、組み込み関数を使用して例 4-13 のループをコーディングした例を示します。

例 4-15 組み込み関数でコーディングされた 4 回反復するループ

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

組み込み関数は、実際のインテル® ストリーミング SIMD 拡張命令のアセンブリ・コードに 1 対 1 で対応しています。組み込み関数のプロトタイプを定義している `xmmintrin.h` ヘッダーファイルは、インテル® C++ コンパイラーに含まれています。

インテル® MMX® テクノロジー命令セット用の組み込み関数も定義されており、これらの組み込み関数は、MM レジスターを表す `__m64` データ型を使用します。値は、バイト、short 型整数、32 ビット値、または 64 ビット・オブジェクトを指定できます。

ただし、この組み込み関数のデータ型は、ANSI C データ型ではないため、以下の制限に従って使用する必要があります。

- Y 組み込み関数データ型は、戻り値またはパラメーターとして、代入文の左辺でのみ使用します。ほかの算術式 (“+”、“>>” など) にこのデータ型を使用することはできません。
- Y 組み込み関数データ型は、バイト要素/構造にアクセスするための共用体などの集合体のオブジェクトとして使用してください。`__m64` オブジェクトのアドレスを指定できます。
- Y 組み込み関数データ型のデータは、本書で説明するインテル® MMX® テクノロジーの組み込み関数でしか使用できません。

ハードウェア命令の詳細については、『Intel® Architecture MMX® Technology Programmer's Reference Manual』を参照してください。さらに詳しい情報については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』をご覧ください。

### 4.3.1.3 クラス

インテル® C++ コンパイラーでは、一連の C++ クラスライブラリーも定義されています。これによって、高水準の抽象化が可能になり、インテル® MMX® テクノロジー、ストリーミング SIMD 拡張命令、ストリーミング SIMD 拡張命令 2 および最新の SIMD 命令セットを使用して、さらに柔軟なプログラミングが可能になります。これらのクラスは、組み関数向けの使いやすく柔軟なインターフェイスを提供します。これにより、開発者は、特定の操作に対してどの組み関数またはアセンブリー言語命令を使用するかを気にせずに、自然な C++ コードを記述できます。これらの C++ クラスは、組み関数をベースにしているため、C++ クラスを使用したアプリケーションのパフォーマンスは、組み関数を使用したアプリケーションに近いレベルに達します。C++ クラスの使用方法に関する詳細は、『SIMD 操作向けインテル® C++ クラス・ライブラリー・ユーザー・ガイド (ドキュメント番号 693500)』を参照してください。

例 4-16 に、ベクトルクラス・ライブラリーを使用した C++ コードを示します。この例では、ルーチンに渡される配列は、16 バイトにアライメントされているものと仮定します。

例 4-16 ベクトルクラスを使用した C++ コード

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
    F32vec4 *av=(F32vec4 *) a;
    F32vec4 *bv=(F32vec4 *) b;
    F32vec4 *cv=(F32vec4 *) c;
    *cv=*av + *bv;
}
```

この例で、fvec.h はクラス定義ファイルであり、F32vec4 は 4 つの浮動小数点値の配列を表すクラスです。“+” および “=” 演算子は多重定義されており、前の例における実際のインテル® ストリーミング SIMD 拡張命令のコードは抽象化されています。このコードは元のコードによく似ていますが、より簡単かつ短時間でプログラミング可能です。

この例では、ルーチンに渡される配列は、16 バイトにアライメントされているものと仮定します。

### 4.3.1.4 自動ベクトル化

インテル® C++コンパイラーの最適化では、例 4-13 のようなループを自動的にベクトル化できます (つまり、インテル® ストリーミング SIMD 拡張命令コードに変換する)。コンパイラーは、プログラマーが行うのと類似した方法で、どのループが SIMD コードへの変換に適しているかを識別します。その際に、コンパイラーは、以下の条件がベクトル化の妨げになっていないかを判断します。

- Y ループのレイアウトとデータ構造体
- Y それぞれの反復の間でのデータアクセスの依存関係

コンパイラーは、このような判断を基にループをベクトル化します。これで、アプリケーションは SIMD 命令を使用できます。

ただし、自動ベクトル化を適用できるのは、特定のタイプのループだけであり、ほとんどの場合、自動ベクトル化の機能を利用するには、ユーザーがコンパイラーに情報を与える必要があります。

例 4-17 に、例 4-13 の 4 回反復されるループを自動的にベクトル化するコードを示します。

例 4-17 ループの自動的ベクトル化

```
void add (float *restrict a,
         float *restrict b,
         float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

インテル®C++コンパイラー (バージョン 9.0 以降) の /Qax (Windows\*), -ax (Linux\* および macOS\*) と /Qrestrict (Windows\*), -restrict (Linux\* および macOS\*) オプションを使用して、このコードをコンパイルします。

引数リストの restrict 指示子は、ポインターが指すメモリーへのエイリアスがほかにないことをコンパイラーに知らせます。つまり、使用されるポインターは、そのポインターが有効なスコープ内でそのメモリーへアクセスする唯一の方法であることを示します。コンパイラーは、restrict 指示子がなくても、ランタイムデータの依存関係テストを利用してループをベクトル化します。この場合、生成されるコードは、パラメーターのオーバーラップに基づいて、ループのシーケンシャル実行またはベクトル実行を動的に選択します (インテル® C++ コンパイラーのドキュメントを参照)。restrict 指示子を利用すると、関連するオーバーヘッドをすべて防止できます。

詳細は、インテル® C++ コンパイラーのドキュメントを参照してください。

## 4.4 スタックとデータ・アライメント

SIMD 技術向けに記述されたコードのパフォーマンスを最大限に発揮するには、この節で説明するガイドラインに従って、メモリー上にデータを配置する必要があります。アセンブリー・コードでアライメントの合っていないデータにアクセスすると、パフォーマンスが大きく低下します。

### 4.4.1 アライメントとデータ・アクセス・パターンの隣接性

インテル® MMX® テクノロジーで定義された 64 ビット・パックド・データ型と、インテル® ストリーミング SIMD 拡張命令およびインテル® ストリーミング SIMD 拡張命令 2 を含む最新の SIMD 命令セットで利用される 128 ビット・パックド・データ型では、データアクセスのアライメントがずれる可能性が高くなります。インテル® MMX® テクノロジー命令とインテル® ストリーミング SIMD 拡張命令を使用する場合、多くのアルゴリズムのデータ・アクセス・パターンは基本的にアライメントが合っていません。以下では、パディングや、配列へのデータ要素の編成など、データアクセスを改善する手法について説明します。インテル® SSE3 では、キャッシュライン分割を防止する専用命令 LDDQU が提供されています (5.7.1.1 節「キャッシュライン分割を防止する補助的な手法」を参照)。

#### 4.4.1.1 パディングによるデータのアライメント

SIMD 命令を使用して SIMD データにアクセスする場合、宣言を変更するだけでデータへのアクセスを改善できます。例えば、メモリー空間内の位置と属性を表す構造体を宣言する場合を考えてみます。

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

SIMD ワードの 4 つの要素のうち 3 つを使用して、X, Y, Z の計算を繰り返し実行するものと仮定します。例については、4.5.1 節「データ構造体のレイアウト」を参照してください。配列 pt 内の最初の要素のアライメントが合っている場合でも、2 番目の要素は 7 バイト後に始まるため、アライメントが合わなくなります (各 2 バイトの short 型値 3 つ + 1 バイト = 7 バイト)。

パディング変数 `pad` を追加することで、この構造体のサイズは 8 バイトになります。これにより、最初の要素のアライメントが 8 バイト (64 ビット) に合っていれば、それに続くすべての要素も 8 バイトにアライメントします。次に例を示します。

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

#### 4.4.1.2 配列のデータ隣接性を保証

次のコードについて考えてみます。

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

このコードでは、2 次元の `y` に `scale` 値を乗算します。for ループが配列 `pt` 内の各次元の `y` にアクセスするため、隣接するデータにアクセスできません。そのため、キャッシュミスが増え、フェッチされる各キャッシュラインの利用効率が低下し、複数のキャッシュラインにまたがるアクセスが増えるなどの理由で、アプリケーションのパフォーマンスが低下する可能性があります。

次の宣言によって、`scale` 変数による乗算をベクトル化し、データ・アクセス・パターンのアライメントを改善できます。

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

SIMD 技術を使用する場合、データ編成の選択が重要です。データに対して実行される操作の種類に基づいて、データ編成を慎重に選択する必要があります。アプリケーションによっては、従来のデータ配置では十分なパフォーマンスが得られないことがあります。

この問題の典型的な例として、FIR フィルターが上げられます。FIR フィルターは、実質的には、係数タップ数の全体にわたるベクトルドット積です。

次のコードについて考えてみます。

```
(data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1]),
```

このコードで、データ要素 `i` のフィルター操作がデータ要素 `j` から始まるベクトルドット積であるとすれば、データ要素 `i+1` のフィルター操作はデータ要素 `j+i` から始まります。

64 ビットにアライメントされているデータベクトルと 64 ビットにアライメントされた係数ベクトルがあると仮定すると、最初のデータ要素に対するフィルター操作は完全にアライメントが合いますが、2 番目のデータ要素では、データベクトルに対するアクセスのアライメントが合わなくなります。FIR フィルターのアライメントのずれを防ぐ方法の例は、インテル® ストリーミング SIMD 拡張命令およびフィルターに関するインテルのアプリケーション・ノートを参照してください。

データ構造体の複製とパディングによって、本来アライメントが合わないアルゴリズム内のデータアクセスの問題を回避できます。4.5.1 節「データ構造体のレイアウト」では、データ構造体の編成とトレードオフについて、詳しく説明しています。

#### 注意

複製およびパディングは、データサイズが大きくなる代わりに、アライメントのずれの問題を解決して、アライメントの合わないデータアクセスの大きなペナルティーを排除する方法です。コードを開発する際には、このトレードオフを考慮して、目的とするパフォーマンスが得られる方法を選択する必要があります。

## 4.4.2 128 ビット SIMD 技術向けのスタック・アライメント

インテル® ストリーミング SIMD 拡張命令、インテル® ストリーミング SIMD 拡張命令 2、および最新の SIMD 拡張命令 (インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2) で最大限のパフォーマンスを得るには、メモリーオペランドのアライメントが 16 バイト境界に合っていないと、パフォーマンスが低下します。アライメントの合っていないデータを操作すると、大きなペナルティが生じます。しかし、ほとんどのコンパイラーがサポートしている既存の IA-32 向けのソフトウェア規則 (STDCALL、CDECL、FASTCALL) は、特定のローカルデータと特定のパラメーターの 16 バイト・アライメントを保証する機構を備えていません。そのため、インテルでは、新しい `__m128*` データ型 (`__m128`、`__m128d`、`__m128i`) をサポートするアライメントに関する新しい IA-32 ソフトウェア規則を定義しました。この規則には、以下の条件があります。

- Y インテル® ストリーミング SIMD 拡張命令、インテル® ストリーミング SIMD 拡張命令 2、および最新の SIMD 拡張命令 (インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2) のデータを使用する関数は、16 バイトにアライメントが合ったスタックフレームを用意する必要があります。
- Y `__m128*` パラメーターは 16 バイト境界にアライメントが合っていないと、パフォーマンスが低下します。これによって、引数ブロック内に (パディングによる) 「すき間」が生じる可能性があります。

この節で説明する、インテル® C++ コンパイラーがサポートする新しい規則は、アセンブリー言語コードのためのガイドラインとしても使用できます。この節の多くでは、インテル® C++ コンパイラーで定義された、4 つの 32 ビット浮動小数点値の配列を表す `__m128*` データ型の使用を前提としています。

## 4.4.3 インテル® MMX® テクノロジー向けのデータ・アライメント

多くのコンパイラーは、変数のアライメントを調整する機能を備えています。この機能は、変数のビット長を適切な境界に揃えます。一部の変数が、指定したとおりに正しくアライメントされない場合、例 4-18 の C アルゴリズムを使用して、それらの変数のアライメント調整できます。

### 例 4-18 64 ビット・データ・アライメント向けの C アルゴリズム

```
/* Make newp を 64 ビット要素の 64 ビットにアラインした配列のポインターにする*/
double *p, *newp;
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));
newp = (p+7) & (~0x7);
```

例 4-18 の C アルゴリズムは、64 ビット要素の配列のアライメントを 64 ビット境界に合わせます。定数の 7 は、64 ビット要素内のバイト数より 1 小さい値 (すなわち、8-1) から求められます。この方法でデータのアライメントを合わせることで、キャッシュライン境界にまたがるアクセスによって発生するパフォーマンス・ペナルティを避けることができます。

データ・アライメントを調整するもう 1 つの方法は、64 ビット境界にアライメントが合っているメモリー・ロケーションに、データをコピーする方法です。データが頻繁にアクセスされる場合、これによってパフォーマンスが大きく向上する可能性があります。

## 4.4.4 128 ビット・データ向けのデータ・アライメント

インテル® SSE/インテル® SSE2/インテル® SSE3/インテル® SSSE3/インテル® SSE4.1/インテル® SSE4.2 が使用する 128 ビット XMM レジスターとの間でデータをロードまたはストアする場合、データのアライメントは 16 バイトである必要があります。データのアライメントが合っていないと、パフォーマンスが大幅に低下したり、実行中にフォルトが発生することがあります。

アライメントが合っていないデータを XMM レジスターとの間でコピーできる MOVE 命令 (および組み込み関数) も用意されていますが、この操作はアライメントが合ったアクセスに比べはるかに時間がかかります。データが 16 バイトにアライメントされていない場合、プログラマーやコンパイラーがこれに気付かずに、アライメントが合ったデータ用の命令を使用すると、フォルトが発生します。したがって、データの 16 バイト・アライメントを常に維持する

必要があります。インテル® MMX® テクノロジーの必要条件は 8 バイト・アライメントですが、データの 16 バイト・アライメントは、インテル® MMX® テクノロジー・コードにも効果があります。

以下では、インテル® C++ コンパイラーがサポートするインテル® Pentium® 4 プロセッサ向けのデータ・アライメント手法について説明します。

#### 4.4.4.1 コンパイラーがサポートするアライメント

インテル® C++ コンパイラーは、データのアライメントを保証するため、以下の手法を用意しています。

##### F32vec4 または \_\_m128 データ型によるアライメント

コンパイラーは、F32vec4 または \_\_m128 のデータ宣言またはパラメーターを検出すると、グローバルデータ、ローカルデータ、パラメーターに対し、オブジェクトのアライメントを強制的に 16 バイト境界に合わせます。この宣言が関数の中にある場合、コンパイラーは、関数のスタックフレームのアライメントも合わせて、ローカルデータとパラメーターの 16 バイト・アライメントを保証します。デバッグコンパイルと最適化されたコンパイル（「リリース」モード）の両方でコンパイラーが生成するスタックフレームのレイアウトの詳細は、インテル® コンパイラーのドキュメントを参照してください。

##### `__declspec(align(16))` 指定

この定義をデータ宣言の前に置くと、強制的にデータ・アライメントを 16 バイトに合わせます。この方法は、128 ビット・データ型に代入されるローカルデータまたはグローバルデータを宣言するときに便利です。構文は次のようになります。

```
__declspec(align(integer-constant))
```

`integer-constant` は、2 の整数乗（32 を超えない値）です。例えば、以下のコードは、アライメントを 16 バイトにします。

```
__declspec(align(16)) float buffer[400];
```

この宣言の後、変数 `buffer` は、\_\_m128 型または F32vec4 型のオブジェクトを 100 個含むものとして参照できます。次のコードでは、F32vec4 オブジェクト `x` は、データのアライメントが合った状態で構成されます。

```
void foo() {
    F32vec4 x = *(__m128 *) buffer;
    ...
}
```

`__declspec(align(16))` の宣言がないとフォルトが発生します。

##### union 構造体を使用したアライメント

union と 128 ビット・データ型を組み合わせて使用すると、デフォルトでデータ構造体のアライメントを合わせるように、コンパイラーに指示ができます。できるだけ、この方法を使用することが望ましく、この union によるアライメントはプログラムの真の意図（`__declspec(align(16))` データが使用されること）をコンパイラーに通知できるため、`__declspec(align(16))` による強制的なアライメントより適しています。次に例を示します。

```
union {
    float f[400];
    __m128 m[100];
} buffer;
```

この例では、union の中に \_\_m128 型があるため、デフォルトで 16 バイト・アライメントが適用されます。\_\_declspec(align(16)) を使用してアライメントを強制する必要はありません。

C++ では、以下のコードのように、強制的に class/struct/union 型のアライメント指示することも可能です (ただし、C ではこの手法は使用できません)。

```
struct __declspec(align(16)) my_m128
{
    float f[4];
};
```

このような class 内のデータがインテル® ストリーミング SIMD 拡張命令またはインテル® ストリーミング SIMD 拡張命令 2 で使用される場合、union を使用して、アライメントを明示的に指定する方が推奨されます。C++ では、名前なし union を使用できるため、この方法はさらに便利です。

```
class my_m128 {
    union {
        __m128 m;
        float f[4];
    };
};
```

union が無名であるため、名前 m と f を、my\_m128 の直接のメンバー名として使用できます。これに対して、\_\_declspec(align(16)) は、C と C++ のどちらでも、class、struct、または union のメンバーに指定されると無効になります。

#### \_\_m64 または double データ型を使用したアライメント

状況に応じて、コンパイラーは、\_\_m64 または double 型データを使用するルーチンのアライメントを、デフォルトで 16 バイトに合わせます。コマンドライン・オプション /Qsalign16 (Windows\* のみ) を使用して、128 ビット・データを含むルーチンだけをアライメントするようコンパイラーに指示できます。デフォルトでは、/Qsalign8 が適用されます。このオプションは、8 バイトまたは 16 バイト・データ型を使用するルーチンだけを 16 バイトにアライメントするようコンパイラーに指示します。

詳細については、インテル® C++ コンパイラーのドキュメントを参照してください。

## 4.5 メモリー使用効率の改善

インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4.1、インテル® SSE4.2、およびインテル® MMX® テクノロジーの組み込み関数向けにデータとアルゴリズムを再配置すると、メモリーのパフォーマンスを向上できます。メモリーのパフォーマンスを向上させる方法には、以下のものがあります。

- Y データ構造体のレイアウト
- Y ベクトル化とメモリーを効率良く使用するためのストリップマイニング
- Y ループ・ブロッキング

また、キャッシュ制御命令、プリフェッチ、ストリーミング・ストアを使用して、メモリーの使用効率を大幅に改善できます。

関連情報: 第 7 章「キャッシュ利用の最適化」も参照してください。



## 4.5.1 データ構造体のレイアウト

3D 変換や照明計算などのアルゴリズムには、頂点のデータを編成する基本的な方法が 2 つあります。従来の方法は、それぞれの頂点を 1 つの構造体で表す、構造体配列 (AoS) 編成です (例 4-19)。しかし、この方法では、SIMD 技術の機能を十分に活用することができません。

例 4-19 AoS データ構造

```
typedef struct{
    float x,y,z;
    int a,b,c;
    ...
} Vertex;
Vertex Vertices[NumOfVertices];
```

SIMD 技術を使用するコードに適した処理方法は、各座標を表す配列でデータを編成することです (例 4-20 を参照)。このデータ編成は、配列構造体 (SoA) と呼ばれます。

例 4-20 SoA データ構造

```
typedef struct{
    float x[NumOfVertices];
    float y[NumOfVertices];
    float z[NumOfVertices];
    int a[NumOfVertices];
    int b[NumOfVertices];
    int c[NumOfVertices];
    ...
} VerticesList;
VerticesList Vertices;
```

AoS 形式のデータを計算するには、2 つの方法があります。1 つは元の AoS 形式のままデータを操作する方法で、もう 1 つはデータを入れ替えて、動的に SoA 形式に再編成する方法です。例 4-21 に、ドット積計算における 2 つの方法のコード例を示します。

例 4-21 AoS コードと SoA コードの例

```
; ベクトルの配列 (Array) と固定ベクトル (Fixed) の内積は、3D 光源処理の一般的な計算です。
; Array = (x0,y0,z0),(x1,y1,z1),... で、Fixed = (xF,yF,zF) とするとき、
; 内積はスカラー量 d0 = x0*xF + y0*yF + z0*zF として定義されます。
;
; 構造体の配列 (AoS) コード
; DC と記されている値は、"don't-care." の意味です。

; AoS モデルでは、頂点データは xyz フォーマットになります。
movaps xmm0, Array ; xmm0 = DC, x0, y0, z0
movaps xmm1, Fixed ; xmm1 = DC, xF, yF, zF
mulps xmm0, xmm1 ; xmm0 = DC, x0*xF, y0*yF, z0*zF
movhps xmm, xmm0 ; xmm = DC, DC, DC, x0*xF

addps xmm1, xmm0 ; xmm0 = DC, DC, DC,
                    ; x0*xF+z0*zF
movaps xmm2, xmm1
shufps xmm2, xmm2, 55h ; xmm2 = DC, DC, DC, y0*yF
addps xmm2, xmm1 ; xmm1 = DC, DC, DC,
                    ; x0*xF+y0*yF+z0*zF

; 配列の構造体 (SoA) コード
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
```

```

; Z = z0,z1,z2,z3
; A = xF,xF,xF,xF
; B = yF,yF,yF,yF
; C = zF,zF,zF,zF
movaps xmm0, X ; xmm0 = x0,x1,x2,x3
movaps xmm1, Y ; xmm1 = y0,y1,y2,y3
movaps xmm2, Z ; xmm2 = z0,z1,z2,z3
mulps xmm0, A ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
mulps xmm1, B ; xmm1 = y0*yF, y1*yF, y2*yF, y3*yF
mulps xmm2, C ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
addps xmm0, xmm1
addps xmm0, xmm2 ; xmm0 = (x0*xF+y0*yF+z0*zF), ...
    
```

元の AoS 形式に対して SIMD 演算を実行すると、計算の回数が多くなり、SIMD 要素の一部の利点を活用できない計算が行われます。一般的に、この方法は効率的ではありません。

AoS 形式のデータ計算に推奨される方法は、要素の各セットを SoA 形式に再編成 (スウィズリング) してから、SIMD 技術を使用してそのデータを処理することです。データの入れ替えは、プログラムの実行時に動的に行うことも、データ構造体の生成時に静的に行うこともできます。例については、第 5 章と第 6 章を参照してください。通常は、元の AoS を使用するより、データを動的に入れ替える方が適切です。ただし、データを動的に編成すると、計算時の命令の数が増えるため、実行効率が多少低下します。これに対して、データ構造体がレイアウトされるときにデータを静的に入れ替えれば、実行時のオーバーヘッドは生じません。したがって、これが最も良い方法といえます。

すでに説明したように、SoA 配置の方が、SIMD 技術の並列処理を効率的に使用できます。これは、計算でアクセスするデータが垂直形式で準備されるためです。つまり、4 つの SIMD 実行スロットを使用して、要素 x0, x1, x2, x3 に xF, xF, xF, xF を掛けて、4 つの結果を求めることができます。これに対して、AoS データを直接計算すると、水平的な操作が行われ、SIMD 実行スロットを消費しますが、1 つのスカラー結果しか得られません (このことは、例 4-21 の多くが “don't-care” (DC) スロットであることによって示されています)。

データ構造体に SoA 形式を使用すると、キャッシュと帯域幅の使用効率も向上します。構造体の各要素に対するアクセス頻度が異なる場合 (例えば、要素 x, y, z がほかのエントリーの 10 倍の頻度でアクセスされる場合) は、SoA を使用すると、メモリーを節約でき、不要なデータアイテム a, b, c のフェッチを避けることができます。

例 4-22 ハイブリッド SoA データ構造体

```

NumOfGroups = NumOfVertices/SIMDwidth
typedef struct{
    float x[SIMDwidth];
    float y[SIMDwidth];
    float z[SIMDwidth];
} VerticesCoordList;
typedef struct{
    int a[SIMDwidth];
    int b[SIMDwidth];
    int c[SIMDwidth];
    ...
} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];
    
```

ただし、SoA には、独立したメモリーストリームの参照の回数が増える欠点があります。配列 x,y,z を使用する計算は、3 つの別々のデータストリームを必要とします (例 4-20 を参照)。

そのため、プリフェッチとアドレス生成計算の数が増え、DRAM ページアクセスの効率も大きな影響を受けます。

これに対して、ハイブリッド SoA 手法は、AoS と SoA の 2 つの手法を組み合わせたものです (例 4-22 を参照)。この例の場合、別々のアドレスストリームが 2 つ生成され、参照されます。1 つのストリームは xxxx, yyyy, zzzz,

zzzz, ... で構成され、もう 1 つのストリームは aaaa, bbbb, cccc, aaaa, dddd, ... で構成されます。変数 x, y, z が常に一緒に使用されるとすれば (変数 a, b, c も一緒に使用されますが、変数 x, y, z と同時に使用されることはありません)、この方法によって、不要なデータのプリフェッチを排除できます。

ハイブリッド SoA 手法には、次のような利点があります。

- Y 垂直的な SIMD 計算を効率的に実行できるようにデータが構成される。
- Y AoS よりアドレス生成が簡単になり、アドレス生成の回数も少なくなる。
- Y ストリーム数が少ないため、DRAM ページミスが減少する。
- Y ストリーム数が少ないため、プリフェッチの回数が少なくなる。
- Y 同時に使用されるデータ要素が、キャッシュラインに効率良く格納される。

SIMD 技術の出現により、データ編成の選択が重要となりました。データに対して実行される操作の種類に基づいて、データ編成を慎重に選択する必要があります。インテル® Pentium® 4 プロセッサおよび将来のプロセッサでは、この重要性はますます高まります。アプリケーションによっては、従来のデータ配置では十分なパフォーマンスが得られないことがあります。アプリケーション開発者は、計算を効率良く行えるように、さまざまなデータ配置およびデータセグメントの形式を検討する必要があります。これには、特定のアプリケーションで AoS, SoA, ハイブリッド SoA を組み合わせて利用することも含まれます。

## 4.5.2 ストリップマイニング

ループのセクション化として知られるストリップマイニングは、メモリーのパフォーマンスを改善する手段で、ループの SIMD エンコーディングを可能にするループ変換テクニックです。この手法は、最初はベクトライザーのために導入されたもので、特定のベクトルマシン上の最大ベクトル長またはそれ以下のサイズのデータに対して各ベクトル操作が行うコードを生成できます。大きなループをより小さなセグメントやストリップに分割すると、ループ構造を変換して次のことを実現できます。

- Y データがアルゴリズムの複数のパスで再利用される場合、データキャッシュの時間と空間の局所性を向上させます。
- Y ループの反復回数を、1/(各ベクトルの長さ) に減らします (ベクトル長は、1 回の SIMD 操作で実行される操作の数に相当する)。インテル® ストリーミング SIMD 拡張命令の場合、ベクトル長 (ストリップ長) は 4 であるため、1 回のインテル® SSE 単精度浮動小数点 SIMD 操作あたり 4 つの浮動小数点データアイテムが処理されます。例えば、例 4-23 の場合を考えてみます。

例 4-23 ストリップマイニング前の疑似コード

```
typedef struct _VERTEX {
    float x, y, z, nx, ny, nz, u, v;
} Vertex_rec;
main()
{
    Vertex_rec v[Num];
    ....
    for (i=0; i<Num; i++) {
        Transform(v[i]);
    }
    for (i=0; i<Num; i++) {
        Lighting(v[i]);
    }
    ....
}
```

メインループは、変換 (Transform) と照明 (Lighting) 計算の 2 つの関数で構成されています。このメインループは、各オブジェクトに対し、変換ルーチン呼び出して一部のデータを更新し、次に照明計算ルーチン呼び出してさらにデータを処理しています。配列 v[Num] のサイズがキャッシュより大きい場合は、Transform(v[i]) の実行時に

キャッシュに格納された  $v[i]$  の座標は、 $\text{Lighting}(v[i])$  を実行する前にキャッシュから排出されてしまいます。そのため、 $v[i]$  をメインメモリーから 2 回フェッチすることになり、パフォーマンスが低下します。

例 4-24 では、ループのストリップマイニングが行われ、 $\text{strip\_size}$  サイズに分割されています。 $\text{strip\_size}$  の値は、配列  $v[\text{Num}]$  の  $\text{strip\_size}$  の要素がキャッシュ階層に納まるように選択されます。これによって、 $\text{Transform}(v[i])$  によってキャッシュに格納された要素  $v[i]$  は、 $\text{Lighting}(v[i])$  を実行する時点でまだキャッシュに残っています。したがって、ストリップマイニングが行われていないコードと比べて、パフォーマンスが向上します。

例 4-24 ストリップマイニング後のコード

```

MAIN()
{
    Vertex_rec v[Num];
    ....
    for (i=0; i < Num; i+=strip_size) {
        FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
            TRANSFORM(V[J]);
        }
        FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
            LIGHTING(V[J]);
        }
    }
}

```

### 4.5.3 ループ・ブロッキング

ループ・ブロッキングは、メモリー・パフォーマンスを最適化するもう 1 つの有効な手法です。ループ・ブロッキングの主な目的は、ストリップマイニングと同様に、キャッシュミスをできるだけ減らすことです。この手法は、メモリー領域全体をシーケンシャルに横断するのではなく、特定のメモリー領域を小さいブロックに変換します。各ブロックは、データを最大限に再利用できるように、特定の計算に使用されるすべてのデータがキャッシュに納まる程度に小さくなります。ループ・ブロッキングを、2 次元またはそれ以上の次元におけるストリップマイニングと見なすことができます。例えば、例 4-23 のコードと図 4-4 のアクセスパターンについて考えてみます。2 次元配列 A は、まず  $j$  (列) 方向に、その後、 $i$  (行) 方向に参照されます (列優先順)。配列 B は逆の順序で参照されます (行優先順)。メモリー配置が列優先順であると仮定します。したがって、このコードの配列 A と B のアクセスのストライドは、それぞれ、1 と MAX になります。

例 4-25 ループ・ブロッキング

```

A. オリジナルループ
float A[MAX, MAX], B[MAX, MAX]
for (i=0; i < MAX; i++) {
    for (j=0; j < MAX; j++) {
        A[i,j] = A[i,j] + B[j, i];
    }
}

B. ブロッキング後の変換されたループ
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i < MAX; i+=block_size) {
    for (j=0; j < MAX; j+=block_size) {
        for (ii=i; ii < i+block_size; ii++) {
            for (jj=j; jj < j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}

```

内側ループの最初の反復では、配列 B にアクセスするたびにキャッシュミスが発生します。配列 A の 1 行のサイズ (つまり  $A[2][0:MAX-1]$ ) が十分に大きい場合、2 回目の反復が始まるまでに、配列 B にアクセスするたびに必ずキャッシュミスが発生します。例えば、float 型の変数は 4 バイト、各キャッシュラインは 32 バイトであるため、最初の反復で、 $B[0][0]$  が参照されると、 $B[0][0:7]$  を含むキャッシュラインがキャッシュに格納されます。キャッシュの容量には制限があるため、このラインは、内側ループが終わりに達する前に、競合ミスのために排出されます。外側ループの次の反復では、 $B[0][1]$  を参照すると、次のキャッシュミスが発生します。このように、配列 B の各要素が参照されるたびに、1 回のキャッシュミスが起こります。つまり、配列 B については、キャッシュ内のデータは全く再利用されていません。

キャッシュのサイズを考慮してループをブロック化すれば、この状態を避けることができます。図 4-4 では、ループ・ブロック係数として `block_size` が選択されています。block\_size が 8 の場合、各配列の変換後のブロックのサイズは 8 キャッシュラインになります (各ラインが 32 バイトに相当)。内側ループの最初の反復では、 $A[0][0:7]$  と  $B[0][0:7]$  がキャッシュに取り込まれます。 $B[0][0:7]$  は、外側ループの最初の反復ですべて参照されます。したがって、 $B[0][0:7]$  のキャッシュミスは、元のアルゴリズムでは 8 回発生していましたが、ループ・ブロックの最適化を適用した後のアルゴリズムでは 1 回しか発生しません。図 4-5 に示すように、配列 A と配列 B は、A と B の変換後の 2 つのブロックの合計サイズがキャッシュのサイズより小さくなるように、小さな矩形のブロックに変換されます。これによって、データを最大限に再利用できます。

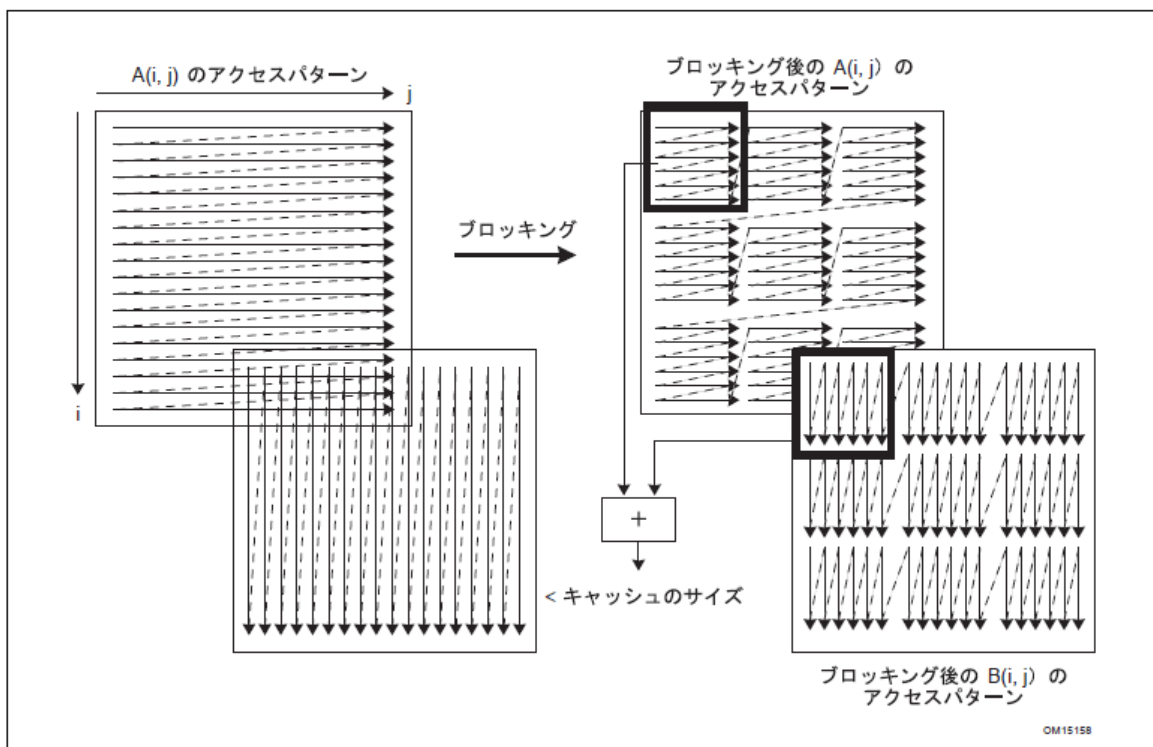


図 4-5 ループ・ブロック化のアクセスパターン

このように、ループ・ブロック化手法を適用すると、すべての冗長なキャッシュミスは排除できます。MAX が非常に大きい場合は、ループ・ブロック化によって、DTLB (データ・トランレーション・ルックアサイド・バッファ) のミスによるペナルティーも軽減できます。また、この最適化手法は、キャッシュとメモリのパフォーマンスを向上させるだけでなく、外部バス帯域幅の節約にもなります。

## 4.6 命令の選択

この節では、タスクを実行する命令を選択するガイドラインについて説明します。

SIMD 計算の障害の 1 つは、データに依存する分岐の存在です。条件付き移動を使用することで、データに依存する分岐を排除できます。例 4-26 に示すように、マスクされた比較と論理演算子を使用して、SIMD 計算で条件付き

移動をエミュレートします。インテル® SSE4.1 では、ループ内のデータ依存分岐をベクトル化可能にするパックド・ブレンディング命令が提供されています。

例 4-26 条件付き移動のエミュレーション

```

高レベルなコード:
__declspec(align(16)) short A[MAX_ELEMENT], B[MAX_ELEMENT], C[MAX_ELEMENT],
D[MAX_ELEMENT],
E[MAX_ELEMENT];
for (i=0; i<MAX_ELEMENT; i++) {
    if (A[i] > B[i]) {
        C[i] = D[i];
    } else {
        C[i] = E[i];
    }
}
}
インテル® MMX アセンブリ・コードは反復ごとに 4 つの short を処理:
    xor eax, eax
top_of_loop:
    movq mm0, [A + eax]
    pcmpgtw xmm0, [B + eax]          ; 比較マスクを生成
    movq mm1, [D + eax]
    pand mm1, mm0                   ; A<B の要素をドロップ
    pandn mm0, [E + eax]            ; A>B の要素をドロップ
    por mm0, mm1                    ; 単一ワードを生成
    movq [C + eax], mm0
    add eax, 8
    cmp eax, MAX_ELEMENT*2
    jle top_of_loop
インテル® SSE4.1 アセンブリ・コードは反復ごとに 8 つの short を処理:
    xor eax, eax
top_of_loop:
    movdqq xmm0, [A + eax]
    pcmpgtw xmm0, [B + eax]          ; 比較マスクを生成
    movdqa xmm1, [E + eax]
    pblendv xmm1, [D + eax], xmm0 ;
    movdqa [C + eax], xmm1          ;
    add eax, 16
    cmp eax, MAX_ELEMENT*2
    jle top_of_loop

```

レジスターの 1 つのインスタンスを複数の命令が参照する場合は、それらの命令をできるだけ近くにまとめます。ただし、データを参照する側の命令を、データを生成する側の命令の近くにスケジューリングしてはいけません。

### 4.6.1 SIMD の最適化とマイクロアーキテクチャー

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、およびインテル® Core™ Duo プロセッサーのマイクロアーキテクチャーは、Intel NetBurst® マイクロアーキテクチャーとは異なります。以下の項では、インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーを対象とした SIMD コードの最適化について説明します。

以下の命令のレジスター-レジスター形式では、インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーのパフォーマンスがインテル® Pentium® M プロセッサーと比べて向上しています。それは、命令を構成するマイクロオペレーション (μop) が 3 つではなく 2 つであるためです。これに該当する命令は、unpcklps、unpckhps、packsswb、packuswb、packssdw、pshufd、shuffps、shuffpd です。

**推奨事項:** インテル® Core™ Solo プロセッサーおよびインテル® Core™ Duo プロセッサーを対象にコードを生成する場合、3 つ以上のマイクロオペレーション (μop) からなる命令よりも、2 つのマイクロオペレーション (μop) からなる命令を優先します。

インテル® Core™ マイクロアーキテクチャーでは一般に、レイテンシーとスループットの面で従来のマイクロアーキテクチャーよりも効率良く SIMD 命令を実行します。大部分の 128 ビット SIMD 命令では、スループットが 1 サイクルです (シャッフル/パック/アンパック操作を除きます)。Intel NetBurst® マイクロアーキテクチャーに対するインテル® Core™ マイクロアーキテクチャーの関係についても、同じことが当てはまります。

拡張版インテル® Core™ マイクロアーキテクチャーでは、専用の 128 ビット・シャッフルと基数 16 の除算器ハードウェアが提供されています。これらの機能やインテル® SSE4.1 命令を利用すると、128 ビット SIMD 命令を使用したベクトル化がさらに効率的かつ効果的なものになります。

**推奨事項:** インテル® Core™ マイクロアーキテクチャーと拡張版インテル® Core™ マイクロアーキテクチャーの 128 ビット SIMD ハードウェアの普及に伴い、インテル® MMX® 命令で作成された整数 SIMD コードについては、128 ビット SIMD 命令を使用した効率的な実装への移行を検討すべきです。

## 4.7 開発の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常に動作することを確認した後に、アプリケーションをチューニングする最も良い方法は、システム上でアプリケーションを実行しながら、プロファイラーを使用してアプリケーションのパフォーマンスを測定することです。インテル® VTune™ Amplifier は、パフォーマンスを向上させるためアプリケーションのどこを修正したら良いかを特定するのに役立ちます。インテル® VTune™ Amplifier は、パフォーマンスを最適化する各種の工程で使用できます。詳細は、付録 A.3.1 の「インテル® VTune™ Amplifier」を参照してください。最適化手法を適用するたびに、パフォーマンスがどの程度向上したかをチェックして、どこを最適化すればパフォーマンスが大きく向上するかを確認すると良いでしょう。

この章では、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSE4.1 で利用できる SIMD (single-instruction, multiple-data) 浮動小数点命令を最適化する際の規則について説明します。また、単精度および倍精度の SIMD 浮動小数点アプリケーション向けの最適化手法についても、具体例を示して説明します。

## 6.1 SIMD 浮動小数点コード向けの一般的な規則

この節で述べる規則と提案に従うことで、SIMD 浮動小数点命令を使用した浮動小数点コードの最適化が容易になります。一般に、効率の良い SIMD 浮動小数点コードを作成するには、ポート利用率を理解しバランスを取ることが重要です。次に基本的な規則を示します。

- Y 第 3 章 と第 4 章 に示したすべてのガイドラインに従います。
- Y 高いパフォーマンスを達成するため、例外をマスクします。例外をマスクしないと、ソフトウェアのパフォーマンスが低下します。
- Y デノーマル状態とアンダーフロー状態を処理するペナルティーを避け、高いパフォーマンスを実現するためゼロ・フラッシュ・モードと DAZ モードを利用します。
- Y 精度を上げるため、逆数命令に続けて反復処理を行います。逆数命令では精度は低下しますが、実行速度はかなり速くなります。次の点に注意してください。
  - 精度の低下が許容される場合には、反復処理を用いずに逆数命令を使用します。
  - 完全に近い精度が必要な場合は、ニュートン・ラフソン反復法を適用します。
  - 完全な精度が必要な場合は、より高い精度の得られる除算と平方根命令を使用します。ただし速度は低下します。

## 6.2 計画上の留意事項

既存のアプリケーションに導入する場合でも、新しくアプリケーションを作成する場合でも、SIMD 浮動小数点命令を使用して最適なパフォーマンスを得るには、考慮すべきいくつかの課題があります。一般に、最適化の候補を選ぶときは、大量の浮動小数点計算が行われているコードセグメントを特定します。また、キャッシュ・アーキテクチャーの効果的な使用も検討する必要があります。

以下の節では、実装前によく問われる次のような疑問に答えていきます。

- Y 並列処理やキャッシュの使用効率を向上させるデータレイアウトは可能であるか?
- Y コードのどの部分が SIMD 浮動小数点命令の効果を得られるか?
- Y 現在のアルゴリズムは SIMD 浮動小数点命令に最適であるか?
- Y 対象となるコードは浮動小数点を多用しているか?
- Y 単精度浮動小数点演算と倍精度浮動小数点演算では、どちらが十分な数値範囲と精度を得られるか?
- Y ゼロ・フラッシュ・モードや DAZ モードを有効にすると、計算結果に影響はあるか?
- Y SIMD 浮動小数点レジスターを効率良く利用できるようにデータが配置されているか?
- Y 対象となるアプリケーションは、SIMD 浮動小数点命令を実行できないプロセッサをターゲットとしているか?

関連情報: 4.2 節「SIMD プログラミングへのコード変換に関する留意事項」も参照してください。



## 6.3 x87 浮動小数点と SIMD 浮動小数点との併用

SIMD 浮動小数点演算に使用される XMM レジスターは、独立したレジスターであり、既存の x87 浮動小数点スタックにはマッピングされません。そのため、SIMD 浮動小数点コードは、x87 浮動小数点コードや、64 ビット SIMD 整数コードと混在して利用できます。

インテル® Core™ マイクロアーキテクチャーでは、128 ビット SIMD 整数命令は 64 ビット SIMD 整数命令よりもはるかに高い効率を発揮します。ソフトウェアは、可能な限り SIMD 浮動小数点命令と SIMD 整数命令を XMM レジスターとともに使用すべきです。

## 6.4 スカラー浮動小数点コード

SIMD レジスターの最下位の要素しか演算しない SIMD 浮動小数点命令があります。これらの命令は、「スカラー命令」と呼ばれます。スカラー命令を用いると、汎用の浮動小数点演算に XMM レジスターを使用できるようになります。

パフォーマンス面から見ると、スカラー浮動小数点コードは、x87 浮動小数点コードと同等かそれ以上であり、次の利点があります。

- Y SIMD 浮動小数点コードはフラットなレジスターモデルを使用するのにに対し、x87 浮動小数点コードではスタックモデルが使用されます。スカラー浮動小数点コードを使用することで、FXCH 命令は必要なくなります。FXCH 命令は、インテル® Pentium® 4 プロセッサで若干の制約があります。
- Y インテル® MMX® テクノロジーと混在させてもペナルティーが生じません。
- Y ゼロ・フラッシュ・モードが利用できます。
- Y x87 浮動小数点よりもレイテンシーが小さくなります。

スカラー浮動小数点命令を使用する場合、データをベクトル形式で表現する必要はありません。ただし、アライメント、スケジューリング、命令の選択など、第 3 章と第 4 章で述べた最適化手法は適用する必要があります。

## 6.5 データアライメント

SIMD 浮動小数点データは 16 バイトにアライメントされます。アライメントの合っていない 128 ビット SIMD 浮動小数点データの参照では、MOVUPS 命令 (Move Unaligned Packed Single) や MOVUPD 命令 (Move Unaligned Packed Double) を使用しないと例外が発生します。非アライメント命令を使用すると、対象となるデータがアライメントされているかどうかに関係なく、アライメント命令と比べてパフォーマンスが劣ります。

関連情報: 4.4 節「スタックとデータアライメント」を参照してください。

### 6.5.1 データ配置

インテル® SSE にもインテル® SSE2 にも SIMD アーキテクチャーが適用されるため、SIMD レジスターを最大限に活用できるようにデータをうまく配置すれば、最適なパフォーマンスが得られます。これには、処理するデータが隙間なく連続して並んでいることが必要ですが、うまくデータを配置することでキャッシュミスが減少します。適切にデータを配置することで、インテル® SSE ではデータ・スループットが 4 倍になり、インテル® SSE2 ではデータ・スループットが 2 倍になる可能性もあります。パフォーマンス改善は、インテル® SSE (MOVAPS 命令) では 4 つの 32 ビット・データ要素を 128 ビット・ロード命令で XMM レジスターへロードできることからもたらされます。同様に、インテル® SSE2 (MOVAPD 命令) を使用することで、2 つの 64 ビット・データ要素を 128 ビット・ロード命令で XMM レジスターへロードできます。

データ配置に関する推奨事項は、4.4 節「スタックとデータアライメント」を参照してください。データの構造や配置によってはミスアライメントの問題が発生する場合がありますが、複製とパディングによって解決できます。このような手法はデータ領域を増加させますが、アライメントの合っていないデータアクセスのペナルティーを排除できます。

アプリケーション (3D ジオメトリなど) によっては、従来のデータ配置を一部変更しないと、SIMD レジスターと並列処理を最大限に活用できません。従来、データレイアウトには、複数の構造体から 1 つの配列を構成する、いわゆる構造体配列 (AoS) が用いられてきました。こうしたアプリケーションで SIMD レジスターを最大限に活用するため、新たなデータレイアウトが提案されています。それは、複数の配列から 1 つの構造体を構成する、いわゆる配列構造体 (SoA) と呼ばれ、より高いパフォーマンスが得られます。

### 6.5.1.1 垂直計算と水平計算

インテル® SSE とインテル® SSE2 の浮動小数点演算命令の大半は、並列データ要素の垂直データ処理において高いパフォーマンスを発揮します。つまり、デスティネーションの各要素は、同じ垂直位置にあるソース要素によって行われた算術演算の結果であることを意味します (図 6-1)。

またインテル® SSE とインテル® SSE2 では、並列データ要素の同種算術演算を補助するため、データ移動命令 (SHUFPS, UNPCKLPS, UNPCKHPS, MOVLHPS, MOVHLPS など) を提供してデータ要素の水平移動を容易にしています。

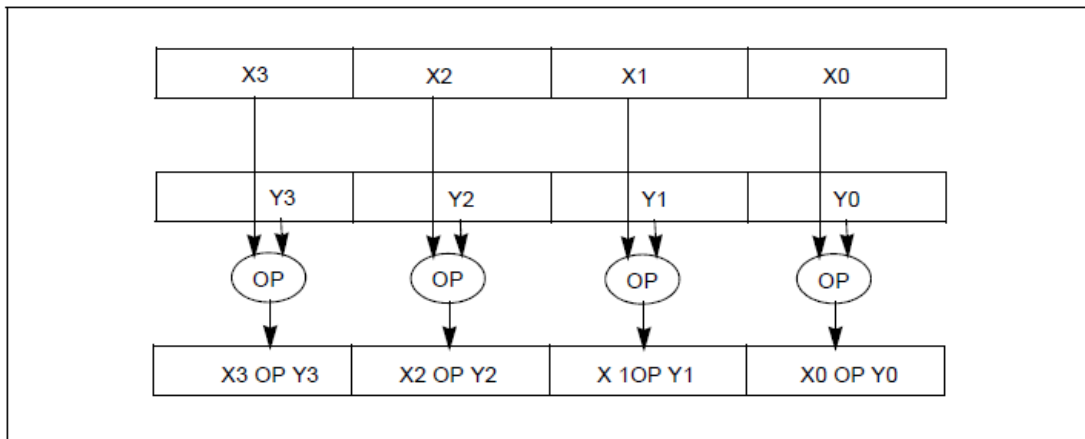


図 6-1 並列データ要素の同種演算

構造化されたデータの編成は、SIMD プログラミングの効率とパフォーマンスに大きな影響を与えます。これについては、データ構造を編成する 2 つの一般的な形式によって説明できます。

- Y 構造体配列 (AoS): これは、複数のデータ構造を 1 つの配列に並べたものです。データ構造内の各メンバーはスカラーであり、この様子を図 6-2 に示します。反復計算シーケンスは通常、配列の各要素、すなわちデータ構造に適用されます。構造体のスカラーメンバーに対する計算シーケンスは、各反復内で同種演算にならない可能性が高くなります。AoS は一般に、水平計算モデルに関連付けられます。

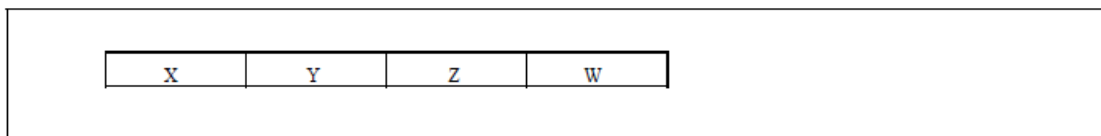


図 6-2 水平計算モデル

- Y 配列構造体 (SoA): データ構造体の各メンバーは配列であり、配列の各要素はスカラーです。この様子を表 6-1 に示します。反復計算シーケンスはスカラー要素に適用され、同じ構造体メンバー内の連続した反復間で同種演算を容易に実行できます。したがって、SoA は通常、垂直計算モデルに関連付けられます。

表 6-1 頂点データを SoA 形式で表現した例

Vx array	X1	X2	X3	X4	.....	Xn
Vy array	Y1	Y2	Y3	Y4	.....	Yn
Vz array	Z1	Z2	Z3	Y4	.....	Zn
Vw array	W1	W2	W3	W4	.....	Wn

SoA 配列への垂直計算に SIMD 命令を使用すると、AoS および水平計算よりも高い効率とパフォーマンスを達成できます。これは各ベクトルに対するドット積演算で見られます。図 6-3 に、SoA 配列のドット積演算を示します。

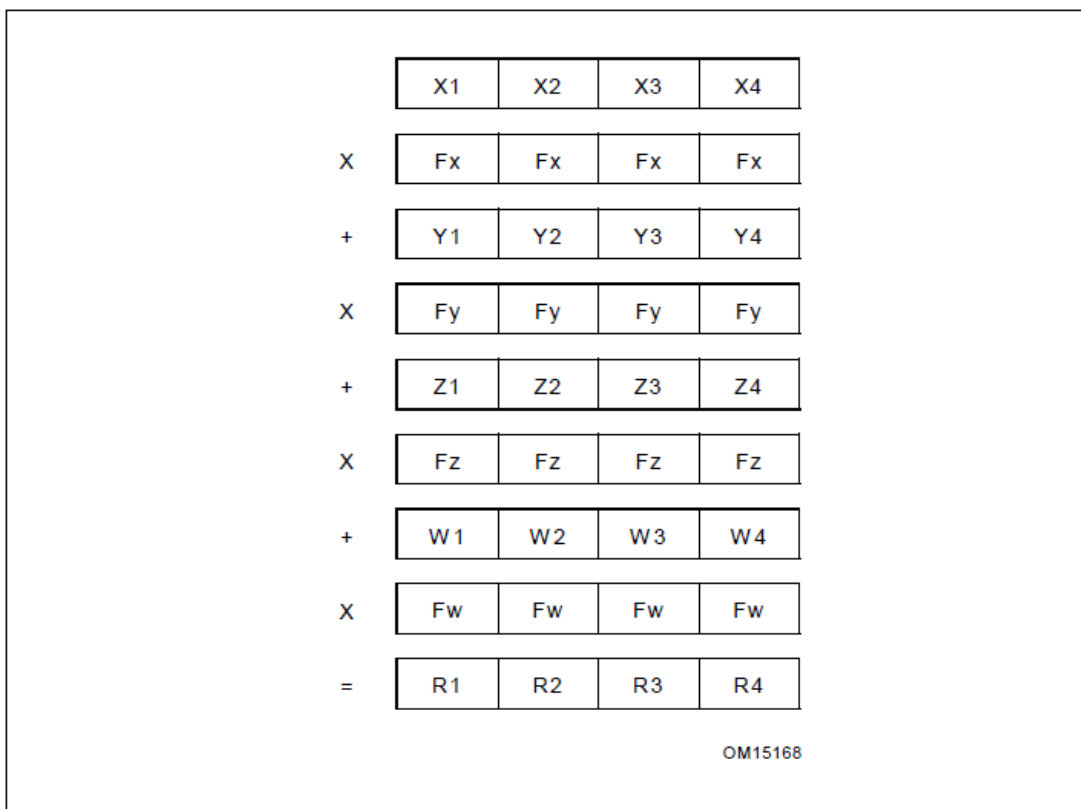


図 6-3 ドット積演算

図 6-1 を見ると、インテル® SSE のみを使用してデータを AoS として編成した場合、7 つの命令で 1 つの計算結果が得られることが分かります。つまり、計算結果を 4 つ得るには 28 個の命令が必要になります。

例 6-1 水平 (xyz, AoS) 計算の擬似コード

```

mulps ; x*x', y*y', z*z'
movaps ; 次のステップが上書きするので reg->reg 移動
shufps ; a, b, c, d から b, a, d, c を取得
addps ; a+b, a+b, c+d, c+d の取得
movaps ; reg->reg 移動
shufps ; 前の addps から c+d, c+d, a+b, a+b を取得
addps ; a+b+c+d, a+b+c+d, a+b+c+d, a+b+c+d の取得
    
```

次に、データが SoA として編成される場合を考えてみます。例 6-2 を見ると、5 つの命令で 4 つの計算結果が得られることが分かります。

例 6-2 垂直 (xxxx, yyyy, zzzz, SoA) 計算の擬似コード

```

mulps ; 4 頂点の 4 つの x 要素すべてに x*x' を計算
mulps ; 4 頂点の 4 つの y 要素すべてに y*y' を計算
mulps ; 4 頂点の 4 つの z 要素すべてに z*z' を計算
addps ; x*x' + y*y'
addps ; x*x'+y*y'+z*z'
    
```

4 つのコンポーネントを含むレジスターを最も効率良く使用するため、データを SoA 形式に再編成すると、スループットが増加します。その結果、使用する命令のパフォーマンスがより高まります。

この単純な例から分かるように、垂直計算では SIMD レジスター幅をすべて利用して 4 つの計算結果が得られます (ただし、状況によっては計算結果が異なることがあります)。データ構造体が垂直計算に向かない形式でも、実行処理と並行してその構造体を再編成すると SIMD レジスターを活用しやすくなります。このデータ再編成操作を「スウィズリング」と呼び、その逆の操作を「デスウィズリング」と呼びます。

6.5.1.2 データ・スウィズリング

SoA 形式から AoS 形式へのデータのスウィズリングは、3D ジオメトリ、ビデオ、画像処理など多くのアプリケーションに適用できます。浮動小数点データと整数データの操作には、2 つの異なるスウィズリング手法を適用できます。例 6-3 に、SHUFPS、MOVLHPS、MOVHLPS 命令を使用するスウィズリング関数を示します。

例 6-3 SHUFPS、MOVLHPS、MOVHLPS 命令を使用したデータのスウィズリング

```

typedef struct _VERTEX_AOS {
    float x, y, z, color;
} Vertex_aos; // AoS 構造体の宣言
typedef struct _VERTEX_SOA {
    float x[4], float y[4], float z[4];
    float color[4];
} Vertex_soa; // SoA 構造体の宣言
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
    // in の内容: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
    // スウィズル XYZW --> XXXX
    asm {
        mov ebx, in // 構造体のアドレスを取得
        mov edx, out
        movaps xmm1, [ebx ] // x4 x3 x2 x1
        movaps xmm2, [ebx + 16] // y4 y3 y2 y1
        movaps xmm3, [ebx + 32] // z4 z3 z2 z1
        movaps xmm4, [ebx + 48] // w4 w3 w2 w1
        movaps xmm7, xmm4 // xmm7= w4 z4 y4 x4
        movhlps xmm7, xmm3 // xmm7= w4 z4 w3 z3
        movaps xmm6, xmm2 // xmm6= w2 z2 y2 x2
        movlhps xmm3, xmm4 // xmm3= y4 x4 y3 x3
        movhlps xmm2, xmm1 // xmm2= w2 z2 w1 z1
        movlhps xmm1, xmm6 // xmm1= y2 x2 y1 x1
        movaps xmm6, xmm2 // xmm6= w2 z2 w1 z1
        movaps xmm5, xmm1 // xmm5= y2 x2 y1 x1
        shufps xmm2, xmm7, 0xDD // xmm2= w4 w3 w2 w1 => v4
        shufps xmm1, xmm3, 0x88 // xmm1= x4 x3 x2 x1 => v1
        shufps xmm5, xmm3, 0xDD // xmm5= y4 y3 y2 y1 => v2
        shufps xmm6, xmm7, 0x88 // xmm6= z4 z3 z2 z1 => v3
        movaps [edx], xmm1 // store X
        movaps [edx+16], xmm5 // store Y
        movaps [edx+32], xmm6 // store Z
        movaps [edx+48], xmm2 // store W
    }
}
    
```

例 6-4 に、整数ドメインに SIMD 命令を使用する同様のデータ・スウィズリング・アルゴリズムを示します。

例 6-4 UNPCKxxx 命令を使用したデータのスウィズリング

```
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in の内容: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// スウィズル XYZW --> XXXX
asm {
    mov ebx, in // 構造体のアドレスを取得
    mov edx, out
    movdqa xmm1, [ebx + 0*16] //w0 z0 y0 x0
    movdqa xmm2, [ebx + 1*16] //w1 z1 y1 x1
    movdqa xmm3, [ebx + 2*16] //w2 z2 y2 x2
    movdqa xmm4, [ebx + 3*16] //w3 z3 y3 x3
    movdqa xmm5, xmm1
    punpckldq xmm1, xmm2 // y1 y0 x1 x0
    punpckhdq xmm5, xmm2 // w1 w0 z1 z0
    movdqa xmm2, xmm3
    punpckldq xmm3, xmm4 // y3 y2 x3 x2
    punpckldq xmm2, xmm4 // w3 w2 z3 z2
    movdqa xmm4, xmm1
    punpcklqdq xmm1, xmm3 // x3 x2 x1 x0
    punpckhqdq xmm4, xmm3 // y3 y2 y1 y0
    movdqa xmm3, xmm5
    punpcklqdq xmm5, xmm2 // z3 z2 z1 z0
    punpckhqdq xmm3, xmm2 // w3 w2 w1 w0
    movdqa [edx+0*16], xmm1 //x3 x2 x1 x0
    movdqa [edx+1*16], xmm4 //y3 y2 y1 y0
    movdqa [edx+2*16], xmm5 //z3 z2 z1 z0
    movdqa [edx+3*16], xmm3 //w3 w2 w1 w0
}
}
```

新しいマイクロアーキテクチャーでは、MOVLPS/MOVHPS 命令を使用して各ベクトルの半分をロードする手法よりも、例 6-3 の手法 (16 バイトをロード、SHUFPS を使用して、XMM レジスターの半分をコピーする) が推奨されます。これは、MOVLPS/MOVHPS 命令を使用して 8 バイトをロードすると、コードの依存関係が生じ、実行エンジンのスループットが低下する可能性があるためです。

例 6-3 と例 6-4 に関するパフォーマンス上の考慮事項は、各マイクロアーキテクチャーの特性に依存しています。例えばインテル® Core™ マイクロアーキテクチャーでは、SHUFPS を実行すると、PUNPCKxxx 命令よりも低速になる傾向があります。拡張版インテル® Core™ マイクロアーキテクチャーでは、SHUFPS 命令と PUNPCKxxx 命令はすべて、128 ビット・シャッフル実行ユニットによって 1 サイクルのスループットで実行されます。次に重要なことは、PUNPCKxxx を実行できるポートは 1 つしかありませんが、MOVLHPS/MOVHLPS 命令は複数のポートで実行でき、インテル® Core™ マイクロアーキテクチャーでは、SIMD 命令を実行できるポートが 3 つあるため、いずれの手法によるパフォーマンスも従来のマイクロアーキテクチャーに比べて向上しています。拡張版インテル® Core™ マイクロアーキテクチャーでは、128 ビット・シャッフル・ユニットが採用されたことにより、いずれの手法もさらにパフォーマンスが向上しています。

### 6.5.1.3 データ・デスウィズリング

デスウィズリングでは、XXXX、YYYY、ZZZZ が XYZ として再配置されてからメモリーに格納されるように、SoA 形式を AoS 形式に戻す必要があります。例 6-5 に、浮動小数点データ向けのデスウィズリング関数を示します。

例 6-5 単精度 SIMD データのデスウィズリング

```

void deswizzle_asm(Vertex_soa *in, Vertex_aos *out)
{
    __asm {
        mov ecx, in // 構造体のアドレスをロード
        mov edx, out
        movaps xmm0, [ecx] //x3 x2 x1 x0
        movaps xmm1, [ecx + 16] //y3 y2 y1 y0
        movaps xmm2, [ecx + 32] //z3 z2 z1 z0
        movaps xmm3, [ecx + 48] //w3 w2 w1 w0
        movaps xmm5, xmm0
        movaps xmm7, xmm2
        unpcklps xmm0, xmm1 // y1 x1 y0 x0
        unpcklps xmm2, xmm3 // w1 z1 w0 z0
        movdqa xmm4, xmm0
        movlhps xmm0, xmm2 // w0 z0 y0 x0
        movhlps xmm4, xmm2 // w1 z1 y1 x1
        unpckhps xmm5, xmm1 // y3 x3 y2 x2
        unpckhps xmm7, xmm3 // w3 z3 w2 z2
        movdqa xmm6, xmm5
        movlhps xmm5, xmm7 // w2 z2 y2 x2
        movhlps xmm6, xmm7 // w3 z3 y3 x3
        movaps [edx+0*16], xmm0 //w0 z0 y0 x0
        movaps [edx+1*16], xmm4 //w1 z1 y1 x1
        movaps [edx+2*16], xmm5 //w2 z2 y2 x2
        movaps [edx+3*16], xmm6 //w3 z3 y3 x3
    }
}

```

例 6-6 に、SIMD 整数命令を使用する同様のデスウィズリング関数を示します。どちらの手法でも、16 バイトをロードし、レジスター内で水平データ移動を行っています。この方法は、MOVLPS と MOVHPS を使用して XMM レジスターの半分の 8 バイトをストアする手法よりも効率的です。

例 6-6 SIMD 整数命令を使用したデータのデスウィズリング

```

void deswizzle_rgb(Vertex_soa *in, Vertex_aos *out)
{
    //--- rgb をデスウィズリング ---
    // 前提: xmm1=rrrr, xmm2=gggg, xmm3=bbbb, xmm4=aaaa
    __asm {
        mov ecx, in // 構造体のアドレスをロード
        mov edx, out
        movdqa xmm0, [ecx] // load r4 r3 r2 r1 => xmm1
        movdqa xmm1, [ecx+16] // load g4 g3 g2 g1 => xmm2
        movdqa xmm2, [ecx+32] // load b4 b3 b2 b1 => xmm3
        movdqa xmm3, [ecx+48] // load a4 a3 a2 a1 => xmm4
        // デスウィズリングを開始
        movdqa xmm5, xmm0
        movdqa xmm7, xmm2
        punpckldq xmm0, xmm1 // g2 r2 g1 r1
        punpckldq xmm2, xmm3 // a2 b2 a1 b1
        movdqa xmm4, xmm0
        punpcklqdq xmm0, xmm2 // a1 b1 g1 r1 => v1
        punpckhqdq xmm4, xmm2 // a2 b2 g2 r2 => v2
        punpckhdq xmm5, xmm1 // g4 r4 g3 r3
        punpckhdq xmm7, xmm3 // a4 b4 a3 b3
        movdqa xmm6, xmm5
        punpcklqdq xmm5, xmm7 // a3 b3 g3 r3 => v3
        punpckhqdq xmm6, xmm7 // a4 b4 g4 r4 => v4
    }
}

```

```

movdqa [edx], xmm0 // v1
movdqa [edx+16], xmm4 // v2
movdqa [edx+32], xmm5 // v3
movdqa [edx+48], xmm6 // v4
// デスウィズリングを終了
}
}
    
```

### 6.5.1.4 インテル® SSE による水平加算

一般に垂直計算の方が水平計算よりも SIMD のパフォーマンスをうまく引き出せますが、場合によってはコード中で水平演算を使用しなければならないことがあります。

MOVLHPS/MOVLHPS 命令とシャッフル命令を使用すると、データの水平加算が可能となります。例えば、128 ビットのレジスターが 4 つあり、それぞれを水平加算する一方、最終的な計算結果を 1 つのレジスターに格納する場合、MOVLHPS/MOVLHPS の命令を使用して各レジスターの上位部分と下位部分を合わせる必要があります。このような操作によって、垂直加算が可能になります。部分的な水平総和から全体の総和も簡単に求められます。

図 6-4 に、MOVLHPS/MOVLHPS 命令を使用した水平加算を示します。この操作を行うコードを例 6-7 と例 6-8 に示します。

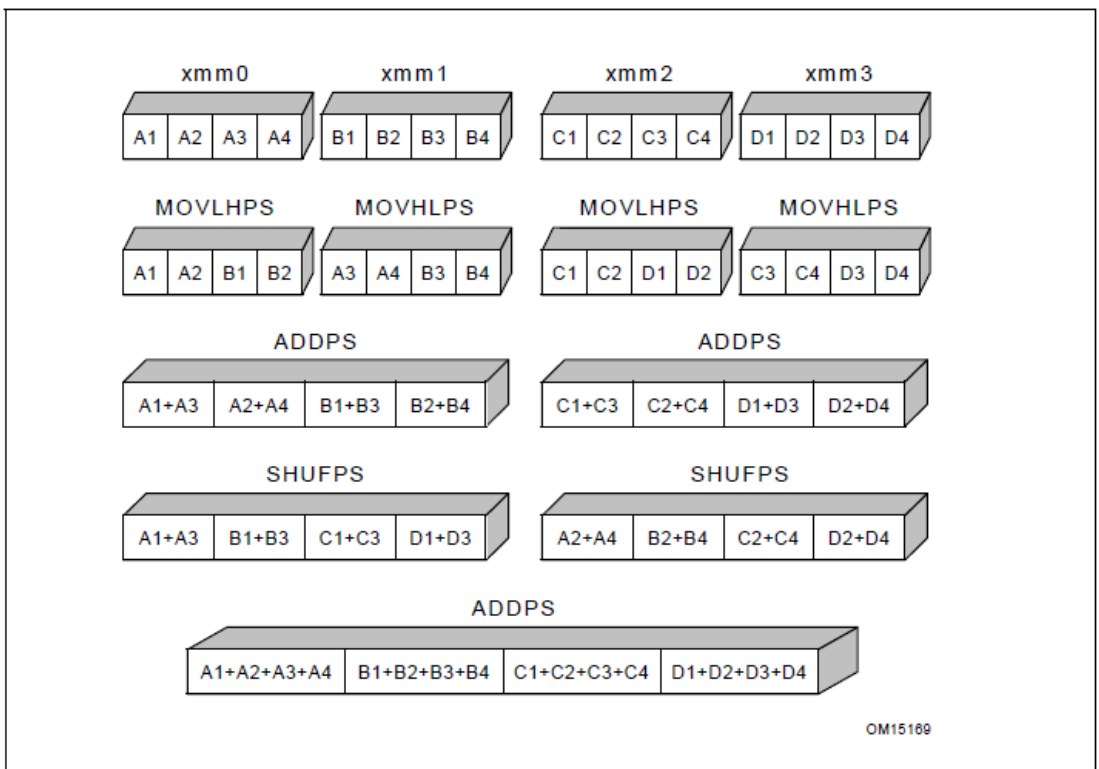


図 6-4 MOVHLPS/MOVLHPS を使用した水平加算

例 6-7 MOVHLPS/MOVLHPS を使用した水平加算

```

void horiz_add(Vertex_soa *in, float *out) {
__asm {
    mov ecx, in // 構造体のアドレスをロード
    mov edx, out
    movaps xmm0, [ecx] // load A1 A2 A3 A4 => xmm0
    movaps xmm1, [ecx+16] // load B1 B2 B3 B4 => xmm1
    movaps xmm2, [ecx+32] // load C1 C2 C3 C4 => xmm2
    movaps xmm3, [ecx+48] // load D1 D2 D3 D4 => xmm3
// 水平加算を開始
    movaps xmm5, xmm0 // xmm5= A1, A2, A3, A4
    movlhps xmm5, xmm1 // xmm5= A1, A2, B1, B2
    movhlps xmm1, xmm0 // xmm1= A3, A4, B3, B4
    addps xmm5, xmm1 // xmm5= A1+A3, A2+A4, B1+B3, B2+B4
    movaps xmm4, xmm2
    movlhps xmm2, xmm3 // xmm2= C1, C2, D1, D2
    movhlps xmm3, xmm4 // xmm3= C3, C4, D3, D4
    addps xmm3, xmm2 // xmm3= C1+C3, C2+C4, D1+D3, D2+D4
    movaps xmm6, xmm3 // xmm6= C1+C3, C2+C4, D1+D3, D2+D4
    shufps xmm3, xmm5, 0xDD
        //xmm6=A1+A3, B1+B3, C1+C3, D1+D3
    shufps xmm5, xmm6, 0x88
        // xmm5= A2+A4, B2+B4, C2+C4, D2+D4
    addps xmm6, xmm5 // xmm6= D, C, B, A
// 水平加算を終了
    movaps [edx], xmm6
}
}

```

例 6-8 MOVHLPS/MOVLHPS と組込み関数を併用した水平加算

```

void horiz_add_intrin(Vertex_soa *in, float *out)
{
__m128 v, v2, v3, v4;
__m128 tmm0, tmm1, tmm2, tmm3, tmm4, tmm5, tmm6;
// 一時変数
    tmm0 = _mm_load_ps(in->x); // tmm0 = A1 A2 A3 A4
    tmm1 = _mm_load_ps(in->y); // tmm1 = B1 B2 B3 B4
    tmm2 = _mm_load_ps(in->z); // tmm2 = C1 C2 C3 C4
    tmm3 = _mm_load_ps(in->w); // tmm3 = D1 D2 D3 D4
    tmm5 = tmm0; // tmm0 = A1 A2 A3 A4
    tmm5 = _mm_movehl_ps(tmm5, tmm1); // tmm5 = A1 A2 B1 B2
    tmm1 = _mm_movelh_ps(tmm1, tmm0); // tmm1 = A3 A4 B3 B4
    tmm5 = _mm_add_ps(tmm5, tmm1); // tmm5 = A1+A3 A2+A4 B1+B3 B2+B4
    tmm4 = tmm2;
    tmm2 = _mm_movehl_ps(tmm2, tmm3); // tmm2 = C1 C2 D1 D2
    tmm3 = _mm_movelh_ps(tmm3, tmm4); // tmm3 = C3 C4 D3 D4
    tmm3 = _mm_add_ps(tmm3, tmm2); // tmm3 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = tmm3; // tmm6 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = _mm_shuffle_ps(tmm3, tmm5, 0xDD);
        // tmm6 = A1+A3 B1+B3 C1+C3 D1+D3
    tmm5 = _mm_shuffle_ps(tmm5, tmm6, 0x88);
        // tmm5 = A2+A4 B2+B4 C2+C4 D2+D4
    tmm6 = _mm_add_ps(tmm6, tmm5);
        // tmm6 = A1+A2+A3+A4 B1+B2+B3+B4
        // C1+C2+C3+C4 D1+D2+D3+D4
    _mm_store_ps(out, tmm6);
}

```



## 6.5.2 CVTTPS2PI/CVTTSS2SI 命令の使用

CVTTPS2PI 命令および CVTTSS2SI 命令は、暗黙的に「切り捨て/絶対値の小さくなる方向への丸め」という制御モードでエンコードされます。これは、MXCSR レジスターで指定される丸め制御モードよりも優先されます。これにより、丸め制御モードを「最も近い値へ丸め」から「切り捨て/絶対値の小さくなる方向への丸め」へ変更してから、再び「最も近い値へ丸め」へ戻さなくても計算を再開できます。

MXCSR レジスターへの書き込みにはペナルティーが生じるため、頻繁に変更するべきではありません。通常、CVTTPS2PI/CVTTSS2SI 命令を使用する場合、MXCSR レジスターの丸め制御モードは常に「近似値へ丸め」に設定します。

## 6.5.3 ゼロ・フラッシュ・モードと DAZ モード

ゼロ・フラッシュ (FTZ) モードとデノーマルをゼロとして扱う (DAZ) モードは、IEEE 規格 754 とは互換性がありません。これらのモードは、アンダーフローが頻発するアプリケーションや、非正規化された結果を生成する必要のないアプリケーションのパフォーマンスを改善するために用意されています。

関連情報: 3.8.3 節「浮動小数点モードと浮動小数点例外」を参照してください。

## 6.6 SIMD の最適化とマイクロアーキテクチャー

インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、およびインテル® Core™ Duo プロセッサーのマイクロアーキテクチャーは、Intel NetBurst® マイクロアーキテクチャーとは異なります。インテル® Core™ マイクロアーキテクチャーは、従来のマイクロアーキテクチャーよりも大幅に効率的な SIMD 浮動小数点機能を提供します。また、インテル® Core™ マイクロアーキテクチャーでは、インテル® SSE3 命令のレイテンシーとスループットが従来のマイクロアーキテクチャーよりもかなり向上しています。

### 6.6.1 インテル® SSE3 を使用した SIMD 浮動小数点プログラミング

インテル® SSE3 は、SIMD 浮動小数点プログラミング向けの 9 つの命令を追加しインテル® SSE とインテル® SSE2 を強化しています。インテル® SSE/インテル® SSE2 の多くの命令が並列データ要素の同種算術演算を提供し、垂直計算モデルを優先しているのに対し、インテル® SSE3 は非対称算術演算と水平データ要素の算術演算を行う命令を提供します。

ADDSUBPS と ADDSUBPD の 2 つの命令は、非対称算術処理機能を備えています (図 6-5 を参照)。HADDPS、HADDPD、HSUBPS、HSUBPD 命令は、水平算術処理機能を備えています (図 6-6 を参照)。また、MOVSLDUP、MOVSHDUP、MOVDDUP 命令は、メモリー (または XMM レジスター) からデータをロードして、一度に複数のデータ要素を複製できます。

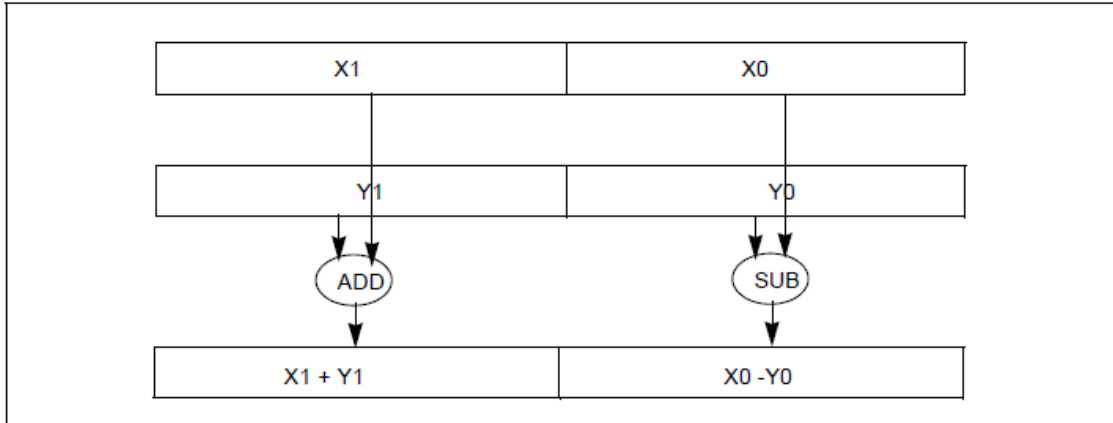


図 6-5 インテル® SSE3 命令の非対称算術演算

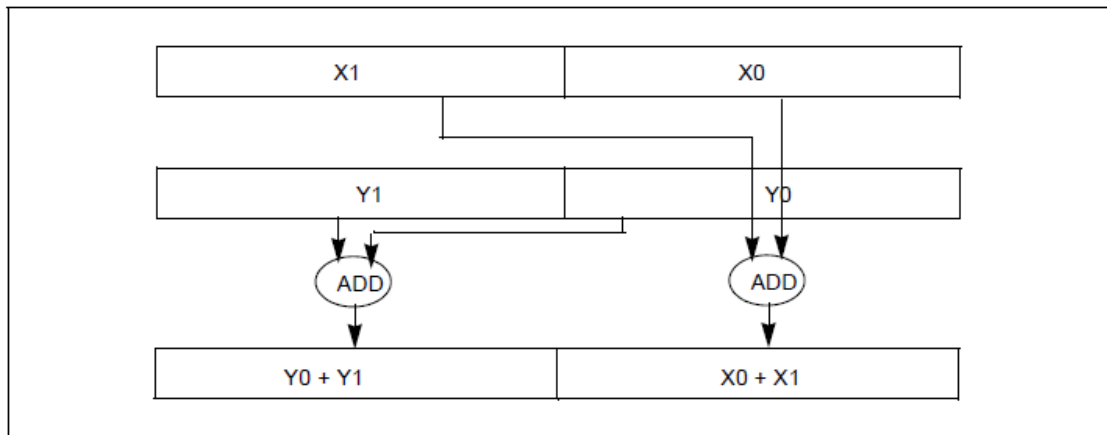


図 6-6 インテル® SSE3 命令 HADDPD の水平算術演算

### 6.6.1.1 インテル® SSE3 と複素数演算

AoS 形式のデータ構造体を処理する際のインテル® SSE3 の柔軟性は、複素数の乗算と除算の例を用いて示すことができます。例えば、複素数は、実数部分と虚数部分で構成された構造体に保存できます。これは自然なことです。構造体配列を利用することになります。例 6-9 では、インテル® SSE3 命令を使用して単精度複素数の乗算を行い、例 6-10 では、インテル® SSE3 命令を使用して複素数の除算を行っています。

#### 例 6-9 2 組の単精度複素数の乗算

```
// (ak + i bk ) * (ck + i dk ) の乗算
// a + i b はデータ構造体として格納
movsldup xmm0, Src1; 実数部分のロード a1, a1, a0, a0
movaps xmm1, src2; 2 番目のペアの複素数をロード i.e. d1, c1, d0, c0
mulps xmm0, xmm1; a1d1, a1c1, a0d0, a0c0 は一時的な結果
shufps xmm1, xmm1, b1; 実数部と虚数部の並び替え c1, d1, c0, d0
movshdup xmm2, Src1; 虚数部を b1, b1, b0, b0 にロード
mulps xmm2, xmm1; b1c1, b1d1, b0c0, b0d0 は一時的な結果
addsubps xmm0, xmm2; b1c1+a1d1, a1c1 -b1d1, b0c0+a0d0, a0c0-b0d0
```

例 6-10 2 組の単精度複素数の除算

```
// (ak + i bk ) * (ck + i dk ) の除算
movshdup xmm0, Src1; 虚数部を b1, b1, b0, b0 にロード
movaps xmm1, src2; 2 番目のペアの複素数をロード i.e. d1, c1, d0, c0
mulps xmm0, xmm1; b1d1, b1c1, b0d0, b0c0 は一時的な結果
shufps xmm1, xmm1, b1; 実数部と虚数部の並び替え c1, d1, c0, d0
movsldup xmm2, Src1; 1 実数部を a1, a1, a0, a0 に格納
mulps xmm2, xmm1; a1c1, a1d1, a0c0, a0d0 は一時的な結果
addsubps xmm0, xmm2; a1c1+b1d1, b1c1-a1d1, a0c0+b0d0, b0c0-a0d0
mulps xmm1, xmm1; c1c1, d1d1, c0c0, d0d0
movps xmm2, xmm1; c1c1, d1d1, c0c0, d0d0
shufps xmm2, xmm2, b1; d1d1, c1c1, d0d0, c0c0
addps xmm2, xmm1; c1c1+d1d1, c1c1+d1d1, c0c0+d0d0, c0c0+d0d0
divps xmm0, xmm2
shufps xmm0, xmm0, b1; (b1c1-a1d1)/(c1c1+d1d1), (a1c1+b1d1)/(c1c1+d1d1),
; (b0c0-a0d0)/(c0c0+d0d0), (a0c0+b0d0)/(c0c0+d0d0)
```

両者とも、複素数は構造体配列に保存されます。MOVSLDUP、MOVSHDUP、非対称 ADDSUBPS 命令を利用すると、データ要素間でスイッチングを行うことなく、同時に 2 組の単精度複素数に対して複素数演算を実行できます。

マイクロアーキテクチャーの違いにより、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサでは、インテル® SSE3 命令を使用して倍精度複素数の乗算を実装すべきです。また、インテル® Core™ Duo プロセッサとインテル® Core™ Solo プロセッサでは、インテル® SSE2 のスカラー命令を使用して倍精度複素数の乗算を実装すべきです。これは、SIMD 実行ユニット間のデータパスはインテル® Core™ マイクロアーキテクチャーでは 128 ビットであるのに対し、従来のマイクロアーキテクチャーでは 64 ビットしかないため 128 ビットの演算ではスループットが低下することが理由です。拡張版インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサでは、通常、従来のマイクロアーキテクチャーよりも効率良くインテル® SSE3 命令が実行されます。また、128 ビット・シャッフル・ユニットも搭載しており、インテル® Core™ マイクロアーキテクチャーよりもさらに大きなメリットを複素数演算にもたらしめます。

例 6-11 に示す 2 つのコードでは、インテル® SSE2 のベクトル命令とインテル® SSE3 のベクトル命令を使用して、2 組の複素数に対し倍精度複素数乗算を行っています。例 6-12 に、インテル® SSE2 のスカラー命令を使用した同等のコードを示します。

例 6-11 2 組の複素数に対する倍精度複素数乗算

インテル® SSE2 ベクトル実装	インテル® SSE3 ベクトル実装
movapd xmm0, [eax] ; y x	movapd xmm0, [eax] ; y x
movapd xmm1, [eax+16] ; w z	movapd xmm1, [eax+16] ; z z
unpcklpd xmm1, xmm1 ; z z	movapd xmm2, xmm1
movapd xmm2, [eax+16] ; w z	unpcklpd xmm1, xmm1
unpckhpd xmm2, xmm2 ; w w	unpckhpd xmm2, xmm2
mulpd xmm1, xmm0 ; z*y z*x	mulpd xmm1, xmm0 ; z*y z*x
mulpd xmm2, xmm0 ; w*y w*x	mulpd xmm2, xmm0 ; w*y w*x
xorpd xmm2, xmm7 ; -w*y +w*x	shufpd xmm2, xmm2, 1 ; w*x w*y
shufpd xmm2, xmm2, 1 ; w*x -w*y	addsubpd xmm1, xmm2 ; w*x+z*y z*x-w*y
addpd xmm2, xmm1 ; z*y+w*x z*x-w*y	movapd [ecx], xmm1
movapd [ecx], xmm2	

例 6-12 インテル® SSE2 のスカラー命令を使用した倍精度複素数乗算

```

movsd xmm0, [eax] ;x
movsd xmm5, [eax+8] ;y
movsd xmm1, [eax+16] ;z
movsd xmm2, [eax+24] ;w
movsd xmm3, xmm1 ;z
movsd xmm4, xmm2 ;w
mulsd xmm1, xmm0 ;z*x
mulsd xmm2, xmm0 ;w*x
mulsd xmm3, xmm5 ;z*y
mulsd xmm4, xmm5 ;w*y
subsd xmm1, xmm4 ;z*x - w*y
addsd xmm3, xmm2 ;z*y + w*x
movsd [ecx], xmm1
movsd [ecx+8], xmm3
    
```

### 6.6.1.2 インテル® Core™ Duo プロセッサにおけるパックド浮動小数点のパフォーマンス

インテル® Core™ Solo プロセッサでは、ほとんどのパックド SIMD 浮動小数点コードがインテル® Pentium® M プロセッサと比べて高速化されています。これは、パックド SIMD 命令のデコード性能の向上が要因です。

インテル® Core™ Solo プロセッサでパックド浮動小数点のパフォーマンスがインテル® Pentium® M プロセッサよりも向上するかどうかは、複数の要因に依存しています。一般に、デコーダー依存のコードや、整数命令とパックド浮動小数点命令が混在したコードでは、大幅な向上を期待できません。実行レイテンシーによって制限され、「1 命令あたりのサイクル数」の比率が 1 よりも大きいコードでは、デコーダーの向上によるメリットは得られません。

インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサで複素数演算を行う際に単精度のインテル® SSE3 命令を使用すると、その他の方法よりもパフォーマンスが向上することが期待できます。一方、インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサで倍精度複素数演算が必要なタスクでは、インテル® SSE2 のスカラー命令を使用した方がパフォーマンスは向上します。これは、インテル® SSE2 のスカラー命令では、2 つのポートを介してディスパッチし、2 つの独立した浮動小数点ユニットによって実行されるためです。

一部のタスクでは、インテル® SSE3 のパックド水平命令 (HADDPS と HSUBPS) を使用すると、コードシーケンスを簡素化できます。ただし、インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサのでは、これらの命令は、6 以上のマイクロオペレーション (μop) で構成されます。そのため、水平命令でのレイテンシーとデコードのペナルティによってアルゴリズム上のメリットが相殺されないように注意する必要があります。

### 6.6.2 ドット積と水平 SIMD 命令

多くの代数式は、AoS 形式のデータ編成の方が適していることが多く、その典型的な例がドット積演算です。ドット積演算は、インテル® SSE/インテル® SSE2 命令セットを使用して実装できます。インテル® SSE3 では、水平計算モデルに依存したアプリケーション向けに水平加算/減算命令がいくつか追加されています。インテル® SSE4.1 では、2、3、または 4 つの成分からなるベクトルのドット積演算を直接評価できるように命令が拡張されています。

例 6-13 インテル® SSE/インテル® SSE2 を使用したベクトル長 4 のドット積演算

インテル® SSE/インテル® SSE2 を使用して 1 つのドット積を計算
<pre>movaps xmm0, [eax] // a4, a3, a2, a1 mulps xmm0, [eax+16] // a4*b4, a3*b3, a2*b2, a1*b1 movhlps xmm1, xmm0 // X, X, a4*b4, a3*b3, 上位半分は不要 addps xmm0, xmm1 // X, X, a2*b2+a4*b4, a1*b1+a3*b3, pshufd xmm1, xmm0, 1 // X, X, X, a2*b2+a4*b4 addss xmm0, xmm1 // a1*b1+a3*b3+a2*b2+a4*b4 movss [ecx], xmm0</pre>

例 6-14 インテル® SSE3 を使用したベクトル長 4 のドット積演算

インテル® SSE3 を使用して 1 つのドット積を計算
<pre>movaps xmm0, [eax] mulps xmm0, [eax+16] // a4*b4, a3*b3, a2*b2, a1*b1 haddps xmm0, xmm0 // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1 movaps xmm1, xmm0 // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1 psrlq xmm0, 32 // 0, a4*b4+a3*b3, 0, a4*b4+a3*b3 addss xmm0, xmm1 // -, -, -, a1*b1+a3*b3+a2*b2+a4*b4 movss [eax], xmm0</pre>

例 6-15 インテル® SSE4.1 を使用したベクトル長 4 のドット積演算

インテル® SSE4.1 を使用して 1 つのドット積を計算
<pre>movaps xmm0, [eax] dpps xmm0, [eax+16], 0xf1 // 0, 0, 0, a1*b1+a3*b3+a2*b2+a4*b4 movss [eax], xmm0</pre>

例 6-13、例 6-14、例 6-15 では、1 組のベクトルに対して 1 つのドット積を計算する基本的なコードシーケンスを比較しています。

アプリケーションのメモリー・アクセス・パターンに合わせて最適なシーケンスを選択することは、各種の手法にとってメリットがあります。例えば、各ドット積が後続の計算シーケンスですぐに使用されるのであれば、各種手法の相対速度を比較するのに適しています。ベクトル配列のドット積を計算し、その結果を後続の計算のためキャッシュ内に保持できる場合、適切な選択は命令シーケンスの相対スループットに依存します。

インテル® Core™ マイクロアーキテクチャーでは、例 6-14 の方が例 6-13 よりもスループットは高くなります。ただし、HADDPS 命令のレイテンシーが比較的長いため、例 6-14 の速度は例 6-13 よりもわずかに遅くなります。

拡張版インテル® Core™ マイクロアーキテクチャーでは、例 6-15 の方が速度とスループットの両面において例 6-13 と例 6-14 よりも優れています。DPPS 命令のレイテンシーも比較的長めですが、例 6-15 では、同じ処理量に対する命令数の減少によって補間されています。

アンロールを適用すると、3 つのドット積演算コードのそれぞれでスループットをさらに向上させることができます。例 6-16 に、インテル® SSE2 とインテル® SSE3 による基本シーケンスを 2 回アンロールした例を示します。インテル® SSE4.1 でもアンロール可能であり、INSERTPS 命令を使用して 4 つのドット積をパックできます。

例 6-16 4 つのドット積に対するアンロールの実装

インテル® SSE2 実装	インテル® SSE3 実装
<pre> movaps xmm0, [eax] mulps xmm0, [eax+16] ;w0*w1 z0*z1 y0*y1 x0*x1 movaps xmm2, [eax+32] mulps xmm2, [eax+16+32] ;w2*w3 z2*z3 y2*y3 x2*x3 movaps xmm3, [eax+64] mulps xmm3, [eax+16+64] ;w4*w5 z4*z5 y4*y5 x4*x5 movaps xmm4, [eax+96] mulps xmm4, [eax+16+96] ;w6*w7 z6*z7 y6*y7 x6*x7 movaps xmm1, xmm0 unpcklps xmm0, xmm2 ; y2*y3 y0*y1 x2*x3 x0*x1 unpckhps xmm1, xmm2 ; w2*w3 w0*w1 z2*z3 z0*z1 movaps xmm5, xmm3 unpcklps xmm3, xmm4 ; y6*y7 y4*y5 x6*x7 x4*x5 unpckhps xmm5, xmm4 ; w6*w7 w4*w5 z6*z7 z4*z5 addps xmm0, xmm1 addps xmm5, xmm3 movaps xmm1, xmm5 movhlps xmm1, xmm0 movlhps xmm0, xmm5 addps xmm0, xmm1 movaps [ecx], xmm0                     </pre>	<pre> movaps xmm0, [eax] mulps xmm0, [eax+16] movaps xmm1, [eax+32] mulps xmm1, [eax+16+32] movaps xmm2, [eax+64] mulps xmm2, [eax+16+64] movaps xmm3, [eax+96] mulps xmm3, [eax+16+96] haddps xmm0, xmm1 haddps xmm2, xmm3 haddps xmm0, xmm2 movaps [ecx], xmm0                     </pre>

### 6.6.3 ベクトルの正規化

ベクトルの正規化は、多くの浮動小数点アプリケーションにおいて一般的な操作です。例 6-17 に、(x, y, z) ベクトルの配列を正規化する C コードの例を示します。

例 6-17 ベクトルの配列の正規化

```

for (i=0;i<CNT;i++)
{ float size = nodes[i].vec.dot();
  if (size != 0.0)
    { size = 1.0f/sqrtf(size); }
  else
    { size = 0.0; }
  nodes[i].vec.x *= size;
  nodes[i].vec.y *= size;
  nodes[i].vec.z *= size;
}
                    
```

例 6-18 に、ベクトルの (x, y, z) 成分を正規化するアセンブリ・シーケンスを示します。

例 6-18 インテル® SSE2 を使用した、ベクトル配列 (x, y, z) の成分に対する正規化

```
Vec3 *p = &nodes[i].vec;
__asm
{
    mov eax, p
    xorps xmm2, xmm2
    movups xmm1, [eax] // 入力ベクトルと次のベクトルの x を加えた (x,y,z) をロード
    movaps xmm7, xmm1 // (正規化されていないデータを戻すため) メモリーのデータのコピーを格納
    movaps xmm5, _mask // 正規化するために xmm レジスターから (x,y,z) 値を選択するためにマスクする
    andps xmm1, xmm5 // 1 番目の 3 要素をマスクする
    movaps xmm6, xmm1 // 後で正規化されたベクトルを計算するために (x,y,z) のコピーを格納
    mulps xmm1, xmm1 // 0, z*z, y*y, x*x
    pshufd xmm3, xmm1, 0x1b // x*x, y*y, z*z, 0
    addps xmm1, xmm3 // x*x, z*z+y*y, z*z+y*y, x*x
    pshufd xmm3, xmm1, 0x41 // z*z+y*y, x*x, x*x, z*z+y*y
    addps xmm1, xmm3 // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd xmm1, xmm2 // サイズを 0 と比較
    jz zero
    movaps xmm3, xmm4 // 単位ベクトル (1.0, 1.0, 1.0, 1.0) のプリロード
    sqrtps xmm1, xmm1
    divps xmm3, xmm1
    jmp store
zero:
    movaps xmm3, xmm2
store:
    mulps xmm3, xmm6 // 下位 3 要素のベクトルを正規化
    andnps xmm5, xmm7 // 正規化されない値を保持するために下位 3 要素をマスクオフする
    orps xmm3, xmm5 // 正規化されたベクトルの後に正規化されない要素を並べる
    movaps [eax], xmm3 // 正規化された x, y, z を代入、変更されない値が続く
}
```

例 6-19 に、インテル® SSE4.1 を使用したベクトル (x, y, z) の成分を正規化するアセンブリー・シーケンスを示します。

例 6-19 インテル® SSE4.1 を使用した、ベクトル配列 (x, y, z) の成分に対する正規化

```
Vec3 *p = &nodes[i].vec;
__asm
{
    mov eax, p
    xorps xmm2, xmm2
    movups xmm1, [eax] // 入力ベクトルと次のベクトルの x を加えた (x,y,z) をロード
    movaps xmm7, xmm1 // メモリーからデータのコピーを格納
    dpps xmm1, xmm1, 0x7f // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd xmm1, xmm2 // サイズを 0 と比較
    jz zero
    movaps xmm3, xmm4 // 単位ベクトル (1.0, 1.0, 1.0, 1.0) のプリロード
    sqrtps xmm1, xmm1
    divps xmm3, xmm1
    jmp store
zero:
    movaps xmm3, xmm2
store:
    mulps xmm3, xmm6 // 下位 3 要素のベクトルを正規化
    blendps xmm3, xmm7, 0x8 // 正規化されてない要素を正規ベクトルの次にコピー
    movaps [eax], xmm3
}
```

例 6-18 と例 6-19 では、レイテンシーの大きな DIVPS 命令と SQRTPS 命令によって根本的に命令シーケンスのスループットが制限されます。例 6-19 では、8 つのインテル® SSE2 命令に代わって DPPS 命令を使用して、XMM レジスターの 4 つの要素に対するドット積の評価やブロードキャストを行っています。これにより、例 6-19 の相対速度は例 6-18 よりも向上します。

## 6.6.4 水平 SIMD 命令セットの使用とデータレイアウト

インテル® SSE とインテル® SSE2 にはパックド加算/減算、乗算/除算命令が用意されており、SoA データレイアウトなどの垂直計算モデルを活用するのが理想的です。インテル® SSE3 とインテル® SSE4.1 では、水平加算/減算、ドット積演算などの水平 SIMD 命令が追加しました。これらの新しい SIMD 拡張命令では、垂直 SIMD 計算モデルに対応しないデータレイアウトや演算の問題を解決するツールも提供されています。

ここでは、ベクトル-行列乗算の問題について検討し、各種の水平 SIMD 命令を選択する際に関連する要素について説明します。

例 6-20 に、入出力ベクトルが構造体の配列としてストアされる AoS を使用する、ベクトル-行列データレイアウトを示します。

例 6-20 AoS ベクトル-行列乗算のためのメモリーデータ編成

```
Matrix M4x4 (pMat): M00 M01 M02 M03
                    M10 M11 M12 M13
                    M20 M21 M22 M23
                    M30 M31 M32 M33
4 input vertices V4x1 (pVert): V0x V0y V0z V0w
                               V1x V1y V1z V1w
                               V2x V2y V2z V2w
                               V3x V3y V3z V3w
Output vertices O4x1 (pOutVert): O0x O0y O0z O0w
                                O1x O1y O1z O1w
                                O2x O2y O2z O2w
                                O3x O3y O3z O3w
```

例 6-21 に示す例では、HADDPS と MULPS 命令を使用して、AoS のデータレイアウトのベクトル-行列乗算を行っています。3 つの HADDPS 命令が各出力ベクトル成分の合計を完了した後、出力成分が AoS に配置されます。

例 6-21 HADDPS 命令を使用した AoS ベクトル-行列乗算

```
mov eax, pMat
mov ebx, pVert
mov ecx, pOutVert
xor edx, edx
movaps xmm5,[eax+16] // 列 M1 のロード
movaps xmm6,[eax+2*16] // 列 M2 のロード
movaps xmm7,[eax+3*16] // 列 M3 のロード
lloop:
movaps xmm4, [ebx + edx] // 入力ベクトルのロード
movaps xmm0, xmm4
mulps xmm0, [eax] // m03*vw, m02*vz, m01*vy, m00*vx,
movaps xmm1, xmm4
mulps xmm1, xmm5 // m13*vw, m12*vz, m11*vy, m10*vx,
movaps xmm2, xmm4
mulps xmm2, xmm6 // m23*vw, m22*vz, m21*vy, m20*vx
movaps xmm3, xmm4
mulps xmm3, xmm7 // m33*vw, m32*vz, m31*vy, m30*vx,
haddps xmm0, xmm1
haddps xmm2, xmm3
haddps xmm0, xmm2
movaps [ecx + edx], xmm0 // 長さ 4 のベクトルを格納
add edx, 16
cmp edx, top
jb lloop
```



例 6-22 に示す例では、DPPS 命令を使用して、AoS のベクトル-行列乗算を行っています。

例 6-22 DPPS 命令を使用した AoS ベクトル-行列乗算

```

mov eax, pMat
mov ebx, pVert
mov ecx, pOutVert
xor edx, edx
movaps xmm5, [eax+16] // 列 M1 のロード
movaps xmm6, [eax+2*16] // 列 M2 のロード
movaps xmm7, [eax+3*16] // 列 M3 のロード
lloop:
movaps xmm4, [ebx + edx] // 入力ベクトルのロード
movaps xmm0, xmm4
dpps xmm0, [eax], 0xf1 // 長さ 4 の内積を計算し、最下位の DWORD に格納
movaps xmm1, xmm4
dpps xmm1, xmm5, 0xf1
movaps xmm2, xmm4
dpps xmm2, xmm6, 0xf1
movaps xmm3, xmm4
dpps xmm3, xmm7, 0xf1
movss [ecx + edx + 0*4], xmm0 // ベクトル長が 4 の 1 つ要素を格納
movss [ecx + edx + 1*4], xmm1
movss [ecx + edx + 2*4], xmm2
movss [ecx + edx + 3*4], xmm3
add edx, 16
cmp edx, top
jb lloop
    
```

例 6-21 と例 6-22 では、インテル® SSE3 とインテル® SSE4.1 で提供されている異なる水平処理手法を使用して、AoS データレイアウトを操作しています。それぞれの手法の効果は、内部ループでのレイテンシーが長い命令の頻度、データ移動のオーバーヘッド/効率、HADDPS と DPPS 命令のレイテンシーによって異なります。

HADDPS と DPPS の両方をサポートするプロセッサでは、いずれの手法を選択するかは、アプリケーション固有の考慮事項に依存します。バッチ状況で出力ベクトルがメモリーに直接ライトバックされる場合、例 6-21 の方が例 6-22 よりも推奨されます。それは、DPPS 命令はレイテンシーが長く、各出力ベクトル成分を個別にストアするのは、ベクトル配列のストアにあまり適していないためです。

出力ベクトル成分がほかのベクトル化できない計算によってすぐに使用される、部分的にベクトル化可能な状況では、例 6-21 のように 3 つの HADDPS 命令によって生成されたパックド出力ベクトルを分散させるよりも、個別の成分を生成する DPPS 命令を使用する方が適しています。

### 6.6.4.1 SoA とベクトル-行列乗算

対象となるネイティブデータレイアウトが SoA に対応している場合、MULPS 命令、レイテンシーの長い水平計算命令を使用しない ADDPS 命令、またはパックド形式へのスカラー成分のパック (例 6-22) を利用してベクトル-行列乗算を表現できます。SoA データレイアウトで高いスループットを達成するには、あらかじめ必要なデータを準備するか、または実行時のスイッチング/デスイッチングを理解する必要があります。例えば、ベクトル-行列乗算の SoA データレイアウトを例 6-23 に示します。

パックされた結果を生成するため、各行列要素を 4 回複製することによってデータ移動のオーバーヘッドを最小限に抑えています。

例 6-23 SoA ベクトル-行列乗算のメモリーデータ編成

```

Matrix M16x4 (pMat):
  M00 M00 M00 M00 M01 M01 M01 M01 M02 M02 M02 M02 M03 M03 M03 M03
  M10 M10 M10 M10 M11 M11 M11 M11 M12 M12 M12 M12 M13 M13 M13 M13
  M20 M20 M20 M20 M21 M21 M21 M21 M22 M22 M22 M22 M23 M23 M23 M23
  M30 M30 M30 M30 M31 M31 M31 M31 M32 M32 M32 M32 M33 M33 M33 M33
4 input vertices V4x1 (pVert): V0x V1x V2x V3x
                                V0y V1y V2y V3y
                                V0z V1z V2z V3z
                                V0w V1w V2w V3w
Output vertices O4x1 (pOutVert): O0x O1x O2x O3x
                                O0y O1y O2y O3y
                                O0z O1z O2z O3z
                                O0w O1w O2w O3w
    
```

これに対応する SoA のベクトル-行列乗算の例 (ベクトル反復 4 回をアンロール) を例 6-24 に示します。

例 6-24 ネイティブ SoA データレイアウトを使用したベクトル-行列乗算

```

mov ebx, pVert
mov ecx, pOutVert
xor edx, edx
movaps xmm5,[eax+16] // 列 M1 のロード
movaps xmm6,[eax+2*16] // 列 M2 のロード
movaps xmm7,[eax+3*16] // 列 M3 のロード
lloop_vert:
  mov eax, pMat
  xor edi, edi
  movaps xmm0, [ebx ] // V3x, V2x, V1x, V0x のロード
  movaps xmm1, [ebx ] // V3y, V2y, V1y, V0y のロード
  movaps xmm2, [ebx ] // V3z, V2z, V1z, V0z のロード
  movaps xmm3, [ebx ] // V3w, V2w, V1w, V0w のロード
loop_mat:
  movaps xmm4, [eax] // m00, m00, m00, m00,
  mulps xmm4, xmm0 // m00*V3x, m00*V2x, m00*V1x, m00*V0x,
  movaps xmm4, [eax + 16] // m01, m01, m01, m01,
  mulps xmm5, xmm1 // m01*V3y, m01*V2y, m01*V1y, m01*V0y,
  addps xmm4, xmm5
  movaps xmm5, [eax + 32] // m02, m02, m02, m02,
  mulps xmm5, xmm2 // m02*V3z, m02*V2z, m02*V1z, m02*V0z,
  addps xmm4, xmm5
  movaps xmm5, [eax + 48] // m03, m03, m03, m03,
  mulps xmm5, xmm3 // m03*V3w, m03*V2w, m03*V1w, m03*V0w,
  addps xmm4, xmm5
  movaps [ecx + edx], xmm4
  add eax, 64
  add edx, 16
  add edi, 1
  cmp edi, 4
  jb lloop_mat
  add ebx, 64
  cmp edx, top
  jb lloop_vert
    
```

## 第 7 章 キャッシュ利用の最適化

この 10 年以上でプロセッサの速度は向上しています。それに比べると、メモリーのアクセス速度は遅いペースで向上しています。これにより生じた差を埋めるため、次のいずれかの方法でアプリケーションをチューニングすることが重要になりました: (a) 主要なデータアクセスをプロセッサのキャッシュで行う。 (b) メモリー・レイテンシーを効果的にマスクして、ピーク時のメモリー帯域幅をできるだけ多く利用する。

ハードウェア・プリフェッチ機構は、後者を容易にするマイクロアーキテクチャーの拡張機能であり、ソフトウェア・チューニングと組み合わせるとさらに効果的です。必要なデータをプロセッサのキャッシュからフェッチできるか、メモリー・トラフィックがハードウェア・プリフェッチを効果的に活用できれば、ほとんどのアプリケーションのパフォーマンスが大幅に向上します。

必要なデータを事前にプロセッサに取り込むには、別にプログラムコードを追加するのが通常の方法ですが、そうしたコードは実装が難しく、またパフォーマンス低下を避けるために特殊な手順を考慮しなければならないこともあります。

インテル® ストリーミング SIMD 拡張命令では、各種プリフェッチ命令によってそのような問題点を解決します。

インテル® ストリーミング SIMD 拡張命令では、何種類かの非テンポラルなストア命令がサポートされています。インテル® ストリーミング SIMD 拡張命令 2 では、そのサポート範囲が新たなデータ型にまで広がったほか、32 ビット整数レジスター用の非テンポラルなストア機能も導入されました。

本章では、主に次のテーマに焦点をあてています。

- Y ハードウェア・プリフェッチ機構、ソフトウェア・プリフェッチ命令、キャッシュ制御命令: アプリケーションにおけるデータキャッシュの動作を制御可能なマイクロアーキテクチャーの機能と命令について説明します。
- Y ハードウェア・プリフェッチ機構、ソフトウェア・プリフェッチ命令、およびキャッシュ制御命令:
  - これらの命令を使用してメモリーを最適化する手法について説明します。

### 注意

ここで説明しているプリフェッチおよびキャッシュの利用方法のいくつかは、Intel NetBurst® マイクロアーキテクチャー専用のもが含まれますが、大部分は、将来発表されるプロセッサにも適用できます。

- Y キャッシュ・パラメーターを使用してキャッシュ階層を管理します。

## 7.1 プリフェッチのコーディングに関する一般的なガイドライン

次のガイドラインに従うことで、メモリーシステムにおいて大量のデータ移動が発生するときでもメモリー・トラフィックを軽減し、より効率良くピーク時のメモリーシステムの帯域幅が利用できます。

- Y 前方もしくは後方への連続したパターンでアクセスするデータのプリフェッチは、ハードウェア・プリフェッチ機能によりさらにパフォーマンスが向上します。
- Y ハードウェア・プリフェッチのトリガーとなる間隔の半分未満のアクセスストライドを持つパターンでアクセスされるデータを、ハードウェア・プリフェッチャーの機能を利用してプリフェッチします。
- Y 以下はコンパイラーによる最適化を容易にします。
  - グローバル変数とグローバルポインターの使用を最小限に抑えます。
  - 複雑な制御フローを最小限にします。
  - const 修飾子を使用し、register 修飾子を避けます。

- データ型の選択には注意し (下記参照)、タイプキャストは避けます。
- Y ストリップマイニングなど、キャッシュ・ブロッキング手法を導入します。
  - 1 次元配列に対してはストリップマイニングなどのキャッシュ・ブロッキング手法を用い、2 次元配列に対してはループ・ブロッキング手法を用いて、キャッシュのヒット率を上げます。
  - データ・アクセス・パターンに十分な規則性があり、データアクセスの別の順序付け (タイル化など) によって空間的な局所性の改善が見込める場合は、ハードウェア・プリフェッチ機構の使用を検討します。それ以外の場合は、PREFETCHNTA 命令を使用します。
- Y シングルパス実行とマルチパス実行のバランスをとります。
  - 「シングルパス実行」とは、「非階層化実行」とも呼ばれ、計算パイプライン全体でデータ要素を 1 つ通過させるものです。
  - 「マルチパス実行」とは、「階層化実行」とも呼ばれ、複数のデータ要素から成る 1 つのデータ群を対象にして、パイプラインのステージを 1 段実行してからそのデータ群を次のステージに渡すものです。
  - アルゴリズムがシングルパスのときは PREFETCHNTA 命令を使用し、アルゴリズムがマルチパスのときは PREFETCHT0 命令を使用します。
- Y メモリーバンクの競合問題を解決します。配列グループ化手法を用いて連続して使用するデータをまとめてグループ化するか、4KB のメモリーページに収まるようにデータを割り当て、できる限りメモリーバンクの競合を避けます。
- Y キャッシュ管理の問題を解決します。プロセッサのキャッシュに保持されるテンポラルなデータの汚染をできる限り抑えるため、ストリーミング・ストア命令を使用します。
- Y ソフトウェア・プリフェッチのスケジューリング間隔を最適化します。
  - 中間の計算処理がメモリーアクセス時間をオーバーラップするように、プリフェッチのスケジューリング間隔を十分に広げます。
  - プリフェッチされたデータがデータキャッシュからのデータに置き換わらないように、プリフェッチのスケジューリング間隔を十分に狭めます。
- Y ソフトウェア・プリフェッチを連結します。プリフェッチをいくつかうまく並べて、不要なプリフェッチ命令が内部ループの末尾で実行されないようにし、なおかつ次の外部ループの内側でその内部ループにおける最初の数回の反復がプリフェッチされるようにします。
- Y ソフトウェア・プリフェッチの回数を最小限にします。プリフェッチ命令は、バスサイクル、マシンサイクル、リソースといった観点から見た場合、必ずしも完全に自由に使用できるわけではありません。プリフェッチを多用すると、アプリケーションのパフォーマンスに悪影響を与えることがあります。
- Y ソフトウェア・プリフェッチ命令の間に演算命令をいくつか挿入します。最高のパフォーマンスを得るには、命令シーケンスの中で複数のプリフェッチ命令を (1 カ所に固めて並べるのではなく)、各プリフェッチ命令は間隔を空けて配置し、その間に演算命令を挿入する必要があります。

## 7.2 プリフェッチとキャッシュ制御命令

PREFETCH 命令は、プログラマーまたはコンパイラーによって挿入されます。目的のデータが実際に必要になる前に、PREFETCH 命令でインテル® Pentium® 4 プロセッサのデータのキャッシュラインの少なくとも 2 つにアクセスします (インテル® Pentium® M プロセッサの場合はデータのキャッシュラインの少なくとも 1 つ)。これにより、すでにキャッシュに格納されているデータを処理することで、データアクセスに要するレイテンシーが隠蔽されます。

多くのアルゴリズムでは、必要になるデータに関する情報は事前に判明しています。長いデータパターンでメモリーアクセスが行われるような場合は、ソフトウェア・プリフェッチよりも自動ハードウェア・プリフェッチを優先的に使用する必要があります。

キャッシュ制御命令を使用してデータのキャッシュ方式を制御すると、キャッシュ効率が上がり、かつキャッシュ汚染が最小限に抑えられます。

データ参照パターンは、以下のように分類できます。

- Y テンポラル: 時間的にそれほど間を置かずにくぐまたデータが使用されます。
- Y 空間的: 隣接した場所 (同じキャッシュラインなど) のデータが使用されます。
- Y 非テンポラル: 一度参照された後しばらくデータは再利用されません (例えば 3D グラフィックス・アプリケーションのバーテックス・バッファーのような、一部のマルチメディア・データ・タイプがこれに相当します)。

上記のデータ特性を用いて、以降の説明を進めます。

## 7.3 プリフェッチ

この節では、ソフトウェアによる PREFETCH 命令の仕組みについて説明します。一般に、ソフトウェア・プリフェッチ命令は、自動ハードウェア・プリフェッチ機構と合わせてアクセスパターンをチューニングする際の補助として使用すべきです。

### 7.3.1 ソフトウェアによるデータ・プリフェッチ

PREFETCH 命令を使用して、実際にデータが必要になる前にフェッチしておけば、アプリケーション・コードの中でも特に高いパフォーマンスの要求される部分でデータアクセスのレイテンシーが隠蔽されます。PREFETCH 命令を使用しても、ユーザーから見たプログラムの機能は変わりませんが、プログラム・パフォーマンスに影響することがあります。

PREFETCH 命令はハードウェアにヒントを与えるにすぎず、それによって例外やフォルトが発生することはまずありません。PREFETCH 命令を実行すると、非テンポラルなデータまたはテンポラルなデータが、指定するキャッシュレベルにロードされます。データ・アクセス・タイプとキャッシュレベルをヒントとして指定します。実装方式に依存しますが、プリフェッチ命令を実行すると、(指定されたアドレスバイトも含めて) アライメントされた 32 バイト以上がフェッチされ、指定されたキャッシュレベルにデータが移動します。

PREFETCH 命令の使用方法はアーキテクチャーの実装方式によって異なります。パフォーマンスを最大にするため、それぞれの実装方式に合わせてアプリケーションをチューニングします。

#### 注意

PREFETCH 命令は、データがキャッシュに収まらない場合にのみ使用することを推奨します。ソフトウェア・プリフェッチは、管理されているまたはアプリケーション内で所有されるメモリアドレスに対してのみ行われるべきです。プリフェッチするアドレスが物理ページに割り当てられていない場合、非決定論的なパフォーマンス上のペナルティーを被ります。例えば、プリフェッチするアドレスとして NULL ポインター (0L) を指定すると、長いレイテンシーの原因となります。

PREFETCH 命令はハードウェアにヒントを与え、特殊ないくつかの場合を除いて、同命令を実行しても例外やフォルトは発生しません (7.3.3 節「PREFETCH 命令とロード命令」を参照)。ただし、PREFETCH 命令を多用するとメモリー帯域幅が浪費され、その結果、リソース上の制約を受けてパフォーマンスが低下する可能性があります。

それでも PREFETCH 命令を使用すると、キャッシュ汚染が防止され、キャッシュもメモリーも利用効率が上がるため、メモリー・トランザクションのオーバーヘッドを軽減できます。この動作は、メモリーバスに代表される重要なシステムリソースを複数のアプリケーションで共有するようになるとき、特に重要となります。7.6.2.1 節「ビデオ・エンコーダー」に記載した例を参照してください。

PREFETCH の主な目的は、メモリー・レイテンシーを他の処理の背後に隠蔽することでアプリケーションのパフォーマンスを改善することにあります。アプリケーションがデータにアクセスするときどのような方式が採られるかあらかじめ予測できる場合 (例えば、ストライド幅がすでに決まっている配列を使用する場合など)、PREFETCH 命令を使用してパフォーマンスを改善できる可能性が高まります。

PREFETCH 命令は次の場所で使用します。

- Y 予測可能なメモリー・アクセス・パターン
- Y 時間を多く消費する最も内側のループ
- Y データが利用できないと実行パイプラインがストールするような場所

### 7.3.2 プリフェッチ命令

インテル® ストリーミング SIMD 拡張命令には 4 種類の PREFETCH 命令 (非テンポラルなものが 1 つに、テンポラルなものが 3 つ) があります。それぞれテンポラルな演算と非テンポラルな演算に対応しています。

さらに、PREFETCHW 命令はプロセッサのより近くにデータをフェッチするヒントを提供し、予測された書き込みによりキャッシュにコピーされたデータを無効化します。

#### 注意

PREFETCH 命令を実行しても、その時点で、この命令によって指定されているキャッシュレベルよりもプロセッサに近いキャッシュレベルにデータがすでに存在している場合、データは移動しません。

プリフェッチ命令におけるヒントの実装は、マイクロアーキテクチャーごとに異なります。以下に概要をまとめます。

- Y PREFETCHNTA: キャッシュの汚染を最小限に抑え、プロセッサに近い非テンポラルキャッシュにデータをフェッチします。
  - インテル® Pentium® III プロセッサ: 1 次キャッシュ
  - Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサ: 2 次キャッシュ
  - インテル® Core™ Duo プロセッサ、インテル® Core™2 プロセッサ、Intel Atom® プロセッサ: 1 次キャッシュではなく 2 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Core™ プロセッサ: 1 次キャッシュではなく 2 次キャッシュ。高速交換では 3 次キャッシュにフェッチできます。
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサ: 高速交換で 3 次キャッシュにフェッチする必要があります。
- Y PREFETCHT0: すべてのキャッシュレベルにデータをフェッチ。
  - インテル® Pentium® III プロセッサ: 1 次キャッシュではなく 2 次キャッシュ
  - Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサ: 2 次キャッシュ
  - インテル® Core™ Duo プロセッサ、インテル® Core™2 プロセッサ、Intel Atom® プロセッサ: 1 次キャッシュと 2 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Core™ プロセッサ: 1 次キャッシュ、2 次キャッシュ、および 3 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサ: 1 次キャッシュ、2 次キャッシュ、および 3 次キャッシュ
- Y PREFETCHT1: 2 次キャッシュと 3 次キャッシュにデータをフェッチ。
  - インテル® Pentium® III プロセッサ: 2 次キャッシュ
  - Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサ: 2 次キャッシュ
  - インテル® Core™ Duo プロセッサ、インテル® Core™2 プロセッサ、Intel Atom® プロセッサ: 2 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Core™ プロセッサ: 2 次キャッシュと 3 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサ: 2 次キャッシュと 3 次キャッシュ

- Y PREFETCHT2: PREFETCHT1 と同じ。
  - インテル® Pentium® III プロセッサ: 2 次キャッシュ
  - Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサ: 2 次キャッシュ
  - インテル® Core™ Duo プロセッサ、インテル® Core™2 プロセッサ、Intel Atom® プロセッサ: 2 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Core™ プロセッサ: 2 次キャッシュと 3 次キャッシュ
  - Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサ: 2 次キャッシュと 3 次キャッシュ
- Y PREFETCHW: 書き込みを予測してデータをキャッシュにフェッチ、キャッシュされたコピーを無効化。
  - Silvermont<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー・ベースの Intel Atom® プロセッサ: 1 次キャッシュと 2 次キャッシュ
  - Broadwell<sup>†</sup> と Skylake<sup>†</sup> マイクロアーキテクチャー・ベースのインテル® Core™ プロセッサ: 1 次キャッシュ、2 次キャッシュ、および 3 次キャッシュ
  - Broadwell<sup>†</sup> と Skylake<sup>†</sup> マイクロアーキテクチャー・ベースのインテル® Xeon® プロセッサ: 1 次キャッシュ、2 次キャッシュ、および 3 次キャッシュ

### 7.3.3 プリフェッチとロード命令

最近の世代のマイクロアーキテクチャーは、非干渉型の実行とメモリー・パイプラインを備えています。これにより命令にデータやリソースの依存性がない場合、メモリーアクセスとは独立して複数の命令を実行することが可能となります。プログラムやコンパイラーは、ダミーロード命令を使用して PREFETCH 命令の機能を模倣できますが、事前ロードと PREFETCH 命令は完全に同じではありません。PREFETCH 命令の方がプリロードよりもパフォーマンス面で優れています。

それは、PREFETCH 命令 が次の特長を備えているためです。

- Y デスティネーション・レジスターがなく、キャッシュラインをいくつか更新するだけです。
- Y 通常の命令のリタイアメント処理をストールさせません。
- Y プログラムの機能面での動作に影響しません。
- Y キャッシュラインの分割アクセスを行いません。
- Y LOCK プリフィクスを使用している場合を除いて、例外を発生しません。PREFETCH 命令とともに使用される LOCK プリフィクスは、有効なプリフィクスではありません。
- Y PREFETCH 命令が原因でフォルトが発生すると、命令の実行は中断されます。

PREFETCH 命令のプリロード命令に対する優位性は、プロセッサの種類によって異なります。これも将来は変わる可能性があります。PREFETCH 命令でデータをプリフェッチできないことがあります。例えば、以下のようなケースです。

- Y 古いマイクロアーキテクチャーで、PREFETCH 命令を実行すると DTLB (データ・トランスレーション・ルックアサイド・バッファー) ミスが発生する場合、Nehalem<sup>†</sup>、Westmere<sup>†</sup>、Sandy Bridge<sup>†</sup> およびそれ以降の新しいマイクロアーキテクチャー、インテル® Core™2 プロセッサ、および Intel Atom® プロセッサでは、DTLB ミスを引き起こす PREFETCH 命令でページ境界を超えたデータをフェッチできます。
- Y 指定されたアドレスにアクセスするとフォルトまたは例外が発生する場合。
- Y 1 次キャッシュと 2 次キャッシュの間にあるリクエストバッファーを、メモリー・サブシステムが使い切った場合。
- Y USWC や UC など、キャッシュできないメモリー領域に対して PREFETCH を行った場合。
- Y LOCK プリフィクスが使用されている場合、LOCK プリフィクスを使用すると、不正オペコード例外が発生します。

## 7.4 キャッシュ制御

この節では、キャッシュ制御命令の仕組みについて説明します。

## 7.4.1 非テンポラルなストア命令

この節では、ストリーミング・ストアの動作について説明し、前の節に示した情報についても一部繰り返して触れています。

インテル® ストリーミング SIMD 拡張命令の MOVNTPS、MOVNTPD、MOVNTQ、MOVNTDQ、MOVNTI、MASKMOVQ、MASKMOVDQU といった命令を実行すると、ストリーミング方式の非テンポラルなストアが行われます。メモリーの特性と順序付けに関しては、これらの命令はライト・コンバイニング (WC) メモリータイプと同様です。

- Y ライト・コンバイニング (Write combining): 同一キャッシュラインに対する連続書き込みが結合されます。
- Y ライトコラプス (Write collapsing): 同一バイトに対して連続書き込みを行ったとき、最後に書き込んだ内容のみが反映されます。
- Y 強制力の弱い順序付け (Weakly ordered): WC ストア同士の間にも、あるいは WC ストアとほかのロード/ストアの間にも順序付けは設定されません。
- Y キャッシュが不可能で、ライトアロケート (write-allocating) が実行されない: ストアされたデータはキャッシュのあちこちに書き込まれるため、対応するキャッシュラインに対する所有権読み込みバス要求は生成されません。

### 7.4.1.1 フェンス操作

ストリーミング・ストアの順序付けは強制力が弱いいため、ストアされたデータをプロセッサからメモリーへ確実にフラッシュするにはフェンス操作が必要になります。フェンス操作を適切に行わないと、プロセッサ内にデータが文字どおり「閉じ込められる」結果になりかねず、そうなると、ほかのプロセッサやシステム・エージェントから見えなくなります。

WC ストアを実行するには、ソフトウェアでフェンス操作を行いデータの整合性を保証する必要があります。詳細は 7.4.5 節「FENCE 命令」をご覧ください。

### 7.4.1.2 ストリーミング方式の非テンポラルなストア

ストリーミング・ストアでは、以下のようにパフォーマンスが改善できます。

- Y キャッシュラインに収まる 64 バイトが連続的に書き込まれる場合はストア帯域幅が増加します (所有権読み出しバス要求が必要なく、64 バイトが単一のバス書き込みトランザクションとして結合されるため)。
- Y キャッシュされたテンポラルなデータのうち、頻繁に使用されるデータの乱れが減少します (プロセッサ・キャッシュのあちこちにキャッシュデータが書き込まれるため)。

一定のメモリー領域にストリーミング・ストアを使用すると、複数のメモリータイプによるクロスエイリアシングが可能になります。例えば、領域の 1 つを、ページ属性テーブル (PAT) やメモリー・タイプ・レンジ・レジスター (MTRRS) を使用してライトバック (WB: write-back) メモリータイプとしてマッピングすることもあります。ストリーミング・ストアで書き込みを行います。

### 7.4.1.3 メモリータイプと非テンポラルなストア

メモリータイプが非テンポラルなヒントよりも優先されることがあるため、以下のような点を検討する必要があります。

- Y プログラマーが、強制力の高い順序付けのキャッシュ不可能メモリー (キャッシュ不可能 (UC) やライトプロテクト (WP) のメモリータイプなど) に非テンポラルなストアを指定した場合、そのストア操作は、キャッシュできないストア操作と同じように動作します。その結果、非テンポラルなヒントは無視され、その領域のメモリータイプは保持されます。



- Y プログラマーが、強制力の弱いライト・コンバイニング (WC) のキャッシュ不可能メモリータイプを指定した場合、非テンポラルなストア操作とその領域が同じセマンティクスとなるため、競合は発生しません。
- Y プログラマーが、キャッシュ可能メモリー (ライトバック(WB) かライトスルー (WT) のメモリータイプなど) に非テンポラルなストアを指定した場合、以下の 2 つの結果となる可能性があります。
  - ケース 1: データがキャッシュ階層の中に存在している場合は、その命令によって整合性が保証されます。プロセッサによっては、別の方法でこれと同じ操作を行うものもあります。次のような方法が考えられます。(a) 当該領域に指定されたメモリー・タイプ・セマンティクスを保存する一方、キャッシュ階層の所定の位置にあるデータを更新します。(b) キャッシュからデータを排出し、WC セマンティクスで新しい非テンポラルなデータをメモリーに書き込みます。

(分割方式にしても結合方式にしても)、このような方式は将来のプロセッサでは変わる可能性があります。インテル® Pentium® 4 プロセッサ、インテル® Core™ Solo プロセッサ、インテル® Core™ Duo プロセッサでは、後者の方式 (すべてのプロセッサ・キャッシュからデータを排出する方式) が採られています。インテル® Pentium® M プロセッサでは、(a) と (b) の両方を組み合わせたものが実装されています。

1 次キャッシュに含まれているラインの 1 つにストリーミング・ストアがヒットした場合、そのストアデータは 1 次キャッシュ内のデータと結合されます。2 次キャッシュに含まれているラインの 1 つにストリーミング・ストアがヒットした場合は、そのラインとストアされたデータが 2 次キャッシュからシステムメモリーへフラッシュされます。

- ケース 2: データがキャッシュ階層の中に存在していない場合や、デスティネーション領域が WB または WT としてマッピングされている場合は、そのトランザクションの順序付けは強制力が弱くなり、すべて WC メモリー・セマンティクスに支配されることになります。非テンポラルなストアでは、ライトアロケートは行われません。実装方式が異なると、ストア操作のコラプス (折り畳み) とコンバイン (結合) が行われることもあります。

#### 7.4.1.4 ライトコンバイン

一般的に WC セマンティクスでは、グラフィックス・カードなどほかのプロセッサやほかのシステム・エージェントに対してソフトウェアで整合性を保証する必要があります。生産 (producer) と消費 (consumer) モデルでは、同期とフェンス操作を正しく行う必要があります (7.4.5 節「FENCE 命令」を参照してください)。

フェンス操作を実行すると、ストアされたデータはどのシステム・エージェントからも認識できるようになります。逆に、フェンス操作に失敗すると、書き込まれたキャッシュラインがプロセッサの中に閉じ込められたままとなり、そのキャッシュラインはほかのエージェントからは見えなくなります。

キャッシュ階層にすでに存在しているデータを更新する非テンポラルなストア操作を実装しているプロセッサは、デスティネーション領域も WC としてマッピングしなければなりません。例えば WB または WT としてマッピングした場合、プロセッサによるスペキュレーティブな読み取り操作によってデータがキャッシュに移動する可能性があります。この場合、非テンポラルなストア操作による更新は実行されますが、その後フェンス操作を行ってもデータはプロセッサからフラッシュされません。

メモリー・タイプ・エイリアシングが発生しているバス上で認識できるメモリータイプは、実装方式によって異なります。例えば、バスに書き込まれたメモリータイプは、プログラムの実行順で最初にキャッシュラインにストアされたメモリータイプが反映されることもあります。ただし、ほかにも可能性はあります。このような機能は予備的なものと見なした方が良いでしょう。特定の方式で実装してしまうと、将来互換性を維持できなくなる可能性があります。

#### 7.4.2 ストリーミング・ストアの利用モデル

ストリーミング・ストアは、主にコヒーレント要求と非コヒーレント要求に使用されます。

## 7.4.2.1 コヒーレント要求

コヒーレント要求とは、システムメモリーに対する通常のロード操作/ストア操作のことです。マルチプロセッサ環境では、別のプロセッサのキャッシュラインにヒットすることもあります。コヒーレント要求では、WC メモリータイプ (PAT か MTRR) でマッピングされている通常のストアと同じ方法でストリーミング・ストアを使用できます。複数プロセッサ間でデータの整合性と認識性を保証するため、生産 (producer) と消費 (consumer) モデルでは SFENCE 命令を使用する必要があります。

シングル・プロセッサ・システムでは、CPU が同じメモリー・ロケーションを読み直すと整合性が保証されます (つまり、このメモリー・ロケーションにアクセスしたときにいつも決まった同じ内容が見えます)。マルチプロセッサ (MP) システムにおいても、生産と消費の同期を行うような MP ソフトウェアが採用されていれば、シングル・プロセッサ・システムと同じことが言えます。

## 7.4.2.2 非コヒーレント要求

非コヒーレント要求は、AGP グラフィックス・カードなど I/O デバイスから発行されます。これらのデバイスは、非コヒーレント要求によりシステムメモリーに対してデータの読み書きを行います。その要求はプロセッサ・バスには反映されないため、プロセッサ・キャッシュへの照会が行われません。複数プロセッサ間でデータの整合性と認識性を保証するため、生産 (producer) と消費 (consumer) モデルでは SFENCE 命令を使用する必要があります。この場合、当該プロセッサが I/O デバイスにデータを書き込んでいるのであれば、ケース 1 (7.4.1.3 節) のように動作するプロセッサでストリーミング・ストアを使用できます。ただし、その領域 (PAT、MTRR) も WC メモリータイプでマッピングされている場合に限られます。

### 注意

対象領域を WC としてマッピングできないと、そのキャッシュラインが投機的にプロセッサ・キャッシュに読み取られる場合があります (誤った分岐予測により間違っただけのパスを通ることになります)。

対象領域が WC としてマッピングされていない場合は、そのストリーミングによってキャッシュのデータが更新されることがあり、その後 SFENCE 命令を実行しても、データがシステムメモリーに書き込まれません。この場合、対象領域を WC として明示的にマッピングすることで、その領域から読み取られたデータのデータも、プロセッサのキャッシュには格納されません。非コヒーレント方式の I/O デバイスからこのメモリー・ロケーションを読み取ると、不正な結果か最新でない結果が返されます。

ケース 2 (7.4.1.3 節) のみを実装しているプロセッサは、キャッシュされたデータのデータもストリーミング・ストアによってメモリー・フラッシュされるため、メモリー領域を WB としてマッピングしなくても、この非コヒーレント・ドメイン中でストリーミング・ストアを使用できます。

## 7.4.3 ストリーミング・ストア命令の説明

レジスターからメモリーへデータをストアするには、MOVNTQ/MOVNTDQ 命令を使用します。これは、インテル® MMX® テクノロジーまたはインテル® ストリーミング SIMD 拡張命令レジスターに格納されているパックド整数を非テンポラルにストアするものです。この命令は、暗黙的に強制力の弱い順序付けを持ち、ライトアロケートは実行しないため、キャッシュ汚染を最小化します。

MOVNTPS 命令は MOVNTQ 命令に似ており、パックド単精度浮動小数点データを非テンポラルにストアします。これは、インテル® ストリーミング SIMD 拡張命令レジスターから 16 バイト単位でメモリーへデータをストアします。MOVNTQ 命令とは異なり、メモリーアドレスを 16 バイト境界にアライメントする必要があります。そうしないと、一般保護例外エラーが発生します。この命令は、暗黙的に強制力の弱い順序付けを持ち、ライトアロケートは実行しないため、キャッシュ汚染を最小化します。

レジスターから、EDI レジスターで指定したロケーションヘデータをストアするには、MASKMOVQ/MASKMOVDQU 命令を使用します。これは、インテル® MMX® テクノロジーまたはインテル® ストリーミング SIMD 拡張命令レジスターに格納されているパックド整数を非テンポラルにバイト・マスク・ストアするものです。マスクレジスターの各バイトの最上位ビットを使用して、バイト単位でソースレジスターの対応するデータを選択的に書き込みます。この命令は、暗黙的に強制力の弱い順序付けを行うため、連続してストア操作を実行しても、メモリーに書き込まれる順番が元のプログラムと一致しないことがあります。また、ライトアロケートが実行されないため、キャッシュ汚染を最小化します。

## 7.4.4 ストリーミング・ロード命令

インテル® SSE4.1 では、MOVNTDQA 命令が導入されました。メモリーソースが WC タイプである場合、MOVNTDQA 命令は非テンポラルなヒントを使用してメモリーから 16 バイトをロードします。WC メモリータイプでは、このデータをキャッシュせずに、アライメントの合ったキャッシュラインと同等のものを一時内部バッファにロードすることによって、非テンポラルなヒントを実装します。バッファリング済み WC データの未読み出し部分に対して後続の MOVNTDQA 命令が読み出しを行うと、データが利用可能な場合、16 バイト・データが一時内部バッファから XMM レジスターに転送されます。

MOVNTDQA 命令を適切に使用すると、WC メモリー領域内のデータをプロセッサにロードする際、ほかの手段よりも大幅に高いスループットを達成できます。

MOVNTDQA 命令の利用法に関連するアプリケーション・ノートの参照先は、第 1 章に記載されています。また、MOVNTDQA 命令を適切な利用方法に関する詳細情報と条件については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 12 章「Programming with SSE3, SSSE3 and SSE4」と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の MOVNTDQA 命令のリファレンス・ページを参照してください。

## 7.4.5 FENCE 命令

FENCE 命令には、SFENCE、LFENCE、MFENCE があります。

### 7.4.5.1 SFENCE 命令

SFENCE (STORE FENCE) 命令は、プログラム中の SFENCE よりも前に実行されたすべての STORE 命令が、SFENCE の後に実行される STORE 命令より先にメモリーに反映されることを確実にします。SFENCE は、順序付けの弱い結果を生成するルーチン間で、順序付けを確実にする方法の 1 つです。

生産と消費の関係など、特定のデータ共有関係によっては、順序付けの弱いメモリータイプの更新が重要になることがあります。順序付けの弱いメモリー更新では、効率良くデータを管理できますが、生産スレッドが更新したデータを確実に消費スレッドに渡すには、順序付けの注意が必要です。

一般的な使用モデルには、順序付けの弱いストア操作の影響を受けるものがあります。次に例を示します。

- Y ライブラリー関数。実行結果を書き込むときに、順序付けの弱いメモリー更新を行います。
- Y コンパイラーが生成したコード。これも、順序付けの弱いメモリー更新によって実行結果を書き込むことで利点がえられます。
- Y 手動で作成したコード。

順序付けの弱いデータ更新が行われていることをデータ消費スレッドがどの程度認識しているかは、上記のケースごとに異なります。したがって、順序付けの弱いデータを生産するルーチンと、そのデータを消費するルーチンの間で順序付けを確実にする場合は、SFENCE を使用しなければなりません。

## 7.4.5.2 LFENCE 命令

LFENCE (LOAD FENCE) 命令は、プログラム中の LFENCE よりも前に実行されたすべての LOAD 命令が、LFENCE の後に実行される LOAD 命令より先にメモリーを読み出すことを確実にします。

LFENCE 命令によって、特定の LOAD 命令を他の LOAD 命令から分離することが可能となります。

## 7.4.5.3 MFENCE 命令

MFENCE (MEMORY FENCE) 命令は、プログラム中の MFENCE よりも前に実行されたすべての LOAD/STORE 命令が、MFENCE の後に実行される LOAD/STORE 命令よりも先にメモリーアクセスを完了することを確実にします。MFENCE によって、特定のメモリー参照命令を他のメモリー参照命令から分離することが可能になります。

ロードフェンスとストアフェンスには互いを拘束する順序付けがないため、LFENCE と SFENCE 命令を組み合わせても、MFENCE と同じ結果は得られません。言い換えると、ロードフェンス命令は前のストア命令の前に実行ができ、ストアフェンス命令は前のロード命令の前に実行できません。

プロセッサからのスペキュレーティブ・メモリー参照を確実にするため CLFLUSH 命令 (cache line flush) でフラッシュ操作を行う場合、フラッシュを妨げないように MFENCE 命令を使用する必要があります。詳細は 7.4.6 節「CLFLUSH 命令」をご覧ください。

## 7.4.6 CLFLUSH 命令

CLFLUSH 命令を実行すると、メモリー・ロケーションのバイト・アドレスを含むリニアアドレスに割り当てられたキャッシュラインは、プロセッサのすべてのキャッシュ階層 (データおよび命令) で無効化されます。無効化されると、そのコヒーレンス・ドメイン全体に無効になったことがいっせいに通知されます。キャッシュ階層の任意のレベルにあるキャッシュラインが、メモリーと整合がとれていない場合 (これを「ダーティー」といいます)、無効化される前にメモリーに書き戻されます。また、以下のような特長があります。

- Y 影響を受けるデータサイズはキャッシュのコヒーレンス・サイズ (CPUID 命令で列挙される) と同じです。大半は 64 バイトです。
- Y 影響を受けるキャッシュラインを含むページのメモリー属性は、この命令の動作に何の影響も与えません。
- Y CLFLUSH 命令は、すべての特権レベルで使用できますが、バイト・ロードに関連したアクセス権チェックやフォルトの影響を受けることがあります。

CLFLUSH 命令は、ほかの CLFLUSH 命令と互いに順序付けし、書き込み、ロック付きの読み込み-更新-書き込み命令、フェンス命令、同じキャッシュライン<sup>4</sup> への CLFLUSHOPT を順序付けします。CLFLUSHOPT は、異なるキャッシュラインの順序付けには影響しません。CLFLUSH とそのほかのメモリー・トラフィックのメモリー順序付けの変更に関する詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の第 3 章の CLFLUSH のリファレンス・ページと、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 8 章「Memory Ordering」を参照してください。

ビデオデータの使用モデルを例にみると、ビデオ・キャプチャー・デバイスが非コヒーレント方式のアクセスを実行してシステムメモリーにキャプチャー・ストリームを直接書き込んでいるとします。この非コヒーレント方式の書き込み操作はプロセッサ・バスにいっせいに通知されないため、プロセッサのキャッシュにある同じロケーションのコピーはフラッシュされません。したがって、プロセッサがキャプチャー・バッファーを読み直す前に CLFLUSH 命令を使用して、キャプチャー・バッファーの古いコピーもプロセッサ・キャッシュからフラッシュしなければなりません。

例 7-1 に、CLFLUSH 命令の疑似コードを示します。

<sup>4</sup>前述のマニュアルで推奨される CLFLUSH の順序付けでは、ソフトウェアで CLFLUSH の後に MFENCE を追加することが要求されました。CLFLUSH 命令を実装するすべてのプロセッサでは、上記に示した他の操作との相対的な順序付けを行うため、CLFLUSH の後の MFENCE は必須ではありません。

例 7-1 CLFLUSH を使用した疑似コード

```
while (!buffer_ready) {}
sfence
for(i=0;i<num_cachelines;i+=cacheline_size) {
    clflush (char *)((unsigned int)buffer + i)
}
prefnta buffer[0];
VAR = buffer[0];
```

CLFLUSH を使用してキャッシュラインをフラッシュするスループットの特性にはばらつきがあり、いくつかの要因に強く依存します。一般に、多数のキャッシュラインをフラッシュするため連続して CLFLUSH を使用すると、適切なサイズのバッファ (4KB 未満) のフラッシュよりもキャッシュラインごとに大きなコストがかかります。CLFLUSH のスループットは 1 桁に落ち込むかもしれません。変更状態のキャッシュラインをフラッシュすると、非変更状態のキャッシュラインをフラッシュするよりもコストがかかります。

### 7.4.7 CLFLUSHOPT 命令

CLFLUSHOPT 命令は、第 6 世代のインテル® Core™ プロセッサで導入されました。CLFLUSH と同様に、CLFLUSHOPT 命令を実行すると、メモリー・ロケーションのバイト・アドレスを含むリニアアドレスに割り当てられたキャッシュラインは、プロセッサのすべてのキャッシュ階層 (データおよび命令) で無効化されます。

CLFLUSHOPT 命令の実行は、ロック付きの読み込み-変更-書き込み (read-modify-write) 命令、フェンス命令、そして無効化されたキャッシュラインへの書き込みを順序付けします (CLFLUSH と CLFLUSHOPT は、同じキャッシュラインを順序付けします)。無効化されているキャッシュライン以外への書き込みは順序付けされません (CLFLUSH と CLFLUSHOPT は、異なるキャッシュラインの順序付けには影響しません)。ソフトウェアは、CLFLUSHOPT と CLFLUSHOPT で順序付けされるべき別のキャッシュラインへのストアとの間に SFENCE を挿入できます。

一般に、CLFLUSHOPT のスループットは CLFLUSH よりも高くなっています。これは、上記と 7.4.6 節で説明したように、CLFLUSHOPT はより小さなセットのメモリー・トラフィックに対し自身を順序付けするためです。CLFLUSHOPT のスループットにもばらつきがあります。CLFLUSHOPT を使用して変更 (M) 状態のキャッシュラインをフラッシュすると、非変更状態のキャッシュラインをフラッシュするよりも高いコストを伴います。CLFLUSHOPT は、キャッシュラインがどのようなコヒーレンス状態であっても CLFLUSH を超えるパフォーマンスを提供します。(数キロバイトを超えるような) 大きなバッファをフラッシュする場合、CLFLUSH よりも CLFLUSHOPT の方が適しています。Skylake<sup>†</sup> マイクロアーキテクチャーでは、シングルスレッドのアプリケーションが CLFLUSHOPT を使用してバッファをフラッシュすると、CLFLUSH を使用するよりも最大 9 倍のパフォーマンスが得られます。

図 7-1 に、いくつかのバッファサイズ (1K - 16K バイト) で CLFLUSHOPT と CLFLUSH を実行したパフォーマンス特性の比較を示します。

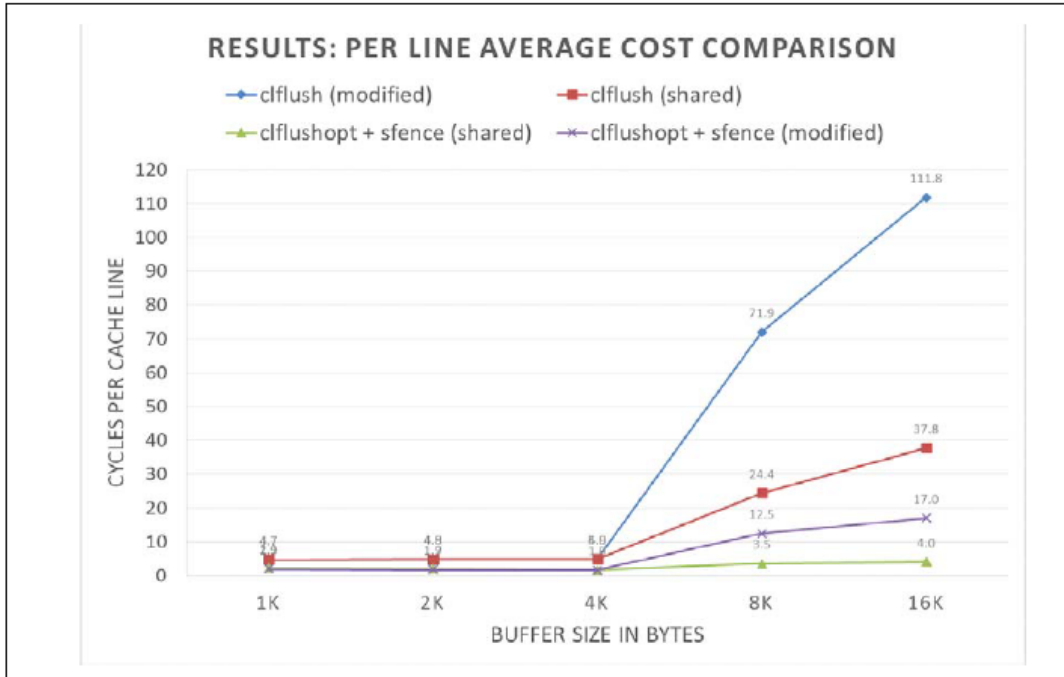


図 7-1 Skylake<sup>+</sup> マイクロアーキテクチャーにおける CLFLUSHOPT と CLFLUSH

**ユーザー/ソース・コーディング規則 17:** CLFLUSHOPT が利用できる場合、CLFLUSH よりも CLFLUSHOPT を使用します。CLFLUSHOPT が書き込みの順番をグローバルに反映することを確実にするため最後に SFENCE を挿入します。CLFLUSHOPT が利用できない場合、CLFLUSH を 4KB 未満の小さなチャンクで使用して、大きなバッファをフラッシュすることを検討してください。

例 7-2 に、CLFLUSH または CLFLUSHOPT を使用してキャッシュラインをフラッシュするアセンブリーのシーケンスを示します。次に対応する C のシーケンスを示します。

CLFLUSH:

```
for (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflush( &pBufferToFlush[ i ] );
```

CLFLUSHOPT:

```
_mm_sfence();
for (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflushopt( &pBufferToFlush[ i ] );
_mm_sfence();
```

例 7-2 CLFLUSH または CLFLUSHOPT を使用してキャッシュラインをフラッシュする

CLFLUSH には MFENCE が必要なくなりました	CLFLUSHOPT と SFENCE
<pre>xor rcx, rcx mov r9, pBufferToFlush mov rsi, iSizeOfBufferToFlush ;; mfence - 以前の方法 loop: clflush [r9+rcx] add rcx, 0x40 cmp rcx, rsi jl loop ;; mfence - 以前の方法</pre>	<pre>xor rcx, rcx mov r9, pBufferToFlush mov rsi, iSizeOfBufferToFlush sfence loop: clflushopt [r9+rcx] add rcx, 0x40 cmp rcx, rsi jl loop sfence</pre>
<p>*メモリーの順序付け規則がアプリケーションにとって重要である場合、メモリー書き込みの順番を保証するため CLFLUSHOPT 命令の実行は SFENCE 命令で保護される必要があります。上図のように、このような解決策は CLFLUSH 命令を使用するよりも高いパフォーマンスを示しますが、これは 2048 バイト以上のバッファの CLFLUSHOPT と同等です。</p>	

## 7.5 プリフェッチを使用したメモリの最適化

最近の世代のインテル® プロセッサには、2 つのデータ・プリフェッチ機構があります。1 つはソフトウェア制御プリフェッチであり、もう 1 つは自動ハードウェア・プリフェッチです。

### 7.5.1 ソフトウェア制御プリフェッチ

ソフトウェア制御プリフェッチを行うには、ストリーミング SIMD 拡張命令とともに導入された 4 つの PREFETCH 命令を使用します。これらの命令は、データを含むキャッシュラインを要求されるレベルとモデルのキャッシュ階層に移動する際のヒントとして機能します。ソフトウェア制御プリフェッチは、コードのプリフェッチには適しません。コードが共有されているときにソフトウェア制御プリフェッチを使用すると、マルチプロセッサ・システムのパフォーマンスが著しく低下することがあります。

ソフトウェア・プリフェッチには以下の特長があります。

- Y ハードウェア・プリフェッチャーがトリガーされない不規則なアクセスパターンに対応します。
- Y ハードウェア・プリフェッチに比べて、占有するバス帯域幅が低くなります。下記を参照してください。
- Y ソフトウェア・プリフェッチを行うには新しいコードを追加しなければならず、既存のアプリケーションには利点がありません。

### 7.5.2 ハードウェア・プリフェッチ

自動ハードウェア・プリフェッチは、事前に発生したデータミスが引き金となって、ユニファイド最終レベルキャッシュにキャッシュラインを取り込みます。自動ハードウェア・プリフェッチは、プリフェッチ・ストリームの前に 2 つのキャッシュラインをプリフェッチします。ハードウェア・プリフェッチャーには次のような特長があります。

- Y データ・アクセス・パターンには以下のような規則性が求められます。
  - データ・アクセス・パターンに一定した間隔がある場合、アクセス間隔がハードウェア・プリフェッチのトリガーとなる間隔の半分未満のときにハードウェア・プリフェッチは有効になります。
  - アクセス間隔が一定でない場合、連続する 2 つのキャッシュミスの間隔がトリガーしきい値未満であると (間隔の狭いメモリー・トラフィック)、自動ハードウェア・プリフェッチはメモリー・レイテンシーを隠匿できません。
  - 連続する 2 つのキャッシュミスの間隔がトリガーしきい値未満で、かつ 64 バイトに近い場合に、自動ハードウェア・プリフェッチは最も効果的です。
- Y プリフェッチャーが起動するまで時間がかかったり、処理の終わった後もフェッチが続行する可能性があります。配列が短いと、オーバーヘッドを相殺できない可能性があります。
  - ハードウェア・プリフェッチャーは、データミスが数回発生しないと起動しません。
  - ハードウェア・プリフェッチでは、データ配列の終端を超えるプリフェッチが行われます (本来そのデータは利用されません)。この動作はバス帯域幅を浪費します。さらにこの動作は、次の配列の先頭からフェッチを開始する際のペナルティーとなります。ソフトウェア・プリフェッチでは、このような状況を認識して対処できます。
- Y 4KB ページ境界をまたぐプリフェッチは行われません。ハードウェア・プリフェッチャーが新たなページからのプリフェッチを開始する前に、プログラムはそのページに対するロードを要求しなければなりません。
- Y キャッシュミスの間隔がハードウェア・プリフェッチのトリガーしきい値の距離よりも長く (間隔の長いメモリー・トラフィック)、アプリケーションのメモリー・トラフィックの大部分を占める場合、ハードウェア・プリフェッチャーは余分なシステム帯域幅を消費することがあります。
- Y 既存のアプリケーションで有効であるかどうかは、メモリー・トラフィックのアクセス間隔が長い、短いかに依存します。テンポラルな局所性に優れた、間隔の短いメモリー・トラフィックが多いアプリケーションでは、自動ハードウェア・プリフェッチから大きなメリットが得られます。

Y 一部の状況では、間隔が広くキャッシュミスが多いメモリー・トラフィックを、データ・アクセス・シーケンスの再配列によって、間隔の短いキャッシュミスが集中するようなトラフィックに変換できます。ただし、間隔が短いキャッシュミスでは自動ハードウェア・プリフェッチの利点を活用できません。

### 7.5.3 ハードウェア・プリフェッチで実効レイテンシーを削減する例

アクセス間隔が一定な循環ポインター追尾シーケンスに対応するデータを配列にする状況について考えてみます (例 7-3 を参照)。メモリーからキャッシュラインをフェッチする際に、自動ハードウェア・プリフェッチを利用して実効レイテンシーを削減するには、循環ポインター追尾を適用する配列へデータを配置するときに、64 バイト間のアクセス間隔や、ハードウェア・プリフェッチのトリガーしきい値の距離を変化させることによって実証できます。

例 7-3 一定間隔の循環ポインター追尾向けの配列の配置

```

register char ** p;
char *next;          // 一定のストライドで続く多重間接ポインターの追加
                    // p = (char **) *p; は次の位置の値をロード

p = (char **)&pArray;
for ( i = 0; i < aperture; i += stride) {
    p = (char **)&pArray[i];
    if (i + stride >= g_array_aperture) {
        next = &pArray[0 ];
    }
    else {
        next = &pArray[i + stride];
    }
    *p = next; // 次のノードのアドレスを設定
}
    
```

図 7-2 に、いくつかのマイクロアーキテクチャーでの実効レイテンシーの削減を示します。一定間隔のアクセスパターンでは、自動ハードウェア・プリフェッチのメリットは、トリガーしきい値の半分から現れ始めます。最大のメリットは、キャッシュミスの間隔が 64 バイトのときに得られます。

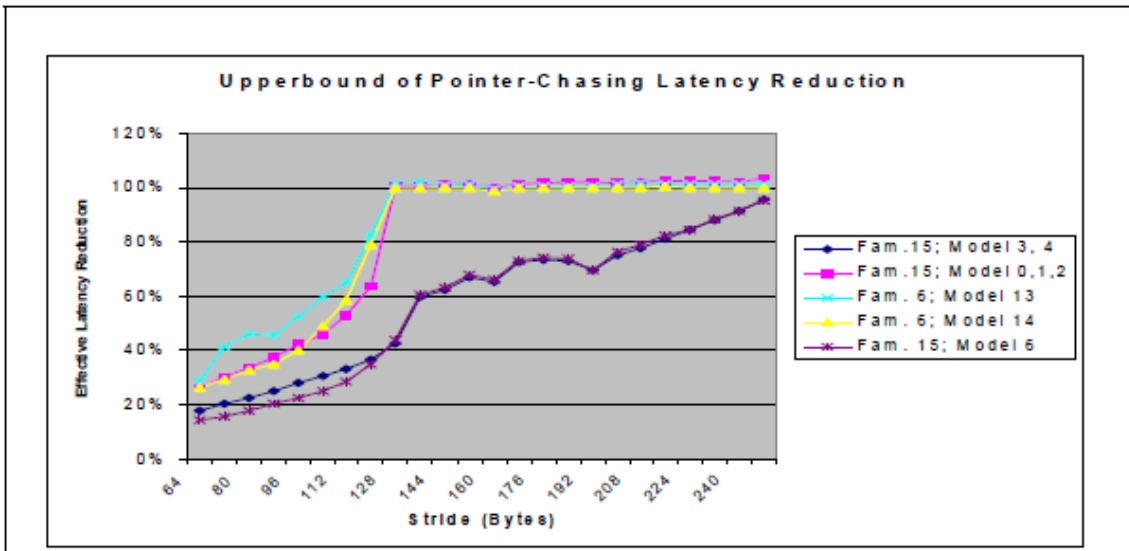


図 7-2 アクセス間隔に応じた実効レイテンシーの削減



## 7.5.4 ソフトウェア・プリフェッチ命令でレイテンシーを隠蔽する例

PREFETCH 命令を使用してメモリーを最大限に最適化するには、使用するマシンのアーキテクチャーを理解する必要があります。この節では、アーキテクチャーの持つ基本的な意味を、プログラマーが活用できるように簡単な形にして示します。

図 7-3 と図 7-4 に、単純化された 3D ジオメトリー・パイプラインの処理工程の 2 つの例を示します。通常、3D ジオメトリー・パイプラインは、バーテックス・レコードを一度に 1 つずつフェッチし、そのレコードに対して変換処理と照明処理を行います。どちらの図にも、実行パイプラインが 1 つにメモリー・パイプライン（フロントサイド・バス）が 1 つ、計 2 つの独立したパイプラインが描かれています。

インテル® Pentium® 4 プロセッサー以降では、(インテル® Pentium® II プロセッサーやインテル® Pentium® III プロセッサーと同じように) 実行機能とメモリーアクセス機能は完全に分離されているため、これら 2 つのパイプラインを同時に処理できます。図 7-2 は、実行とメモリー・パイプラインで、いわゆる「バブル」が発生する様子を表しています。この図では、バーテックス・データにアクセスするロード命令が発行されると、実行ユニットはアイドル状態のままデータが取得されるのを待機しています。一方、メモリーバスは、実行ユニットがバーテックス・データを処理している間、アイドル状態になります。この方法では、分離されたアーキテクチャーを全く活用していません。

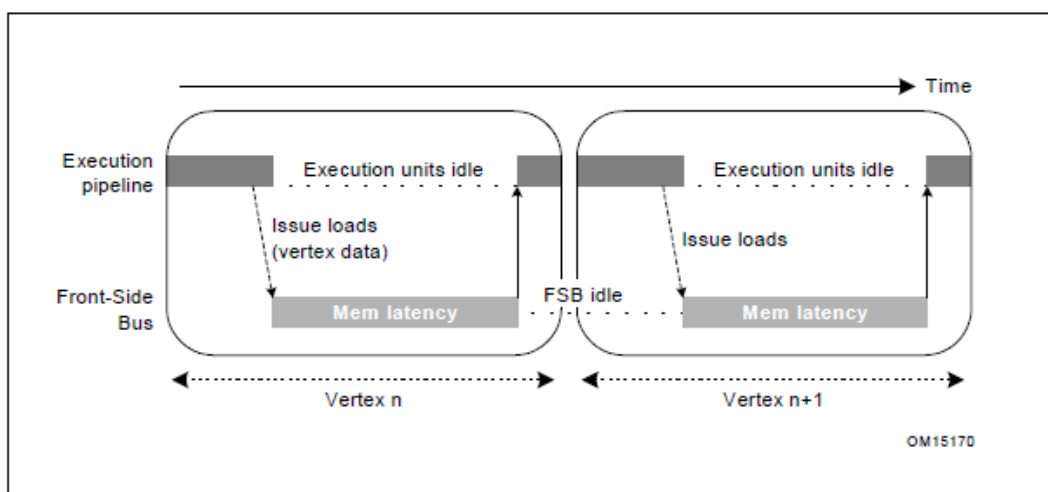


図 7-3 プリフェッチを使用しない場合のメモリー・アクセス・レイテンシーと実行

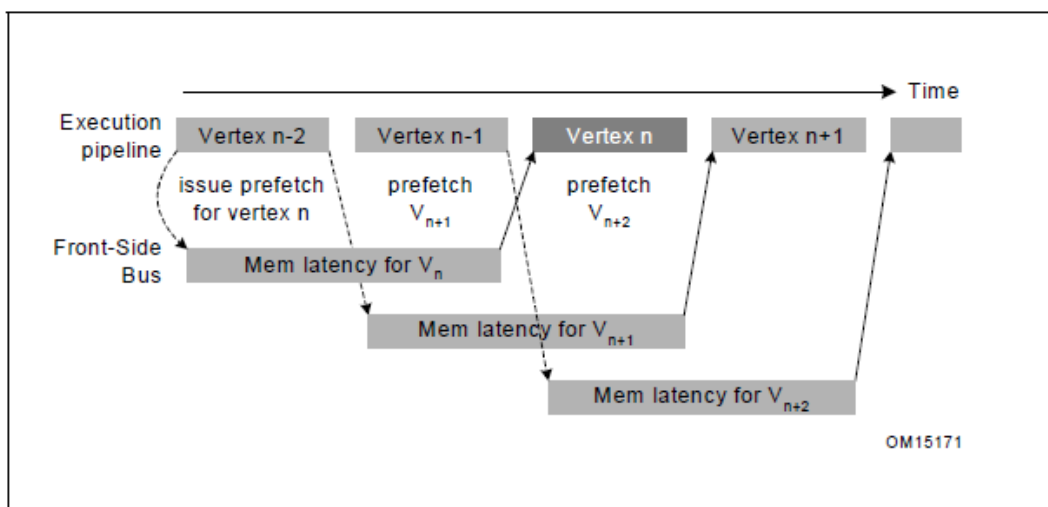


図 7-4 プリフェッチを使用した場合のメモリー・アクセス・レイテンシーと実行

リソースをうまく活用できないとパフォーマンスが低下しますが、複数の PREFETCH 命令を適切に使用すれば、そのようなパフォーマンスの低下を防ぐことができます。図 7-4 では、2 つ先のバーテックス・データに対して PREFETCH 命令を発行しています。これは、1 回の反復で処理するバーテックス・データが 1 つだけであり、反復ごとにデータのキャッシュラインが新たに 1 つずつ必要になる状況を想定しています。そのため、(反復  $n$ 、バーテックス  $V_n$ ) を処理する際に、要求されたデータはキャッシュにすでに取り込まれています。その間、フロントサイド・バスは反復  $n+1$ 、バーテックス  $V_{n+1}$  で必要となるデータを転送しています。 $V_{n+1}$  のデータと  $V_n$  での処理間に依存関係はないため、 $V_{n+1}$  のデータアクセスでレイテンシーが生じても、それを  $V_n$  の処理の背後に隠蔽できます。このような仕組みにすることで、パイプラインに「バブル」が発生しないため、可能な限り最高のパフォーマンスが得られます。

プリフェッチは、大量に計算を実行する内部ループに有用ですが、計算処理の制約とメモリー帯域幅の制約が同程度の内部ループにも有用です。プリフェッチは、メモリー帯域幅の制約が圧倒的に多いループにはそれほど有用ではありません。

すでにデータが 1 次キャッシュに格納されていると、プリフェッチを実行しても実益がなく、状況によってはパフォーマンスが低下することもあります。これは、余計なマイクロオペレーション ( $\mu\text{op}$ ) がメモリーアクセスを待機したり、場合によっては、そのマイクロオペレーション ( $\mu\text{op}$ ) がすべて無用になるためです。この動作は、プラットフォーム固有であり、将来は変わることもあります。

### 7.5.5 ソフトウェア・プリフェッチを使用する際の確認事項

ソフトウェア・プリフェッチ命令を正しく使用する際に、検討し解決しなければならない問題点を以下に示します。

- Y ソフトウェア・プリフェッチのスケジューリング間隔を決定します。
- Y ソフトウェア・プリフェッチを連結します。
- Y ソフトウェア・プリフェッチの回数を最小限にします。
- Y ソフトウェア・プリフェッチ命令と演算命令を混在させます。
- Y キャッシュ・ブロッキング手法 (例えばストリップマイニングなど) を使用します。
- Y シングルパス実行とマルチパス実行のバランスを考えます。
- Y メモリーバンクの競合問題を解決します。
- Y キャッシュ管理の問題を解決します。

上記の問題を、次の各節で説明します。

### 7.5.6 ソフトウェア・プリフェッチのスケジューリング間隔

コードの中にどのようにプリフェッチ命令を配置するのが理想であるかは、アーキテクチャーに関連する多くの要因によって異なります。そうした要因の中には、プリフェッチするメモリーの容量、キャッシュ・ルックアップ・レイテンシー、システムメモリー・レイテンシー、計算サイクルの予測などがあります。データをプリフェッチする理想的な間隔は、プロセッサやプラットフォームによって異なります。間隔が狭すぎると、プリフェッチを実行しても、フェッチ操作で発生するレイテンシーを計算処理の背後に隠蔽できません。逆に、あまり先行してプリフェッチを行うと、プリフェッチされたデータが必要になったときにはすでにキャッシュからフラッシュされている可能性があります。

「プリフェッチ距離」は曖昧な用語であるため、ここでは「プリフェッチ・スケジューリング間隔 (PSD)」を使用します。「PSD」の単位は反復回数です。大きなループでは、プリフェッチ・スケジューリング間隔は 1 に設定 (つまり反復を 1 回分だけ先行してプリフェッチ命令を実行) できます。ループの本体が小さいとき (つまり計算処理の少ないループ反復のとき) は、プリフェッチ・スケジューリング間隔を反復 1 回よりも大きくする必要があります。

PSD を求める単純な数式は、数学モデルから導かれます。

例 7-4 に、ループ本体の内側でプリフェッチを使用する場合を示します。この例では、プリフェッチ・スケジューリング間隔は 3 に設定されています。ESI は任意のラインを指すポインターであり、EDX は参照されるデータのアドレスです。xmm1 ~ xmm4 は、計算に使用されるデータです。例 7-5 では、反復 1 回につき 2 つの独立したデー

タ・キャッシュラインを使用しています。反復 1 回あたりに使用するキャッシュラインの数が 2 つよりも多い場合は、PSD を増やし、2 つよりも少ない場合は PSD を減らす必要があります。

#### 例 7-4 プリフェッチのスケジューリング間隔

```

top_loop:
    prefetchnta [edx + esi + 128*3]
    prefetchnta [edx*4 + esi + 128*3]
    .....
    movaps xmm1, [edx + esi]
    movaps xmm2, [edx*4 + esi]
    movaps xmm3, [edx + esi + 16]
    movaps xmm4, [edx*4 + esi + 16]
    .....
    .....
    add esi, 128
    cmp esi, ecx
    jl top_loop
    
```

### 7.5.7 ソフトウェア・プリフェッチの連結

メモリー・レイテンシーのペナルティーを被らなければ、実行パイプラインが最大スループットのときに最高のパフォーマンスが得られます。これは、ループ反復で連続して使用されるデータをプリフェッチすることで達成できます。メモリーのパイプライン処理が途切れると、パイプライン中にバブルが発生します。

パフォーマンスの面からこのような問題を説明するため、ストリップ (帯状) 形式の 3D バーテックスを処理する 3D ジオメトリー・パイプラインを例として使用します。ストリップは、バーテックスの並び順があらかじめ定義されていて、その並び順によって三角形がいくつか連続するリストを含んでいます。プリフェッチ命令がうまく配置されていないと、ストリップ境界上でメモリーのパイプライン処理が途切れますが、その様子は簡単に確認できます。この実行パイプラインは、ストリップごとに、最初の 2 回の反復処理が実行されている間ストールします。その結果、反復を 1 回終えるのに要する平均レイテンシーは 165 (FIX) クロックとなります。

メモリーのパイプライン処理が途切れると、メモリー・パイプラインと実行パイプラインの効率が悪くなります。パイプライン処理の途切れによる影響は、「プリフェッチ連結」と呼ばれる手法を適用して排除できます。この手法を使用すると、メモリーアクセス処理も実行処理も完全にパイプライン化できるため利用率は高まります。

複数のループが入れ子になっていると、内部ループの最後の反復から、対応する外部ループの次の反復までの間に、メモリーのパイプライン処理が途切れる場合があります。プリフェッチ命令を挿入するときは特別な注意を払わなければなりません。そうしないと、内部ループの最初の反復でロード命令がキャッシュミスして、データを取得するまでの間実行パイプラインがストールして、パフォーマンスが低下します。

例 7-5 では、A[iii][0] を含むキャッシュラインがプリフェッチされないため、毎回キャッシュミスが発生します。

これは、配列 A[iii] のどの要素もキャッシュに存在していないのを前提としています。メモリーのパイプライン処理がストールしてペナルティーが生じても、それは内部ループを反復するうちに償却されます。しかし、内部ループが短いと、さらに有害となることもあります。さらに、PSD の最後の反復における最後のプリフェッチが無駄になり、マシンのリソースを浪費します。プリフェッチ連結の手法が導入される理由は、メモリーのパイプライン処理の途切れによるパフォーマンス低下を防ぐためです。

例 7-5 プリフェッチ連結の使用例

```
for (ii = 0; ii < 100; ii++) {
    for (jj = 0; jj < 32; jj+=8) {
        prefetch a[ii][jj+8]
        computation a[ii][jj]
    }
}
```

プリフェッチ命令を連結すると、内部ループとそれに対応した外部ループの境界で、実行パイプラインにバブルが発生しないようにできます。単純に、内部ループの最後の反復をアンロールし、それに続く反復で使用されるデータのプリフェッチ・アドレスを指定するだけで、メモリーのパイプライン処理の途切れによるパフォーマンスの低下を完全に排除できます。例 7-6 に、変更したコードを示します。

例 7-6 内部ループの最後の反復を連結しアンロールした例

```
for (ii = 0; ii < 100; ii++) {
    for (jj = 0; jj < 24; jj+=8) { /* N-1 反復 */
        prefetch a[ii][jj+8]
        computation a[ii][jj]
    }
    prefetch a[ii+1][0]
    computation a[ii][jj] /* 最後の反復 */
}
```

データ・プリフェッチの例を示すこのコードは改善されており、計算時間がメモリー・レイテンシーよりも長いと仮定すると、メモリー・アクセス・レイテンシーのペナルティーは外部ループの最初の反復時だけで生じます。

入れ子になったループ計算に入る前に必要となる最初のデータ要素を得るためのプリフェッチ命令を挿入しておけば、外部ループの最初の反復処理が始まるときのペナルティーが完全に排除できるか、軽減できます。このように複雑でないハイレベルのコードを最適化すると、メモリーのパフォーマンスを飛躍的に改善できます。

## 7.5.8 ソフトウェア・プリフェッチの数を最小化する

プリフェッチ命令は、たとえクロックやメモリー帯域幅をほとんど要しないとしても、バスサイクル、マシンサイクル、リソースといった観点からは、必ずしも完全に自由に使用できるわけではありません。

プリフェッチを多用すると、フロントエンドで命令発行に関連する問題が発生したり、メモリー・サブシステムでリソース競合が発生するため、パフォーマンス低下につながる可能性があります。これは、対象となるループが小さかったり、ループが命令発行の制約を受ける場合などは、問題になることがあります。

プリフェッチの多用による問題を解決する方法の 1 つは、ループをアンロールしたり、ソフトウェアでループのパイプライン処理を実装して、プリフェッチの実行回数を減らすことです。図 7-5 のコード例に示した、プリフェッチとループのアンロールの使用方法を参照してください。この例では、先行して発行されたプリフェッチ命令のアドレスと、同じアドレス指定されるプリフェッチ命令がすべて取り除かれています。特にこの例では、元のループを 1 回アンロールすることによってプリフェッチ命令が 6 つ減っています。また、ほかのすべての反復に含まれている、条件分岐を実行する命令が 9 つ減っています。

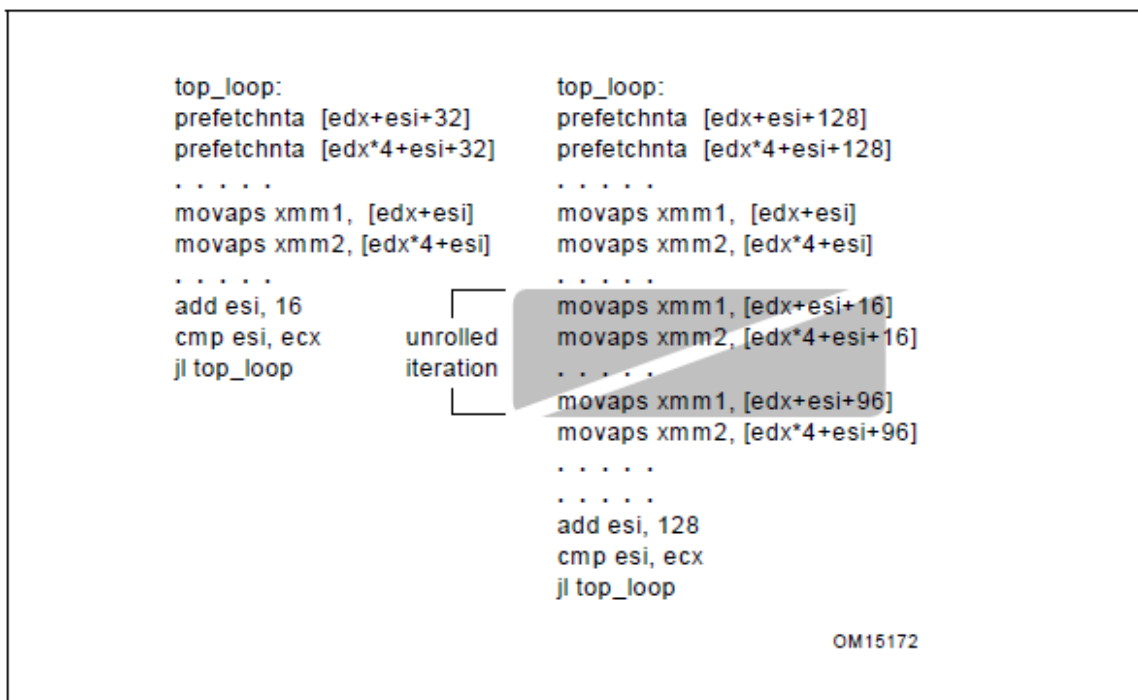


図 7-5 プリフェッチとループのアンロール

図 7-6 に、ソフトウェア・プリフェッチがレイテンシーを隠蔽する様子を図解します。

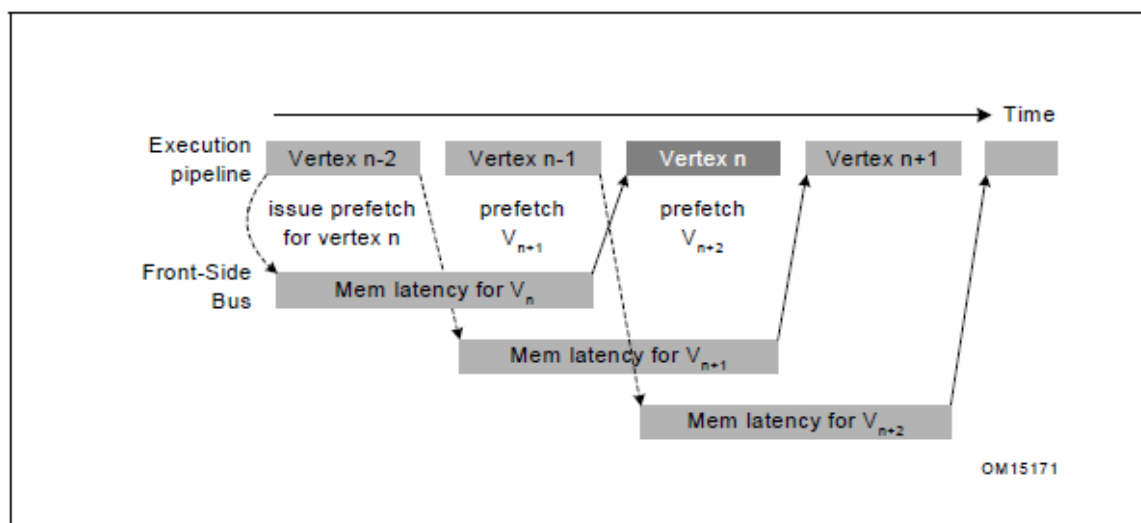


図 7-6 プリフェッチを使用した場合のメモリー・アクセス・レイテンシーと実行

図 7-6 の X 軸はループ 1 回あたりの計算クロック数であり、反復処理同士に依存関係はないものと仮定しています。左側の Y 軸には、ループ 1 回あたりのクロック数で測定した実行時間を示します。右側の Y 軸はバス帯域幅の利用率です。このテストでは、次の各要素を変数として実行しました。

- Y ロード/ストアストリームの数: ロードストリームもストアストリームもそれぞれ 1 回の反復で 128 バイトのキャッシュライン 1 つにアクセスします。
- Y ループ 1 回あたりの計算量: この変数を変えるときには、依存関係を持つ算術演算の実行回数を増やす方法を用います。
- Y ループ 1 回あたりのソフトウェア・プリフェッチの回数: それぞれ、16 バイト、32 バイト、64 バイト、128 バイトとして実行しています。

予想どおりですが、図 7-6 の左側を見ると分かるように、計算量が少なくてメモリアクセスのレイテンシーが重なり合うほどでないときには、プリフェッチを実行しても効果はなく、処理は基本的にメモリーのパフォーマンスに支配されます。また、余分なプリフェッチがあると、パフォーマンスが向上しないことも分かります。

### 7.5.9 ソフトウェア・プリフェッチ命令と演算命令を混在させる

ループ本体の始まりか、ループの前に PREFETCH 命令をすべてまとめて配置するのが便利のように思われますが、これは著しいパフォーマンス低下につながる可能性があります。最高のパフォーマンスを引き出すには、複数の PREFETCH 命令を集中して配置のではなく、命令シーケンスの中で、ほかの計算命令の間に分散して配置する必要があります。できるだけ PREFETCH 命令はロード命令から離して配置します。そうすることで、命令レベルでの並列性が高まり、命令リソースがストールする可能性も減少します。また、複数のプリフェッチ命令を分散するとメモリアクセス・リソースにかかる負担が減り、その結果、PREFETCH 命令がデータをフェッチしないままリタイアする可能性が減少します。

図 7-7 に、複数の PREFETCH 命令を分散する様子を示します。インテル® Pentium® 4 プロセッサでは、20 から 25 クロックごとに PREFETCH 命令を 1 つ挿入するのが有用なことが判明しています。キャッシュリソースに大きな負担がかかっているコードでは、PREFETCH 命令の位置を並べ替えるだけで驚くほどスピードアップすることがあります。

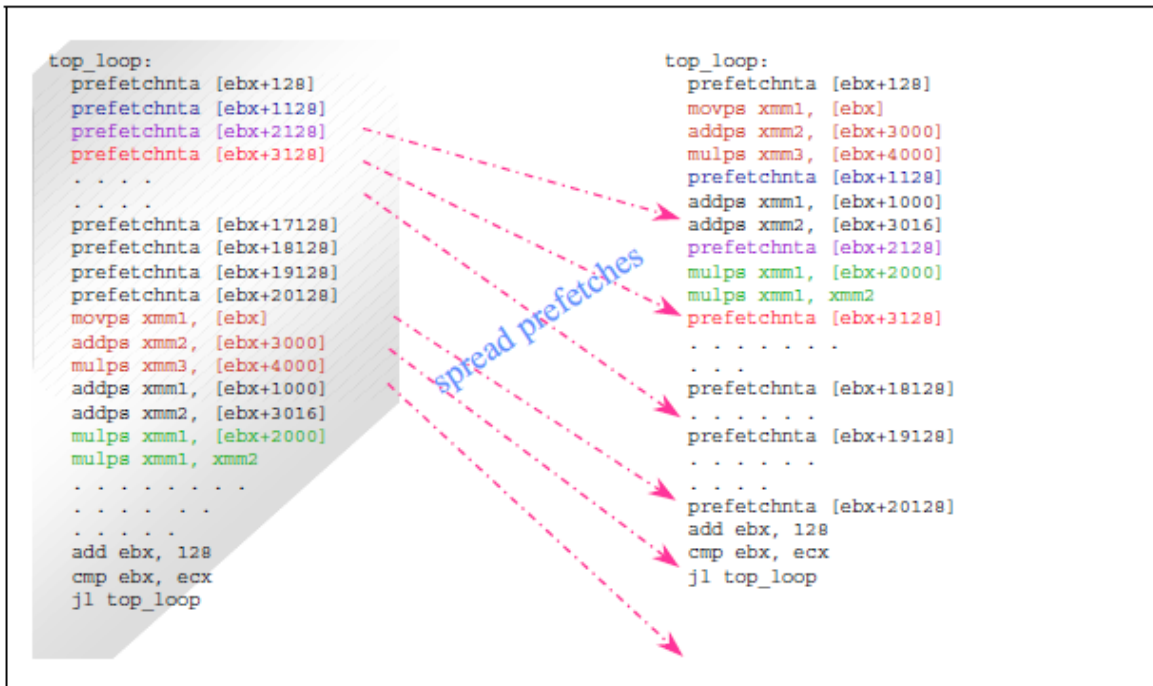


図 7-7 PREFETCH命令の分散

#### 注意

リソースを使いすぎると命令実行がストールする場合があります。それを避けるため、計算命令の間に PREFETCH 命令を分散します。

### 7.5.10 ソフトウェア・プリフェッチとキャッシュ・ブロッキング

時間的局所性を改善し、それによってキャッシュヒット率を上げるには、(ストリップマイニングなどに代表される) キャッシュ・ブロッキング手法を使用します。ストリップマイニングとは、メモリーの 1 次元的な時間的局所性を最適化する手法です。プログラムで 2 次元配列が使用されている場合、ループ・ブロッキング手法を利用してメモリーのパフォーマンスを改善できます。ループ・ブロッキング手法は、2 次元点以外ではストリップマイニングとほぼ同じです。

1 つのループを繰り返すときに再利用できる大きなデータセットを使用するアプリケーションであれば、ストリップマイニングの効果があります。キャッシュよりも大きなデータセットは、キャッシュに格納できるように小さないくつかのグループに分けて処理されます。こうすることで、より長い期間にわたってテンポラルなデータをキャッシュに保持できるため、バス・トラフィックが減少します。

ストリップマイニングされたコードに PREFETCH 命令をどのように適用するかは、データセットのサイズと時間的局所性（データ特性）によります。図 7-8 に、複数のデータが時間的に隣接している場合と、していない場合について単純化して示します。

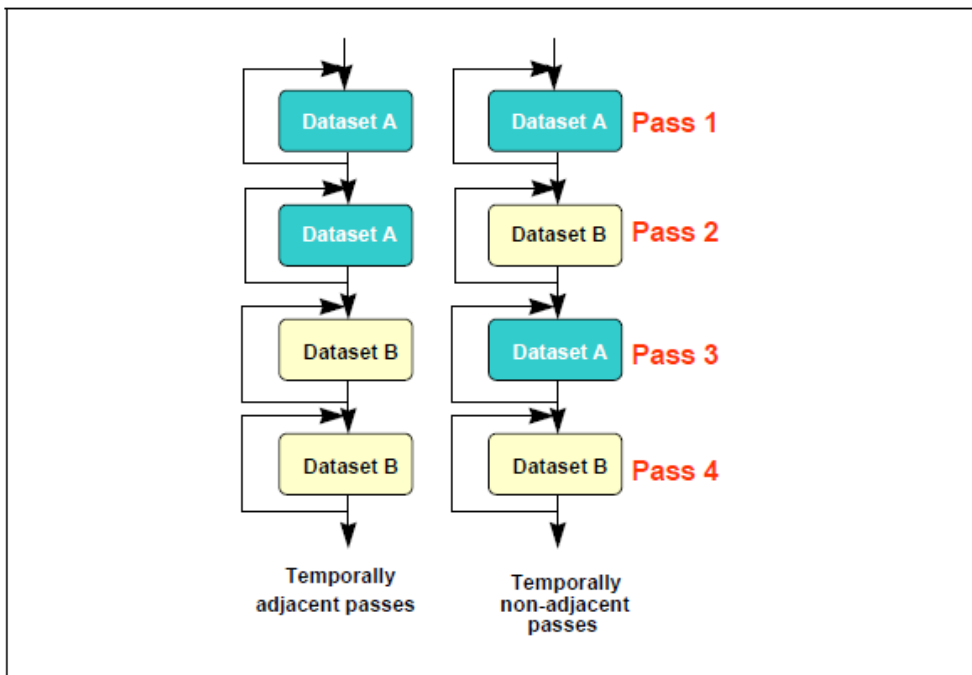


図 7-8 キャッシュ・ブロッキング - 複数のパスが時間的に隣接している場合としない場合

「時間的に隣接している場合」は、隣接している 2 つのパスのうち、後のパスでも同じデータが使われ、しかもそのデータは 2 次キャッシュにすでに格納されています。プリフェッチに関連する問題を別にすれば、こちらが望ましいでしょう。「時間的に隣接していない場合」は、 $m$  番目のパスで使われたデータが  $(m+1)$  番目のパスによって強制的に追い出されるため、データをフェッチし直して 1 次キャッシュに格納し直す必要があります。また、後のパスがそのデータを再利用するような場合は、2 次キャッシュにも格納する必要があります。データセットが両方とも 2 次キャッシュに入れば、パス 3、パス 4 でのロード操作はほとんど負担になりません。

図 7-9 は、プリフェッチ命令とストリップマイニングを適用するとパフォーマンスが向上することを示したものです。

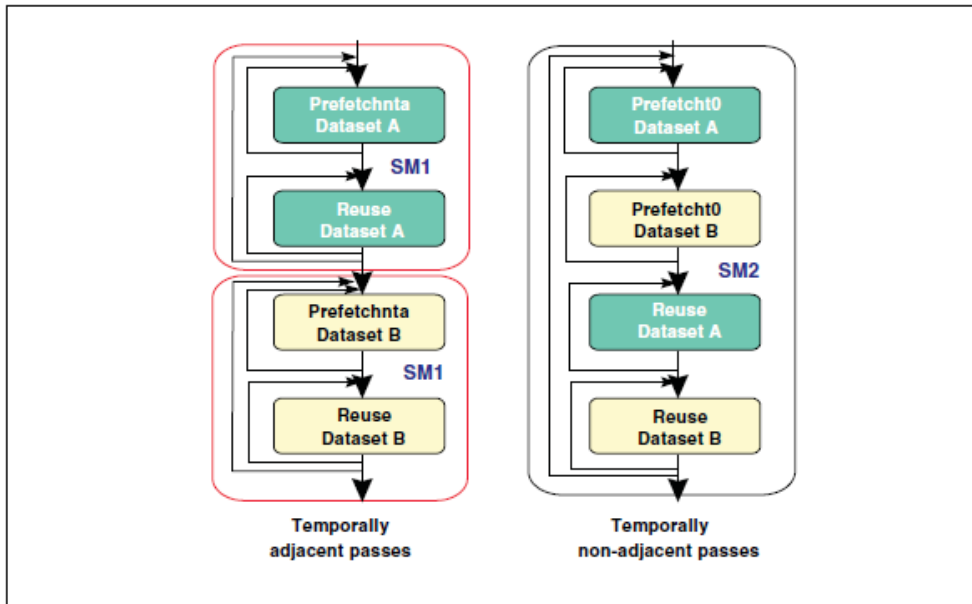


図 7-9 複数のパスループが時間的に隣接している場合と時間的に隣接していない場合でのプリフェッチとストリップマイニングの例

図 7-9 の左側は、インテル® Pentium® 4 プロセッサでの PREFETCHNTA 命令の使用例を示したものです。この例では、プリフェッチされたデータは、2 次キャッシュのウェイのうち選択されたいくつかのウェイにのみ格納され、2 次キャッシュの汚染は最小限に抑えられます。図中の「SM1」は、2 次キャッシュのウェイのうちの 1 つがストリップマイニングで処理されることを意味します。上位キャッシュでのキャッシュ汚染を最小限に抑えるため、実行パスの処理全体を通じて一度しか利用されないデータには PREFETCHNTA 命令を使用します。これは、プリフェッチ命令が十分に先行して発行されると仮定すれば、読み出しアクセス命令が発行されるとすぐに効果が表れます。

図 7-9 の右側では、2 次キャッシュのウェイのうちの 1 つにデータを格納し続けると、キャッシュの局所性が改善されないことを示しています。したがって、データのプリフェッチには PREFETCHT0 命令を使用します。これにより、パス 1、パス 2 でのメモリー参照に要するレイテンシーが隠蔽され、また、そのデータのコピーが 2 次キャッシュに保存されるため、パス 3、パス 4 でのメモリー・トラフィックとレイテンシーが減少します。このレイテンシーをさらに減らしたいときは、パス 3、パス 4 でメモリー参照を行う前にいくつか PREFETCHNTA 命令を使用することを検討してください。

例 7-7 を用いて、3D ジオメトリー・エンジンのデータ・アクセス・パターンを考えてみます。最初に、ストリップマイニングを使用しない例を示し、その後、ストリップマイニングを適用した例を示します。インテル® Pentium® III プロセッサの 4 ワイド SIMD 命令では、1 回の反復で 4 つのバーテックスを処理できます。

ストリップマイニングを使用しない場合は、4 つのバーテックス x、y、z 座標をすべて、2 番目のパス、すなわち照明計算ループのときに、メモリーからフェッチし直さなければなりません。そのため、照明計算ループで帯域幅が浪費されるのはもちろんですが、変換ループの最中にフェッチされたいくつかのキャッシュラインの利用率も低下します。



例 7-7 3D ジオメトリ・エンジンのデータアクセス (ストリップマイニングは未使用)

```

while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data // v =[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    TRANSFORMATION コード // バーテックス x,y,z,tu,tv のみを使用
    nvtx+=4
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data // v =[x,y,z,nx,ny,nz,tu,tv]
    // x,y,z の再フェッチ
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    照明ベクトルを計算 // x,y,z のみ使用
    LOCAL 照明コード // nx,ny,nz のみ使用
    nvtx+=4
}
    
```

次に、ストリップマイニングがループに適用された例 7-8 のコードについて考えてみます。

例 7-8 3D ジオメトリ・エンジンのデータアクセス (ストリップマイニングを使用)

```

while (nstrip < NUM_STRIP) {
/* データを L2 キャッシュの 1 つのウェイに収めるストリップマイニング */
    while (nvtx < MAX_NUM_VTX_PER_STRIP) {
        prefetchnta vertexi data // v =[x,y,z,nx,ny,nz,tu,tv]
        prefetchnta vertexi+1 data
        prefetchnta vertexi+2 data
        prefetchnta vertexi+3 data
        TRANSFORMATION コード
        nvtx+=4
    }
    while (nvtx < MAX_NUM_VTX_PER_STRIP) {
/* x y z 座標は L2 キャッシュ内にあり、プリフェッチを必要としない */
        照明ベクトルを計算
        POINT 照明のコード
        nvtx+=4
    }
}
    
```

ストリップマイニングを使用すると、ストリップマイニングが適用されたループ変換の処理中ずっとすべてのバーテックス・データをキャッシュ (例えば、2 次キャッシュのウェイの 1 つ) に保存でき、照明計算ループの中でそのバーテックス・データを再利用できるようになります。キャッシュにデータを維持しておく、パストラフィックもプリフェッチの使用回数も減少します。

ストリップマイニングとプリフェッチ命令を組み合わせた基本的な使用モデルの手順を表 7-1 にまとめています。手順は以下のとおりです。

- Y データセットが 2 次キャッシュに収まるようにループを分割するストリップマイニングを適用します。
- Y データが一度しか使われないか、データセットが 32KB (2 次キャッシュのウェイの 1 つ) に収まる場合は、PREFETCHNTA 命令を使用します。データセットが 32KB を超える場合は、PREFETCHT0 命令を使用します。

上記の手順はプラットフォームによって異なり、実装例の 1 つでしかありません。特定のプラットフォーム上で特定のアプリケーションを実行する場合に最高のパフォーマンスを引き出すには、NUM\_STRIP および MAX\_NUM\_VX\_PER\_STRIP 変数のヒューリスティックを決定する必要があります。

表 7-1 ストリップ・マイニング・コードへのソフトウェア・プリフェッチの組み込み

読み出しが 1 回の配列参照	読み出しが複数回の配列参照	
	隣接パス	非隣接パス
Prefetchnta	Prefetch0, SM1	Prefetch0, SM1 (2 次キャッシュ汚染)
1 方向への排出。汚染を最小限に抑制	各配列の最初のパスでメモリー・アクセス・コストを消費。以降のパスで最初のパスのコストを吸収	各ストリップの最初のパスでメモリー・アクセス・コストを消費。以降のパスで最初のパスのコストを吸収

### 7.5.11 ソフトウェア・プリフェッチとキャッシュ・ブロッキング

読み出しが複数回のメモリー参照の最初のパスと、読み出しが 1 回のメモリー参照の一部では、自動ハードウェア・プリフェッチ機構に合わせてデータ・アクセス・パターンをチューニングすると、メモリー・アクセス・コストを最小限に抑えられます。読み出しが 1 回のメモリー参照が行われる状況は、列方向優先で読み出し、行方向優先で書き込む (またはその反対)、行列またはイメージの転置によって例証できます。

例 7-9 に、典型的な行列/イメージ転置の問題である、ネストされたデータ移動ループを示します。配列の次元が大きい場合、データセットの要素が最終レベルキャッシュに収まらないだけでなく、広い間隔でキャッシュミスが発生します。次元が 2 の累乗である場合は、ウェイ・アソシアティビティーの有限数に基づくエイリアシング条件 (第 3 章の「キャッシュの容量制限とエイリアシング」を参照) によって、キャッシュ排出の可能性が増加します。

例 7-9 ハードウェア・プリフェッチを使用した、読み出しが 1 回のメモリー・トラフィックの向上

```

a) 最適化されていないイメージ転置
// dest と src は 2 次元配列
for(i = 0; i < NUMCOLS; i++) {
// 内部ループが 1 つの列を読み出す
    for(j = 0; j < NUMROWS; j++) {
        // 個々の読み出し参照は、ストライドの大きなキャッシュミスを引き起こす
        dest[i*NUMROWS + j] = src[j*NUMROWS + i];
    }
}

b)
// tilewidth = L2SizeInBytes/2/TileHeight/Sizeof(element)
for(i = 0; i < NUMCOLS; i += tilewidth) {
    for(j = 0; j < NUMROWS; j++) {
        // 内部ループで同じ行の複数の要素にアクセス
        // ハードウェア・プリフェッチしやすいパターンでアクセスし、ヒット率が改善
        for(k = 0; k < tilewidth; k++)
            dest[j+ (i+k)* NUMROWS] = src[i+k+ j* NUMROWS];
    }
}
    
```

例 7-9 (b) では、ハードウェア・プリフェッチに最適なタイルサイズとタイル幅を選択するタイル化手法を適用しています。タイル化によって、最終レベルキャッシュに収まるように、2 つのタイルのサイズを選択できます。メモリー読み出し参照で各タイルの幅を最大にすると、コードが実際にリニアアドレスを参照する前に、ハードウェア・プリフェッチはバス要求を開始してキャッシュラインを読み出せます。

## 7.5.12 シングルパス実行とマルチパス実行の比較

1 つのアルゴリズムで、シングルパス実行かマルチパス実行を選択できます。

- Y 「シングルパス実行」とは、「非階層化実行」とも呼ばれ、計算パイプライン全体でデータ要素を 1 つ通過させるものです。
- Y 「マルチパス実行」とは、「階層化実行」とも呼ばれ、複数のデータ要素から成る 1 つのデータ群を対象にして、パイプラインのステージを 1 段実行してからそのデータ群を次のステージに渡すものです。

シングルパス実行とマルチパス実行のどちらが良いかは、アルゴリズムの実装方式や、それら両者をどのように使用するかによって異なります。図 7-10 を参照してください。

汎用 API を実装している場合は、マルチパス実行が使いやすいことが多いでしょう。この場合、いくつかのコードパスのうちのどれが選択できるかは、対象となるアプリケーションが選ぶパターン構成要素間の組み合わせによって異なります。例えば、3D グラフィックスの場合は、使用されるバーテックス要素の種類や光源の個数および種類がパターン構成要素に含まれます。

このように、考えられる並べ替えの範囲が広いと、コードサイズや妥当性検証の点から見ると、シングルパス実行は複雑になります。これは、並べ替えの処理ごとにコードシーケンスが 1 つずつ必要になるためです。例えば、A、B、C、D のパターン構成要素を持つオブジェクトがある場合、このオブジェクトは、パターン構成要素の一部（例えば A、B、D）を使用可能にできます。このステージではコードパスが 1 つ使われ、一方、使用可能になったパターン構成要素の別の組み合わせには別のコードパスが 1 つ与えられます。パターン構成要素を選択するためいくつかの条件節がパイプラインの各ステージ内に実装されている場合、そのステージごとに別々のパスとして実行する方が適しています。ストリップマイニングを使用すると、各ステージで処理されるバーテックスの数（例えば、バッチサイズ）を選択して、処理単位となるデータ群がどのパスを通過しても必ずプロセッサ・キャッシュに留めておけます。バーテックス群を現在のステージから次のステージへ、あるいは現在のパスから次のパスへ渡すときは、中間キャッシュバッファを使用します。

任意の時点で使用できるパターン構成要素の数の制約のあるアプリケーションには、シングルパス実行が向いています。シングルパス実行を用いると、マルチパスエンジンで発生する可能性のあるデータコピーの量を減らすことができます。図 7-10 を参照してください。

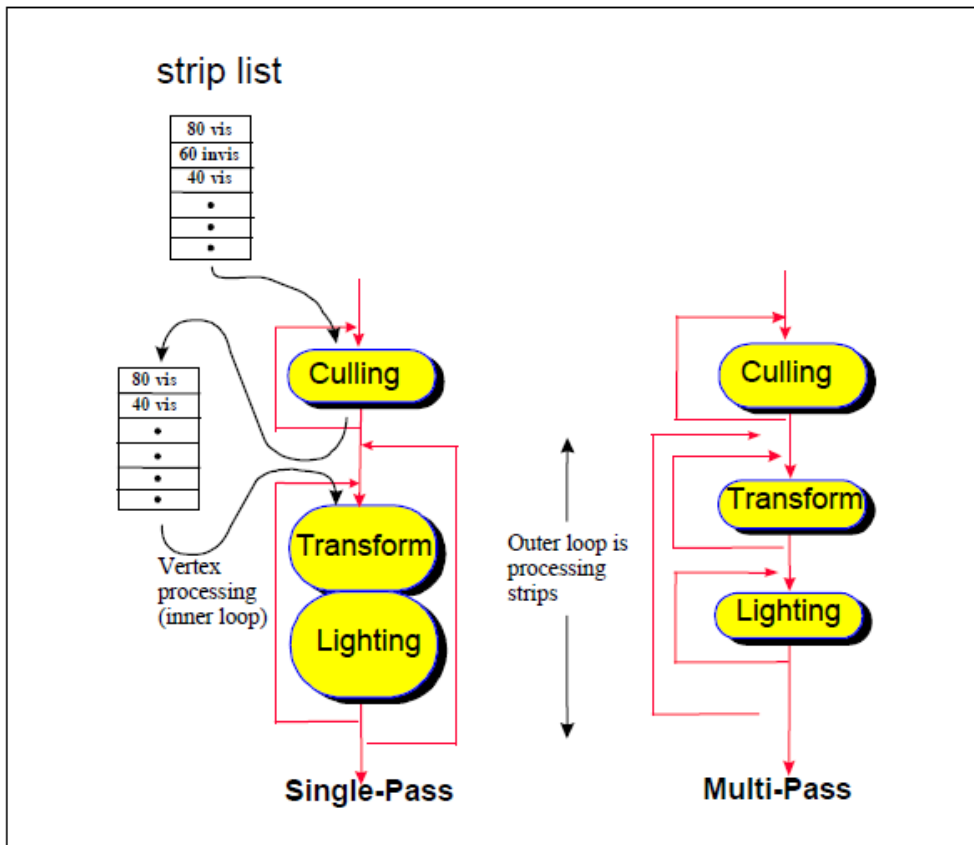


図 7-10 シングルパスとマルチパスでの 3D ジオメトリ・エンジンの比較

シングルパスとマルチパスのいずれかを選択することによって、パフォーマンスにどのような影響が出るかが異なります。例えば、マルチパスのパイプラインの場合、何段かのステージが入力または出力の帯域幅に制約されていると、それによるパフォーマンス低下の多くは全体的な実行時間として表れます。それに対し、シングルパスでは、帯域幅に制約があっても、それによるパフォーマンス低下は、計算量の多いほかの何段かのステージ全体に分散/吸収できます。また、シングルパスまたはマルチパスのどちらを使用するかは、どのプリフェッチ・ヒントを選択すべきかという問題にも影響します。

## 7.6 非テンポラルなストアを使用したメモリの最適化

キャッシュに保存するデータの管理には非テンポラルなストアも使用できます。非テンポラルなストアの用途には、次のものがあります。

- Y キャッシュ階層を乱すことなく、多くの書き込み処理を一体化します。
- Y いくつかのデータ構造のうちどれをキャッシュに残し、どれを残さないかを管理します。

上記の使用モデルの実装については、以降の節で詳しく説明します。

### 7.6.1 非テンポラルなストアとソフトウェアによるライトコンバイン

次のようなデータをストアする場合は、非テンポラルなストアを使用します。

- Y 非テンポラルな 1 回書き込み (ライトワンス) 用のデータ
- Y 大きすぎてキャッシュ・スラッシングの原因となるようなデータ

非テンポラルなストア命令を実行してもキャッシュラインの割り当ては行われません。つまり、非テンポラルなストア命令はライトアロケートではないため、有用なデータ帯域幅と競合してもキャッシュは汚染されず、ダーティーなライトバックも発生しません。

非テンポラルなストア命令を使用しないと、ダーティーなライトバックに起因するキャッシュ・スラッシングが発生する際に、バス帯域幅が悪影響を受けます。

インテル® ストリーミング SIMD 拡張命令では、複数の非テンポラルなストア命令が複数のライトバック・メモリー領域またはライト・コンバイニング・メモリー領域に書き込みを行っても、それらのストア命令の実行順は強制力が弱く、プロセッサのライト・コンバイニング・バッファの内部で結合され、1 つのライン・バースト・トランザクションとしてメモリーに書き込まれます。最高のパフォーマンスを引き出すためには、非テンポラルなストア命令を使用する一方で、そのキャッシュライン境界にデータをアライメントし、さらにそれらのデータを、同じキャッシュライン・サイズで連続して書き込むようにします。プログラミング上の制約から連続書き込みが禁止されている場合、ソフトウェア・ライト・コンバイニング (SWWC) バッファを使用して、ライン・バースト・トランザクションを有効にします。

アプリケーションで小さな SWWC バッファをいくつか (バッファごとにはキャッシュラインを 1 つ) 宣言して、明示的なライト・コンバイニング処理を有効にできます。このプログラムでは、データは非テンポラルなメモリーに直ちに書き込まれるのではなく、SWWC バッファに書き込まれた後バッファの内部で結合されます。プログラムは SWWC バッファが一杯になると、非テンポラルなストア命令を使って 1 キャッシュラインのバッファ (インテル® Pentium® 4 プロセッサでは 128 バイト) を書き出すだけです。このような SWWC を用いた方法では、一時的な読み出し/書き込み処理を実行する明示的な命令が必要になりますが、フロントサイド・バス上のトランザクションは、複数のパーシャル・トランザクションではなくライン・トランザクションが発生します。この方法を取り入れると、アプリケーションのパフォーマンスはかなり向上します。これらの SWWC バッファを 2 次キャッシュに保持すると、そのプログラム全体で何度でも使用できます。

## 7.6.2 キャッシュ管理

PREFETCH や STORE などのストリーミング命令を使用して、データを管理したり、プロセッサのキャッシュに格納されているテンポラルなデータの乱れを最小限にできます。

また、インテル® Pentium® 4 プロセッサでは、インテル® ストリーミング SIMD 拡張命令に対応した C++ 言語機能をサポートするインテル® C++ コンパイラを利用できます。インテル® ストリーミング SIMD 拡張命令およびインテル® MMX® 命令には、キャッシュの最適化を可能にする組込み関数が用意されています。インテル® コンパイラがサポートする組込み関数には、`_mm_prefetch`、`_mm_stream`、`_mm_load`、`_mm_sfence` などがあります。詳細は、『インテル® C/C++ コンパイラ・デベロッパー・ガイドおよびリファレンス』を参照してください。

次に、単純な 8 バイトのメモリーコピーを初め、ビデオ・エンコーダーやビデオデコーダーにおけるプリフェッチ命令の使用例を示します。プリフェッチ命令により効率良くキャッシュを管理すれば、高いパフォーマンスが引き出せることを理解できます。

### 7.6.2.1 ビデオ・エンコーダー

ビデオ・エンコーダーは、エンコード処理中に使用するデータの一部をプロセッサの 2 次キャッシュに格納します。これによりシステムメモリーから読み直さなければならない参照ストリームの数を最小限にできます。他の書き込みが 2 次キャッシュのデータを乱さないように、ストリーミング・ストア命令 (MOVNTQ) を使用してすべてのプロセッサ・キャッシュに書き込みを行います。

プリフェッチによるキャッシュ管理をビデオ・エンコーダーに実装すると、メモリー・トラフィックが減少します。一度しか使用しないビデオ・フレーム・データが 2 次キャッシュに入らないようにすると、2 次キャッシュの汚染が確実に減少します。非テンポラルな PREFETCH 命令 (PREFETCHNTA) を使用すると、2 次キャッシュのウェイの 1 つだけにデータが移動するため、2 次キャッシュの汚染が減少します。

2 次キャッシュに直接移動されたデータが再利用されない場合、非テンポラルなプリフェッチの方がテンポラルなプリフェッチよりもパフォーマンスを改善できます。2 次キャッシュの汚染を回避するため非テンポラルなプリフェッチ命令を使用するエンコーダーでは、2 次キャッシュでのヒット数が増え、汚染を引き起こすライトバックの回数が減少します。プリフェッチ命令だけでなく、2 次キャッシュの利用効率を高くすることで、パフォーマンスを改善できます。

### 7.6.2.2 ビデオデコーダー

ビデオデコーダーでは、完成したフレームデータがグラフィックス・カードの WC (ライト・コンバイニング) メモリータイプにマッピングされるローカルメモリーに書き込まれます。将来データを生成するため、プロセッサによって WB メモリーに参照データのコピーが格納されます。これは、参照データのサイズが大きすぎてプロセッサのキャッシュに収まらないことを想定したものです。ストリーミング・ストア命令によってデータをキャッシュ全体に書き込むと、キャッシュに格納されている他のテンポラルなデータが追い出されなくなります。後にプロセッサは、非テンポラル (NTA) なプリフェッチ命令である PREFETCHNTA 命令により目的のデータを再読み込みします。これによって、帯域幅が最大になる一方、キャッシュされたほかのテンポラルなデータの乱れが最小限に抑えられます。

### 7.6.2.3 ビデオ・エンコーダー/デコーダーの実装から導かれる結論

これらの 2 つの例からいえることは、非テンポラルなプリフェッチ命令と非テンポラルなストア命令を適切に組み合わせることで、2 次キャッシュの汚染が防止でき、有用なデータを 2 次キャッシュに残したまま、コストの高いライトバックのトランザクションを減らすことができます。プリフェッチ命令でデータの準備を整えてもそれほどパフォーマンスの向上が見込まれないアプリケーションでも、2 次キャッシュとメモリーを効率良く使用すれば改善可能です。そのような設計では、メモリーバスなど重要なリソースに対するエンコーダーの要求が減り、システムのバランスが増し、パフォーマンスはさらに高くなります。

### 7.6.2.4 メモリー・コピー・ルーチンの最適化

大量のデータをコピーするルーチンを作成することは、ソフトウェアの最適化における一般的なタスクです。例 7-10 に、単純なメモリーコピーを実行する基本アルゴリズムを示します。

例 7-10 単純なメモリーコピーの基本アルゴリズム

```
#define N 512000
double a[N], b[N];
for (i = 0; i < N; i++) {
    b[i] = a[i];
}
```

このタスクは、さまざまなコーディング手法を用いて最適化できます。その 1 つは、ソフトウェア・プリフェッチ命令とストリーミング・ストア命令を使用する方法です。この方法について以下で述べます。例 7-11 にコード例を示します。

インテル® ストリーミング SIMD 拡張命令を使用して、以下の点を考慮することでメモリーコピーのアルゴリズムを最適化できます。

- Y データのアライメント
- Y メモリー内の各ページの正しいレイアウト
- Y キャッシュサイズ
- Y トランスレーション・ルックアサイド・バッファ (TLB) とメモリーアクセスとの相互作用
- Y プリフェッチ命令とストリーミング・ストア命令の組み合わせ

本章で述べたガイドラインは、このような単純な例でもうまく作用します。インテル® Pentium® 4 プロセッサには、インテル® Pentium® III プロセッサの場合と全く同じように、TLB の初期化が必要です。これは、どちらのプロセッサでも、ソフトウェア・プリフェッチ命令を実行してもページテーブルのページウォークが開始されないためです。

例 7-11 ソフトウェア・プリフェッチを使用したメモリー・コピー・ルーチン

```
#define PAGESIZE 4096;
#define NUMPERPAGE 512 // ページに収まる要素数
double a[N], b[N], temp;
for (kk=0; kk<N; kk+=NUMPERPAGE) {
    temp = a[kk+NUMPERPAGE]; // TLB の準備
    // ブロックサイズ = ページサイズ、
    // ブロック全体をプリフェッチ、ループごとに 1 キャッシュラインを使用
    for (j=kk+16; j<kk+NUMPERPAGE; j+=16) {
        _mm_prefetch((char*)&a[j], _MM_HINT_NTA);
    }
    // ループごとに 128 バイトをコピー
    for (j=kk; j<kk+NUMPERPAGE; j+=16) {
        _mm_stream_ps((float*)&b[j],
        _mm_load_ps((float*)&a[j]));
        _mm_stream_ps((float*)&b[j+2],
        _mm_load_ps((float*)&a[j+2]));
        _mm_stream_ps((float*)&b[j+4],
        _mm_load_ps((float*)&a[j+4]));
        _mm_stream_ps((float*)&b[j+6],
        _mm_load_ps((float*)&a[j+6]));
        _mm_stream_ps((float*)&b[j+8],
        _mm_load_ps((float*)&a[j+8]));
        _mm_stream_ps((float*)&b[j+10],
        _mm_load_ps((float*)&a[j+10]));
        _mm_stream_ps((float*)&b[j+12],
        _mm_load_ps((float*)&a[j+12]));
        _mm_stream_ps((float*)&b[j+14],
        _mm_load_ps((float*)&a[j+14]));
    } // 1 ブロックのコピー終了
} // N 要素のコピー終了
_mm_sfence();
```

### 7.6.2.5 TLB の初期化

TLB は高速なメモリーバッファの一種です。TLB を使用して、ページテーブルの各エントリーに高速にアクセスすることで、仮想メモリーアドレスから物理メモリーアドレスへ変換するパフォーマンスが向上します。メモリーページにアクセスしたときに、TLB にそのページ・テーブル・エントリーが存在しない場合、TLB ミスが発生し、さらにそのページテーブルをメモリーから読み出さなければなりません。

ページ変換されたエントリーがプロセッサ・キャッシュにまだ存在しない時に TLB ミスが発生すると、再度メモリーアクセスを行い TLB を更新しなければならないため、パフォーマンスが低下します。目的ページのアドレスにアクセス (またはタッチ) することで、そのページ・テーブル・エントリーをあらかじめ TLB にロードし、次に必要となるページに備えることができます。これはプリフェッチに似ていますが、先行してロードされるのがデータ・キャッシュラインではなくページ・テーブル・エントリーである点が異なります。これによって、ページ・テーブル・エントリーが TLB に常駐すると、その後プリフェッチ命令の要求と同時にプリフェッチが行われることが保証されます。

## 7.6.2.6 8 バイト・ストリーミング・ストアとソフトウェア・プリフェッチの使用

例 7-11 に、2 次キャッシュを考慮したコピー・アルゴリズムを示します。このアルゴリズムは次の順番で実行されます。

1. `_mm_prefetch` 組込み関数によって、ブロッキング手法を用いてメモリーから 2 次キャッシュへ 8 バイト・データを転送します。128 バイト単位で転送して 1 つのブロックを一杯にします。ブロックのサイズは、2 次キャッシュのサイズの半分未満であることが望まれますが、ループ処理による遅延を十分に吸収できる大きさをなければなりません。
2. `_mm_load_ps` 組込み関数を使用して、XMM レジスターにデータをロードします。
3. `_mm_stream` 組込み関数を使用して、別のメモリー・ロケーションにその 8 バイト・データを転送します。このときキャッシュはバイパスされます。この処理では、メモリーにプリフェッチされたページ・テーブル・エントリーが TLB に事前にロードされていることが重要です。

例 7-11 では、プリフェッチされたデータ (128 バイトのキャッシュライン) のすべてがライトバックされるように、`_mm_load_ps` と `_mm_stream_ps` の組込み関数がそれぞれ 8 つずつ使用されています。このプリフェッチ命令とストリーミング・ストア命令は、データの読み出し/書き込み処理間の移行をできるだけ減らすため、別々のループで実行されます。これによりメモリーアクセスの帯域幅が格段に広がります。

プリフェッチの前に配列 A のページ・テーブル・エントリーが TLB に入るようにするには、`temp = a[kk+CACHESIZE]` を実行します。この文によりキャッシュラインがメモリーから取得したデータで満たされるため、本質的にはプリフェッチそのものと言えます。このループでは `kk+4` からプリフェッチが開始されます。

この例では、コピー先がそのコードと時間的に隣接していないことを前提としています。コピーされたデータが、近い将来再利用される場合、これらのストリーミング・ストア命令は、通常の 128 ビット・ストア命令 (`_mm_store_ps`) と置き換える必要があります。これは、インテル® Pentium® 4 プロセッサのストリーミング・ストアの実装は、データがメモリーに直接書き込まれることでキャッシュの一貫性が保たれるためです。

## 7.6.2.7 16 バイト・ストリーミング・ストアとハードウェア・プリフェッチの使用

広いメモリー領域のコピーを最適化するもう 1 つの手法は、ハードウェア・プリフェッチャーと 16 バイト・ストリーミング・ストアの利点を活用するため、バス読み出しと書き込みトランザクションを分離するセグメント化の手法を適用することです。詳細は 3.6.12 節「バス・レイテンシーの最小化」をご覧ください。

この手法は、2 つの段階で構成されます。第 1 段階では、メモリーからキャッシュ・サブシステムにデータブロックが読み込まれます。第 2 段階は、キャッシュされたデータがストリーミング・ストアによってデスティネーションに書き込まれます。



例 7-12 ハードウェア・プリフェッチとバス・セグメンテーションを利用したメモリーコピー

```

void block_prefetch(void *dst,void *src)
{
    _asm {
        mov edi,dst
        mov esi,src
        mov edx,SIZE
        align 16
    main_loop:
        xor ecx,ecx
        align 16

    prefetch_loop:
        movaps xmm0, [esi+ecx]
        movaps xmm0, [esi+ecx+64]
        add ecx,128
        cmp ecx,BLOCK_SIZE
        jne prefetch_loop
        xor ecx,ecx
        align 16
    cpy_loop:
        movdqa xmm0,[esi+ecx]
        movdqa xmm1,[esi+ecx+16]
        movdqa xmm2,[esi+ecx+32]
        movdqa xmm3,[esi+ecx+48]
        movdqa xmm4,[esi+ecx+64]
        movdqa xmm5,[esi+ecx+16+64]
        movdqa xmm6,[esi+ecx+32+64]
        movdqa xmm7,[esi+ecx+48+64]
        movntdq [edi+ecx],xmm0
        movntdq [edi+ecx+16],xmm1
        movntdq [edi+ecx+32],xmm2
        movntdq [edi+ecx+48],xmm3
        movntdq [edi+ecx+64],xmm4
        movntdq [edi+ecx+80],xmm5
        movntdq [edi+ecx+96],xmm6
        movntdq [edi+ecx+112],xmm7
        add ecx,128
        cmp ecx,BLOCK_SIZE
        jne cpy_loop
        add esi,ecx
        add edi,ecx
        sub edx,ecx
        jnz main_loop
        sfence
    }
}

```

### 7.6.2.8 メモリー・コピー・ルーチンのパフォーマンスの比較

広い領域のメモリー・コピー・ルーチンのスループットは、次のような複数の要因に依存します。

- Y メモリー・コピー・タスクを実装するコーディング手法
- Y システムバスの特性 (速度、最大帯域幅、読み出し/書き込みトランザクション・プロトコルのオーバーヘッド)
- Y プロセッサのマイクロアーキテクチャー

表 7-2 では、前述の 2 つのコーディング手法と最適化されていない 2 つの手法を比較しています。

表 7-2 メモリー・コピー・ルーチンのパフォーマンスの比較

プロセッサ、CPUID シグネチャーおよび FSB スピード	(1) バイト・シーケンシャル	(2) ダブルワード・シーケンシャル	(3) SW プリフェッチ + 8 バイト・ストリーミング・ストア	(4) 4KB ブロック HW プリフェッチ + 16 バイト・ストリーミング・ストア
インテル® Pentium® M プロセッサ、0x6Dn、400	1.3X	1.2X	1.6X	2.5X
インテル® Core™ Solo プロセッサおよび インテル® Core™ Duo プロセッサ、0x6En、667	3.3X	3.5X	2.1X	4.7X
インテル® Pentium® D プロセッサ、0xF4n、800	3.4X	3.3X	4.9X	5.7X

パフォーマンスの比較でベースラインとなるのは、例 7-10 と同様のバイト・シーケンシャル手法により、400MHz システムバス対応の第 1 世代インテル® Pentium® M プロセッサ (CPUID シグネチャー 0x69n) 上で 8MB 領域のメモリーコピーを実行した場合のスループット (バイト/秒)です。システムバス速度が速く、コーディング手法が異なるいくつかの最近のプロセッサ/プラットフォームにおける、パフォーマンス・ベースラインに対する向上の度合いを比較しています。

(2) のコーディング手法では、REP 文字列命令を使い 4 バイト単位でデータを移動しています。(3) では、例 7-10 で示したコーディング手法のパフォーマンスを比較しています。(4) では、一度に 4KB のデータをフェッチし (ハードウェア・プリフェッチを使用してバス読み出しトランザクションを収集) 16 バイト・ストリーミング・ストアによってメモリーに書き込んだ際のスループットを比較しています。

スループット向上の主な要因は、バス速度です。例 7-12 で示した手法では、プラットフォームにおけるバス速度向上の利点をより効率良く活用できます。また、ワーキングセット全体を 2 次キャッシュ内に収めつつブロックサイズを 4KB の倍数に増加すると、スループットをわずかに高めることができます。

表 7-2 で示されている相対的なパフォーマンス値は、プロセッサ内のクリーンなマイクロアーキテクチャー条件 (例: 単純なコードシーケンスを何度もループさせる) を表しています。メモリー・コピー・ルーチンをアプリケーションに統合するメリットは、アプリケーションごとに異なります (機能が豊富なアプリケーションでは、マイクロアーキテクチャーごとの複雑な条件が大量に生じる傾向があります)。

### 7.6.3 キャッシュ・パラメーター

CPUID がパラメーター・リーフをサポートしている場合、ソフトウェアは、リーフを利用してキャッシュ階層の各レベルへの照会を行うことができます。各キャッシュレベルの列挙は、ECX レジスターのインデックス値 (0 から開始) を指定することで行われます (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の第 3 章「CPUID-CPU Identification」を参照してください)。

表 7-3 に、パラメーターの一覧を示します。

表 7-3 決定論的キャッシュ・パラメーター・リーフ

ビット位置	名前	意味
EAX[4:0]	キャッシュタイプ	0 = NULL - キャッシュなし 1 = データキャッシュ 2 = 命令キャッシュ 3 = ユニファイド・キャッシュ 4 ~ 31 = 予約済み
EAX[7:5]	キャッシュレベル	1 から開始
EAX[8]	キャッシュレベルの自己初期化	1: ソフトウェアの初期化が不要
EAX[9]	フル・アソシアティブ・キャッシュ	1: はい
EAX[13:10]	予約済み	
EAX[25:14]	このキャッシュを共有する論理プロセッサの最大数	プラス・エンコーディング
EAX[31:26]	パッケージ中のコアの最大数	プラス 1 エンコーディング
EBX[11:0]	システム・コヒーレンシー・ライン・サイズ (L)	プラス 1 エンコーディング (バイト)
EBX[21:12]	物理ライン・パーティション (P)	プラス 1 エンコーディング
EBX[31:22]	アソシアティブ・ウェイ数 (W)	プラス 1 エンコーディング
ECX[31:0]	セット数 (S)	プラス 1 エンコーディング
EDX	予約済み	
3 より大きく 80000000 より小さい CPUID リーフは、IA32_CR_MISC_ENABLES.BOOT_NT4 (ビット 22) がクリアされている場合 (デフォルト) にのみ参照可能です。		

キャッシュ・パラメーター・リーフは、キャッシュ・パラメーターの列挙に関して、ある程度のフォワード互換をソフトウェアに実装するための手段を提供します。キャッシュ・パラメーターは、以下のような状況で役立ちます。

- Y キャッシュレベルのサイズを決定する。
- Y ハイパースレッディング・テクノロジーや、マルチコアおよびシングルコアのプロセッサ間で、異なる共有トポロジーのキャッシュレベルにキャッシュ・ブロック・パラメーターを適応させる。
- Y MP システムのマルチスレッディング・リソース・トポロジーを決定する (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 7 章「Multiple-Processor Management」を参照してください)。
- Y マルチコア・プロセッサを使用したプラットフォームのキャッシュ階層トポロジーを決定する (第 1 章の最後に記載されているトポロジーの列挙に関するホワイトペーパーとサンプルコードを参照してください)。
- Y スレッドとプロセッサのアフィニティーを管理する。
- Y プリフェッチ間隔を決定する。

特定レベルのキャッシュのサイズは、次の式で求められます。

$$(\text{ウェイ数}) * (\text{パーティション}) * (\text{ラインサイズ}) * (\text{セット}) = (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

### 7.6.3.1 キャッシュ・パラメーターを使用したキャッシュ共有

キャッシュの局所性を改善することは、ソフトウェアの最適化における重要な要素です。例えば、キャッシュ・ブロック・アルゴリズムは、シングル・プロセッサおよび各種マルチプロセッサの実行環境 (HT テクノロジー対応プロセッサやマルチプロセッサなど) での実行時にブロックサイズが最適化されるよう設計できます。

基本的な手法は、ターゲット・キャッシュ・レベルによって扱われる論理プロセッサの数でそのターゲット・キャッシュ・レベルのサイズを割ったものより、ブロックサイズの上限を小さくすることです。この手法は、マルチスレッド・アプ

リケーションのプログラミングに適用されます。また、マルチタスク・ワークロードの一部であるシングルスレッド・アプリケーションにもメリットがあります。

### 7.6.3.2 シングルコアまたはマルチコアでのキャッシュ共有

キャッシュ・パラメーターは、より高度な状況においてマルチスレッド・アプリケーションの共有キャッシュ階層を管理する際に効果的です。特定のキャッシュレベルは、単一のプロセッサ内の複数の論理プロセッサで共有できます。または、単一の物理プロセッサ・パッケージ内の複数の論理プロセッサで共有するようにも実装できます。

キャッシュ・パラメーター・リーフと、プラットフォーム内の各論理プロセッサに関連付けられた初期 APIC\_ID を利用すると、ソフトウェアは、キャッシュレベルを共有する論理プロセッサの数およびトポロジーに関する情報を引き出すことができます。

### 7.6.3.3 プリフェッチ間隔の決定

プリフェッチ間隔 (CPUID.01H.EBX の説明を参照) は、プロセッサが PREFETCHH 命令 (PREFETCHT0、PREFETCHT1、PREFETCHT2、PREFETCHNTA) によってプリフェッチを実行する領域の長さを示します。ソフトウェアは、特定のキャッシュ階層レベルに対してプリフェッチを実行する際に、命令によって識別される間隔として、この長さを使用します。プリフェッチ・サイズは、キャッシュタイプのデータキャッシュ (1) とユニファイド・キャッシュ (3) のみに関連しており、その他のキャッシュタイプでは無視すべきです。ソフトウェアは、コヒーレンシー・ライン・サイズがプリフェッチ間隔であると見なしてはなりません。

プリフェッチ間隔フィールドがゼロの場合は、デフォルトサイズの 64 バイトがプリフェッチ間隔であると見なす必要があります。ソフトウェアは、以下のアルゴリズムを使い、キャッシュ・パラメーター機構とレガシー機構のいずれがサポートされているかに応じて、使用すべきプリフェッチ・サイズを決定します。

- Y プロセッサがキャッシュ・パラメーターをサポートし、ゼロ以外のプリフェッチ・サイズを提供している場合は、そのプリフェッチ・サイズを使用します。
- Y プロセッサがキャッシュ・パラメーターをサポートし、プリフェッチ・サイズを提供していない場合、各キャッシュ階層レベルのデフォルトサイズは 64 バイトです。
- Y プロセッサがキャッシュ・パラメーターをサポートせず、レガシーのプリフェッチ・サイズ・ディスクリプター (0xF0 - 64 バイト、0xF1 - 128 バイト) を提供している場合、すべてのキャッシュ階層レベルのプリフェッチ・サイズはこのディスクリプターで示されます。
- Y プロセッサがキャッシュ・パラメーターをサポートせず、レガシーのプリフェッチ・サイズ・ディスクリプターを提供していない場合、すべてのキャッシュ階層レベルのデフォルトサイズは 32 バイトです。

## 第 8 章 サブ NUMA クラスタリングの概要

サブ NUMA クラスタリング (SNC) は、ラスト・レベル・キャッシュ (LLC) からメモリーへの平均レイテンシーを改善するモードです。これは、以前の世代のインテル® Xeon® プロセッサ E5 ファミリーにおけるクラスターオンダイ (COD) の実装を置き換えるものです。

### 8.1 サブ NUMA クラスタリング

SNC は、LLC をアドレス範囲に基づいて複数のクラスターに分割することで LLC/メモリーの平均レイテンシーを改善します。

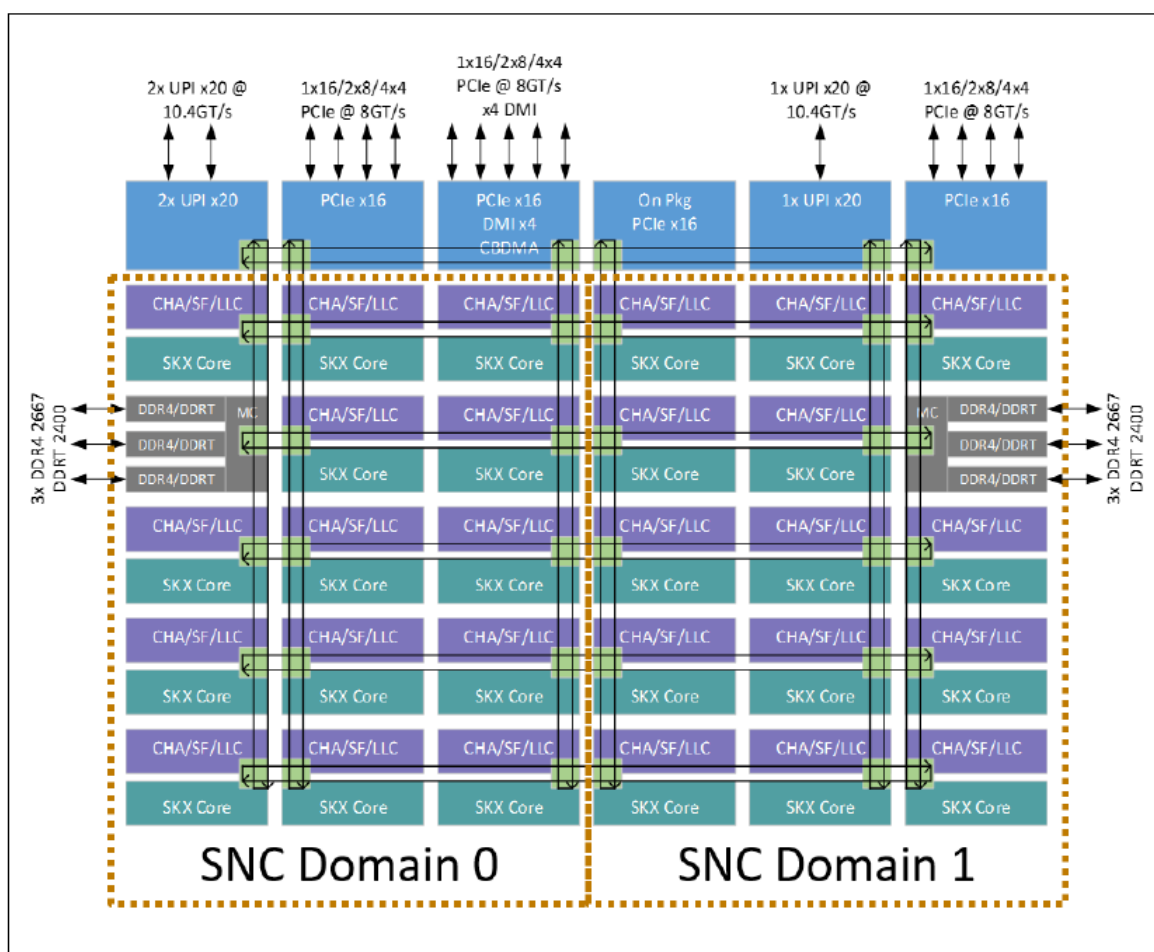


図 8-1 SNC 構成の例

### 8.2 クラスタオンダイとの比較

SNC は COD の不利益を被ることなく、COD と同様の局所的な利点をもたらします。COD とは異なり SNC は次の特性を持ちます。

- Y 単一のウルトラ・パス・インターコネクト (UPI) キャッシュ・エージェントが必要です。
- Y リモートクラスターへのメモリー・アクセス・レイテンシーは小さく、UPI フローは必要ありません。
- Y LLC でラインが複製されることはないため、LLC の容量をより効率良く利用できます。

SNC にも欠点があります。

- Y リモートクラスターのアドレスがローカルクラスターの LCC にキャッシュされることがないため、クラスターオンダイ (COD) 方式に比べレイテンシーが大きくなることがあります。

## 8.3 SNC の利用

この節では次に示すモードと括弧で囲まれたそれらの BIOS 名について説明します (実際の BIOS パラメーター名は BIOS ベンダーとバージョンによって異なります)。

- Y NUMA 無効 (NUMA Optimized: Disabled)
- Y SNC オフ (Integrated Memory Controller (IMC) Interleaving: auto, NUMA Optimized: Enabled, Sub\_NUMA Cluster: Disabled)
- Y SNC オン (IMC Interleaving: 1-way Interleave, NUMA Optimized: Enabled, Sub\_NUMA Cluster: Enabled)

以降に示すコマンドは、2 ソケットのインテル® Xeon® プロセッサベースのシステム (ソケットごとに 28 コア、インテル® ハイパースレッディング・テクノロジー有効) で実行されています。

### 8.3.1 NUMA 構成をチェックする方法

SNC が有効化されるとシステムには追加の NUMA ノードが存在することになります。SNC 機能の利点を活用するには、開発者は NUMA 構成を考慮する必要があります。

この節では、NUMA システムの構成を確認するいくつかの方法を紹介します。

#### libnuma

アプリケーションは、libnuma を使用して NUMA 構成を確認できます。

このコード例では、libnuma を使用して NUMA ノードの最大数を取得しています。

```
#include <stdio.h>
#include <stdlib.h>
#include <numa.h>
int main(int argc, char *argv[])
{
    int max_node;

    /* システムが NUMA をサポートするか確認 */
    max_node = numa_max_node();
    printf("%d\n", max_node);

    return 0;
}
```

#### numactl

Linux\* では、numactl ユーティリティ (numactl-libs と numactl-devel パッケージが必要です) を使用して NUMA 構成を確認できます。

```
$ numactl --hardware
```

```
NUMA disabled:
```

```
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
node 0 size: 196045 MB
node 0 free: 190581 MB
node distances:
node 0
  0: 10
```

```
SNC off:
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83
node 0 size: 96973 MB
node 0 free: 94089 MB
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111
node 1 size: 98304 MB
node 1 free: 95694 MB
node distances:
node 0 1
  0: 10 21
  1: 21 10
```

```
SNC on:
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 7 8 9 14 15 16 17 21 22 23 56 57 58 59 63 64 65 70
71 72 73 77 78 79
node 0 size: 47821 MB
node 0 free: 45759 MB
node 1 cpus: 4 5 6 10 11 12 13 18 19 20 24 25 26 27 60 61 62 66 67 68 69
74 75 76 80 81 82 83
node 1 size: 49152 MB
node 1 free: 47097 MB
node 2 cpus: 28 29 30 31 35 36 37 42 43 44 45 49 50 51 84 85 86 87 91 92
93 98 99 100 101 105 106 107
node 2 size: 49152 MB
node 2 free: 47617 MB
node 3 cpus: 32 33 34 38 39 40 41 46 47 48 52 53 54 55 88 89 90 94 95 96
97 102 103 104 108 109 110 111
node 3 size: 49152 MB
node 3 free: 47231 MB
node distances:
node 0 1 2 3
  0: 10 11 21 21
  1: 11 10 21 21
  2: 21 21 10 11
  3: 21 21 11 10
```

hwloc

また、Linux\* では Istopo ユーティリティ (hwloc パッケージが必要) で NUMA 構成を確認できます。次に例を示します。

```
$ Istopo -p --of png --no-io --no-caches > numa_topology.png
```

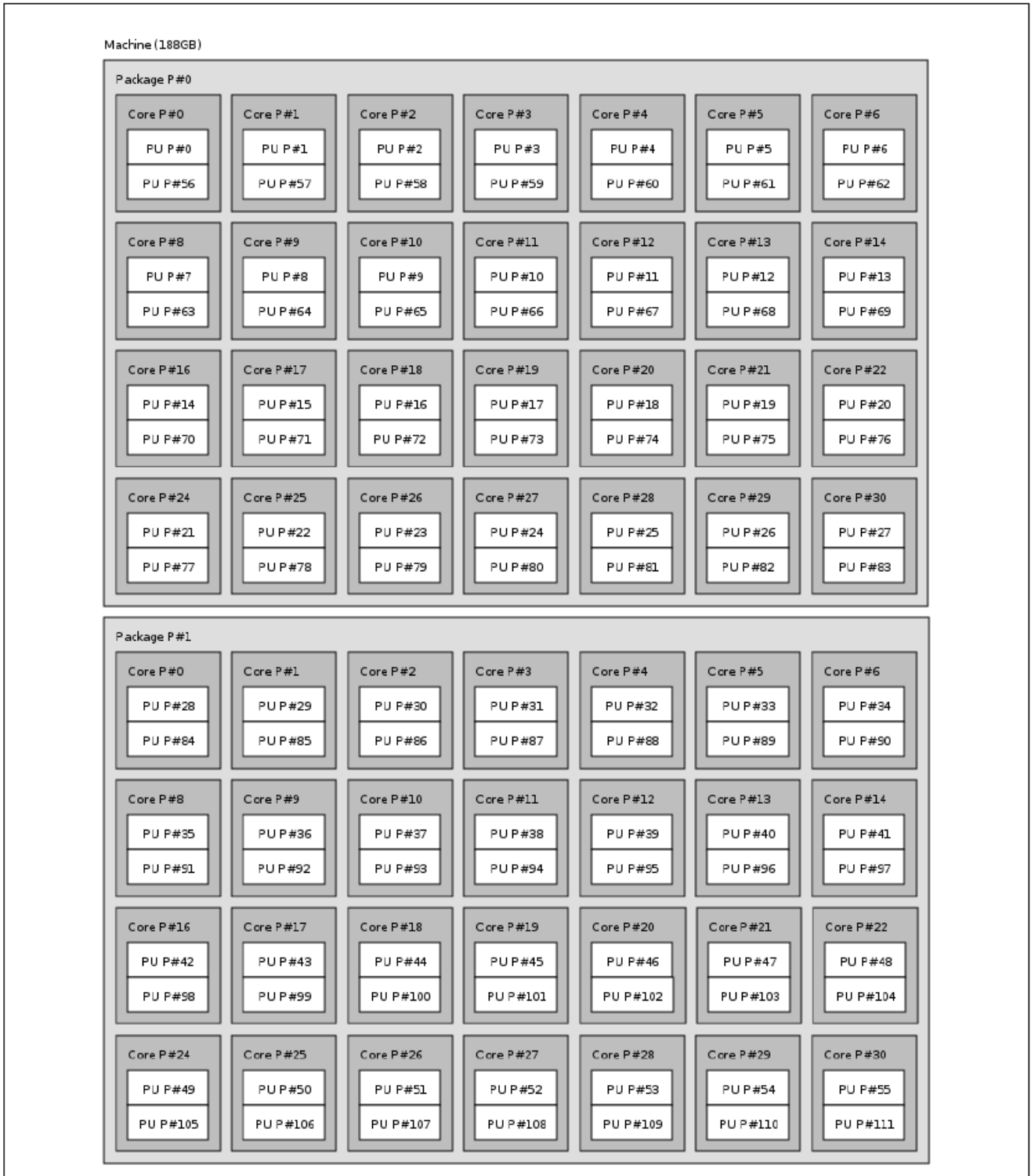


図 8-2 NUMA が無効の場合



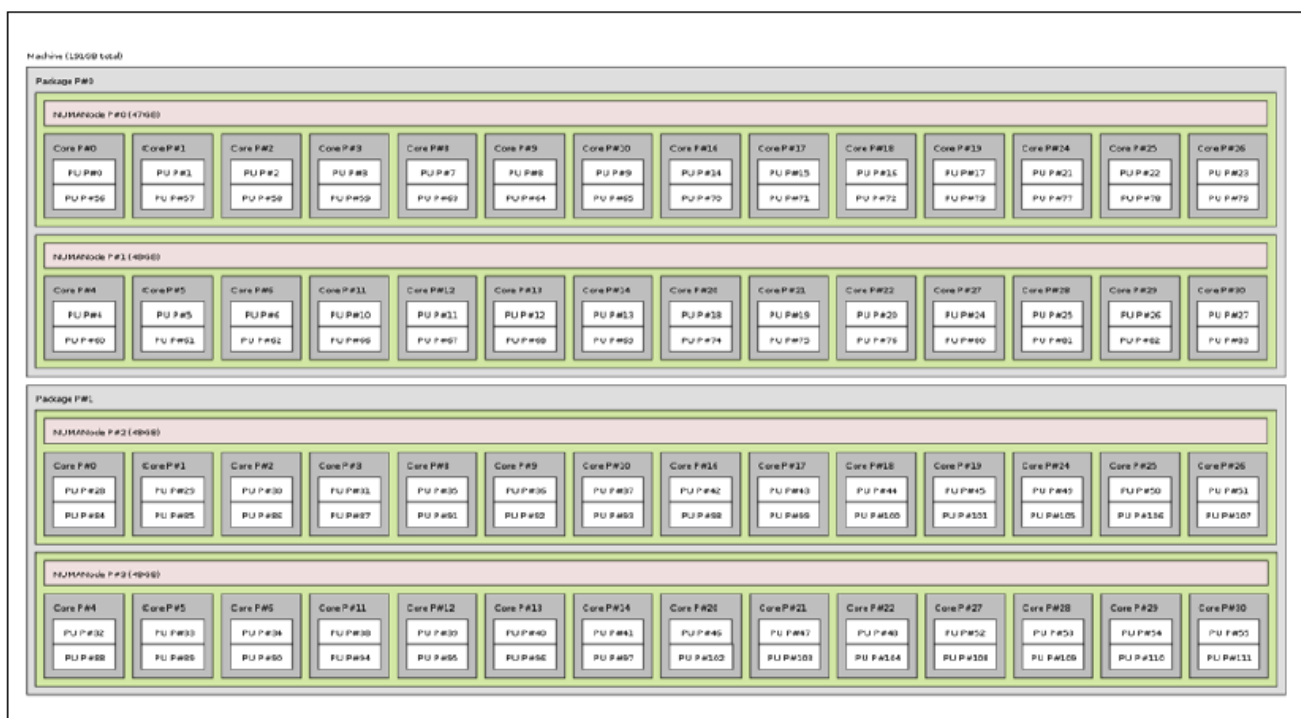


図 8-4 SNC オンの場合

### 8.3.2 SNC 向けに MPI を最適化

ソフトウェアは SNC の恩恵を受けるため NUMA への最適化が必要です。NUMA フレンドリーな動作を保証するようにコードを変更することなく、アクセスの局所性を確実にするため NUMA 領域ごとに 1 つの MPI ランクを実行します。これは SNC を使用してパフォーマンスを向上させる最も簡単な方法です。

インテル® MPI ライブラリーには、いくつかの NUMA に関連する最適化が実装されています。インテル® MPI ライブラリーはアウトオブボックスですぐに動作し、ほとんどの状況に対応できますが、特殊な状況ではパフォーマンスを改善するため環境変数を設定して NUMA 関連の機能を制御する必要があります。

これらの環境変数は主に MPI プロセスの配置 (ピンングやバインド) に関連します。例として、`_MPI_PIN_DOMAIN` 環境変数などがあります。詳細は、インテル® MPI ライブラリーのデベロッパー・リファレンスをご覧ください。この環境変数は、ノード上の論理プロセッサがオーバーラップしないサブセット数 (ドメイン) を定義し、それらのドメインへ MPI プロセスを結びつけるルールを設定することができます。次の図ではドメインごとに 1 つの MPI プロセスを割り当てています。

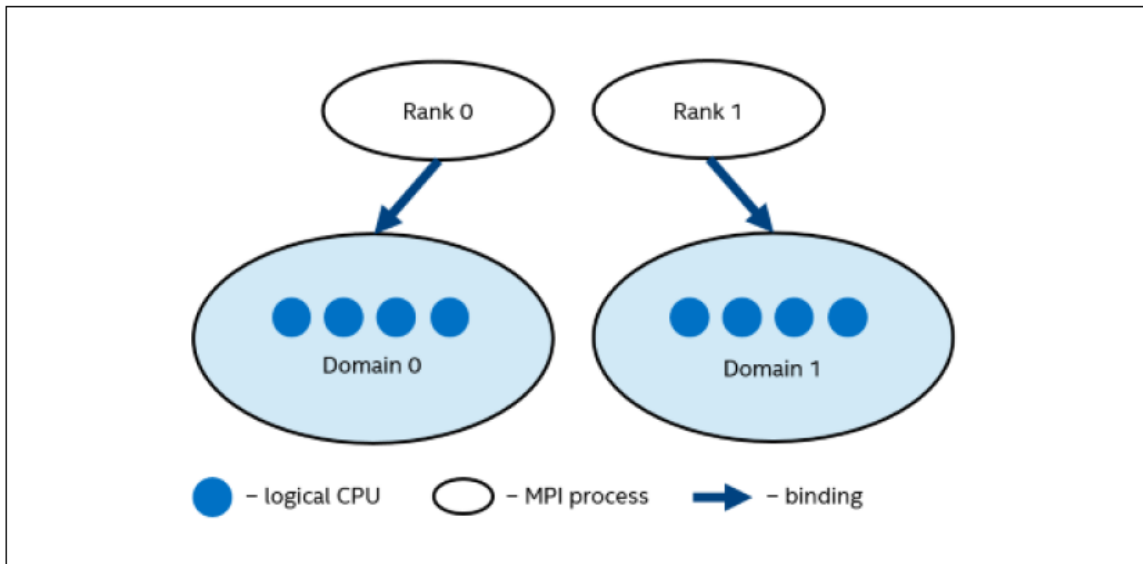


図 8-5 ドメインごとに 1 つの MPI プロセスを割り当てる例

各 MPI プロセスは、対応するドメイン内で実行する子スレッドを作成できます。プロセスのスレッドは、ドメイン内の論理プロセッサからほかの論理プロセッサへ自由に移行できます。

例えば、`I_MPI_PIN_DOMAIN=numa` は、SNC モードが有効化された MPI/OpenMP\* ハイブリッド・アプリケーションで妥当なオプションであると言えます。この場合、各ドメインは特定の NUMA ノードを共有する論理プロセッサで構成されます。マシン上のドメイン数は、マシン上の NUMA ノード数と等しくなります。

詳しく情報については、インテル® MPI ライブラリーのドキュメント、  
<https://software.intel.com/en-us/intel-mpi-library/documentation> (英語) をご覧ください。

### 8.3.3 SNC のパフォーマンス比較

この節では、異なるモードの NUMA ノードのパフォーマンス (レイテンシー) の変化を示すため、インテル® Memory Latency Checker (インテル® MLC) を使用して収集したパフォーマンス・データを掲載しています。

アプリケーションのパフォーマンスを決定する重要な要因は、アプリケーションがプロセッサのキャッシュ階層とメモリー・サブシステムからデータをフェッチするのにかかる時間です。NUMA 構成の複数ソケットシステムでは、ローカルメモリーとソケットを横断するメモリー・レイテンシーはかなり異なります。帯域幅もまた、パフォーマンスを決定する重要な要因です。そのため、検証するシステムでパフォーマンス解析を行う場合、ベースラインを定めてからレイテンシーと帯域幅を測定することが重要です。

インテル® MLC はメモリー・レイテンシーと帯域幅を測定するツールであり、システムの負が高まるにつれてそれらがどのように変化するか観察します。また、特定の組み合わせのコアからキャッシュやメモリーへの帯域幅とレイテンシーを測定できることから、さらに詳しく調査するいくつかのオプションを提供します。

インテル® MLC の詳細については、  
<https://software.intel.com/en-us/articles/intelr-memory-latency-checker> (英語) をご覧ください。

パフォーマンス・データを収集するため次のコマンドを使用しました。

```
% mlc_avx512 --latency_matrix
```

このコマンドは、システムのそれぞれのソケットから別のソケットすべてへの、アイドル・メモリー・レイテンシーを測定し、表形式で結果をレポートします。デフォルトでは、システム中のすべての NUMA ノードへのレイテンシーをレ

ポートします。NUMA レベルのレポートは Linux\* でのみ動作します。Windows\* では、ソケットレベルのレポートがサポートされます。

**注意**

現代のインテル® プロセッサは洗練された HW プリフェッチャーを備えているため、正確にメモリー・レイテンシーを計測するには課題があります。インテル® MLC はレイテンシーを計測する間これらのプリフェッチャーを自動的に無効化し、完了すると元の状態へ戻します。プリフェッチャーの制御は MSR を介して行われ (詳細は、<https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors> (英語) を参照)、MSR へアクセスするには root レベルの権限が必要です。そのため、インテル® MLC は 'root' で実行する必要があります。

この計測に使用したソフトウェアは、インテル® MLC v3.3-Beta2 と Red Hat\* Enterprise Linux\* 7.2 です。

NUMA disabled:

Using buffer size of 2000.000MB

Measuring idle latencies (in ns)...

	Memory node	
Socket	0	1
0	126.5	129.4
1	23.1	122.6

SNC off:

Using buffer size of 2000.000MB

Measuring idle latencies (in ns)...

	Numa node	
Numa node	0	1
0	81.9	153.1
1	153.7	82.0

SNC on:

Using buffer size of 2000.000MB

Measuring idle latencies (in ns)...

	Numa node			
Numa node	0	1	2	3
0	81.6	89.4	140.4	153.6
1	86.5	78.5	144.3	162.8
2	142.3	153.0	81.6	89.3
3	144.5	162.8	85.5	77.4

## 第 9 章

### マルチコアとハイパースレッディング・テクノロジー

この章では、マルチプロセッサ (MP) システムまたはハードウェアベースのマルチスレッディング・サポートを搭載したプロセッサで動作するマルチスレッド・アプリケーションのソフトウェア最適化手法について説明します。マルチプロセッサ・システムとは、複数のソケットを備えたシステムであり、各ソケットは 1 つの物理プロセッサ・パッケージに対応しています。ハードウェア・マルチスレッディング・サポートを提供するインテル® 64 プロセッサと IA-32 プロセッサには、デュアルコア・プロセッサ、クアッドコア・プロセッサ、HT テクノロジー<sup>5</sup> 対応プロセッサなどがあります。

ハードウェア・リソースを増やしてスレッドレベルまたはタスクレベルの並列化を活用すると、マルチスレッディング環境での計算スループットが向上します。ハードウェア・リソースの増加は、物理プロセッサ、パッケージごとのプロセッサ・コア、コアごとの論理プロセッサのいずれか、またはすべてを複数搭載することによって可能となります。したがって、マルチスレッディングの最適化の一部は、MP、マルチコア、HT テクノロジーのすべてに適用されます。また、一部のマイクロアーキテクチャー・リソースは、ハードウェア・マルチスレッディングの構成によって実装方法が異なります (例えば、HT テクノロジーが有効な場合、実行リソースは、異なるコア間では共有されませんが、同じコア内の 2 つの論理プロセッサ間では共有されます)。この章では、これらの状況に適用されるガイドラインについて説明します。

次の内容が含まれます。

- Y 性能の特性と使用モデル
- Y マルチスレッド・アプリケーション向けのプログラミング・モデル
- Y 5 つの特定分野におけるソフトウェア最適化手法

## 9.1 性能および使用モデル

マルチプロセッサ、マルチコア・プロセッサ、HT テクノロジーを使用したときの性能向上は、使用モデル、ワークロードの制御フローにおける並列処理の割合に大きく左右されます。一般的な使用モデルは次の 2 つです。

- Y マルチスレッド・アプリケーション
- Y シングルスレッド・アプリケーションを使用するマルチタスキング

### 9.1.1 マルチスレッディング

アプリケーションでマルチスレッディング・モデルを採用し、ワークロードに対してタスクレベルの並列処理を実装する場合、マルチスレッド・ソフトウェアの制御フローは、並列タスクと順次タスクの 2 つの領域に分割できます。

アムダールの法則は、制御フロー内の並列処理の割合とアプリケーションの性能向上の関係を説明しています。この法則は、並列化するコードモジュール、関数、または命令シーケンスを選択する際に有用な指針となります。順次タスクと制御フローを並列コードに変換して、マルチスレッディング・ハードウェア・サポートを活用して得られる性能向上は、これらの要素により最大化できる可能性が高まります。

<sup>5</sup> インテル® 64 プロセッサと IA-32 プロセッサがハードウェア・マルチスレッディングをサポートしているかどうかは、CPUID の機能フラグ CPUID.01H:EDX[28] によって判別できます。ビット 28 の戻り値が 1 である場合は、少なくとも 1 種類のハードウェア・マルチスレッディングが物理プロセッサ・パッケージに搭載されていることを示します。各パッケージでサポートされている論理プロセッサの数も CPUID から判断できます。また、アプリケーションは、適切なオペレーティング・システム呼び出しを行うことにより、実行時にアプリケーションが利用できる有効な論理プロセッサの数を確認する必要があります。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』をご覧ください。

図 9-1 は、アムダールの法則により、ワークロードに対する性能向上が実現される様子を示しています。図 9-1 のボックスは、個々のタスクユニット、またはアプリケーション全体のワークロードの集合を示しています。

一般に、N 個の物理プロセッサ（もしくはコア）を持つ MP システム上でマルチスレッドを実行すると、シングルスレッドで実行する場合と比較した速度の向上は次の式で表すことができます。

$$\text{RelativeResponse} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} = \left( 1 - P + \frac{P}{N} + O \right)$$

ここで P は並列処理可能なワークロードの比率、O はマルチスレッディングのオーバーヘッド（オペレーティング・システムにより異なる）を示します。この場合、性能向上は相対レスポンスの逆数で示されます。

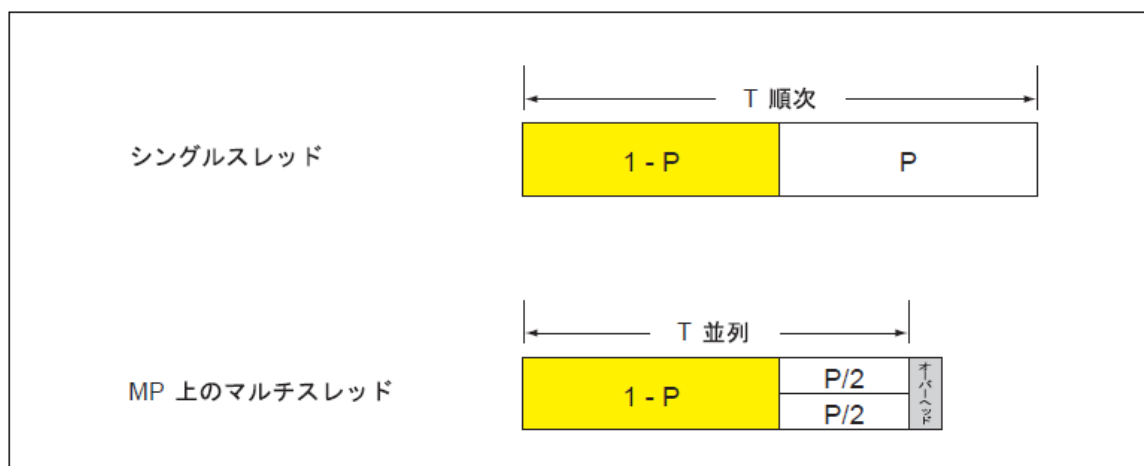


図 9-1 アムダールの法則と MP のスピードアップ

一般に、マルチスレッド環境でアプリケーションの性能を最適化する場合、物理プロセッサ数、および物理プロセッサあたりの論理プロセッサ数に関連する性能スケーリングに最も影響するのは、全体に占める並列処理の割合です。

並列実行可能な領域が 50% しかないワークロードがマルチスレッド・アプリケーションの制御フローに含まれている場合、2 つの物理プロセッサを使用したときの性能は、1 つのプロセッサを使用したときと比べて、最大 33% しか向上しません。4 つのプロセッサを使用した場合でも、60% しかスピードアップしません。したがって、並列処理を利用できる制御フローの割合を最大化することが重要です。スレッド同期の実装が不適切であると、順次制御フローの割合が大幅に増加し、アプリケーションの性能スケーリングがさらに減少します。

制御フローの並列処理を最大化することに加え、スレッド同期とタスク・スケジューリングのインバランスによるスレッド間の相互作用は、プロセッサのスケーリング全体に大きな影響を与えます。

過度なキャッシュミスは、性能スケーリングが低下する原因の 1 つです。マルチスレッド実行環境では、過度なキャッシュミスは以下が原因で発生します。

- Y 同じプロセス内の異なるスレッドによる別名スタックアクセス
- Y スレッド競合によるキャッシュラインの排出
- Y 異なるプロセッサ間でのキャッシュラインのフォルス・シェアリング

こうした状況（およびその他の問題）に対処する手法は、以降の節で説明します。

## 9.1.2 マルチタスキング環境

インテル® 64 と IA-32 プロセッサのハードウェア・マルチスレッディング機能がタスクレベルの並列処理を活用できるのは、ワークロードがいくつかのシングルスレッド・アプリケーションから構成され、それらのアプリケーションが MP 対応オペレーティング・システムの下で並列実行されるようにスケジュールされる場合です。こうした環境では、ハードウェア・マルチスレッディング機能はワークロードに対して高いスループットを提供できます。ただし、シングルタスクの相対的なパフォーマンス（シングルスレッド環境における同一タスクと比較した完了時間）は、共有の実行リソースおよびメモリーがどの程度利用されているかに応じて異なります。

各物理プロセッサ内でタスク・スケジューリングや共有実行リソースのバランスを管理し、スループットを最大限に高めることができる OS カーネルコードは、一般的なオペレーティング・システム (Microsoft® Windows® XP Professional および Home、カーネル 2.4.19 以降<sup>6</sup> の Linux® ディストリビューションなど) に含まれています。

マルチタスキング環境では、アプリケーションは独立して実行されるため、スレッド同期の問題によってスループットのスケールが制限される可能性は低くなります。これは、(プロセッサ間通信が発生せず、システムバスの制約がない限り) ワークロードの制御フローは一般に 100% 並列となるためです。<sup>7</sup>

マルチタスキング・ワークロードでは、バス・アクティビティーやキャッシュのアクセスパターンによって、スループットのスケールが影響を受けやすくなります。同じアプリケーションを 2 つ、または同じアプリケーション・スイートをロックステップで実行すると、パフォーマンス測定に影響が現れることがあります。これは、1 次データキャッシュへのアクセスパターンが過度なキャッシュミスを引き起こし、パフォーマンスに影響する可能性があるためです。この問題は次の方法で解決できます。

- Y アプリケーションの起動時にインスタンス 1 つあたりのオフセットを含めます。
- Y アプリケーションの各インスタンスに対して異なるデータセットを使用し、異なるワークロードを与えます。
- Y 複数のコピーを実行する場合、アプリケーションの起動順序をランダムにします。

マルチタスキング・ワークロードの一部として 2 つのアプリケーションを使用すると、これらの 2 つのプロセス間では同期オーバーヘッドがほとんど発生しません。また、各アプリケーション自体の同期オーバーヘッドを最小にすることも重要です。

プロセス内部の同期を目的とした長いスピループを使用するアプリケーションでは、マルチタスキング・ワークロードで HT テクノロジーの効果が得られる可能性は低くなります。これは、長いスピループによって重要なリソースが消費されるためです。

## 9.2 プログラミング・モデルとマルチスレッディング

並列処理は、マルチスレッド・アプリケーションを設計し、マルチプロセッサで最高のパフォーマンス・スケールを達成する最も重要な概念です。最適化されたマルチスレッド・アプリケーションは、並列処理の割合が高く、以下の点で依存性が低い特長があります。

- Y ワークロード
- Y スレッド間のやり取り
- Y ハードウェア利用効率

ワークロードの並列処理を最大限にする鍵は、アプリケーション内にある相互依存性の低い複数のタスクを識別し、それらのタスクを並列実行するため独立したスレッドを生成することです。

<sup>6</sup> このコードは、Red Hat® Linux Enterprise AS 2.1 に含まれています。

<sup>7</sup> 一般に、マルチタスキング・ワークロードのスループットを評価するソフトウェア・ツールを使用すると、シリアル制御フローが増加します。スレッド同期の問題は、パフォーマンス測定手法における不可欠な要素として検討する必要があります。

独立したスレッドを並列に実行するのは、マルチスレッド・アプリケーションをマルチプロセッシング・システムに導入する際の最も重要な点です。スレッド間のやり取りを管理して、スレッド同期のコストを最小限に抑えるのも、マルチプロセッサでスケール性を高めるために重要です。

並列スレッド間でハードウェア・リソースを効率良く使用するには、ハードウェア・リソースの競合を回避する最適化手法が必要となります。スレッド同期を最適化し、他のハードウェア・リソースを管理するコーディング手法については、以降の節で説明します。

並列プログラミング・モデルは、その後で説明します。

## 9.2.1 並列プログラミング・モデル

次に、独立したタスクの要件をアプリケーション・スレッドに置き換える一般的なプログラミング・モデルを示します。

- Y ドメイン分解
- Y 機能分解

### 9.2.1.1 ドメイン分解

通常、計算主体の大きなタスクでは、多数の小さなサブセットに分割可能なデータセットが使用されます。これらの各サブセットは、計算の独立性が高い傾向があります。これには次のような例が含まれます。

- Y 2次元データに対する離散コサイン変換 (DCT) の計算: 2次元データをいくつかのサブセットに分割し、各サブセットの変換を計算するスレッドを生成します。
- Y 行列乗算: 乗数行列を使用して、行列の半分の乗算を処理する複数のスレッドを生成します。

ドメイン分解は、同じまたは同等のスレッドを生成し、より小さなデータ部分を独立して処理することをベースにしたプログラミング・モデルです。このモデルでは、従来のマルチプロセッサ・システム内にある複製された実行リソースを活用できます。また、HT テクノロジーの 2つの論理プロセッサ間で共有される実行リソースも活用できます。これは、一般にデータ・ドメイン・スレッドでは、オンチップの実行リソースの使用可能な部分のみが消費されるためです。

9.3.4 節「実行リソース最適化の主な慣例」では、データ・ドメイン・スレッドで共有実行リソースを使用し、2つのスレッド間のハードウェア・リソースの競合を排除するガイドラインを詳しく説明しています。

### 9.2.2 機能分解

通常、アプリケーションは、多様な機能と多くの依存性のないデータセットを使用して、幅広いタスクを処理します。例えば、ビデオコーデックでは、DCT、動き評価、カラー変換など各種処理機能が必要となります。アプリケーションは、機能スレッドモデルを使用して、動き評価、カラー変換、その他の機能タスクを実行する独立したスレッドをプログラムできます。

機能分解は、ハードウェア・リソースの複製にあまり依存しない場合、より柔軟なスレッドレベルの並列処理を実現できます。例えば、ソート・アルゴリズムを実行するスレッドと、行列乗算ルーチンを実行するスレッドは、同じ実行ユニットを同時に使用しません。これを考慮して設計すると、従来のマルチプロセッサ・システムや、HT テクノロジー対応のプロセッサを使用するマルチプロセッサ・システムを活用できます。

## 9.2.3 専用プログラミング・モデル

インテル® Core™ Duo プロセッサとインテル® Core™ マイクロアーキテクチャー・ベースのプロセッサでは、同一物理パッケージ内の 2 つのプロセッサ・コアが 2 次キャッシュを共有します。そのため、バス・トラフィックのオーバーヘッドを最小限に抑えて 2 つのアプリケーション・スレッドが同じアプリケーション・データにアクセスできます。

マルチスレッド・アプリケーションでこの種のハードウェア機能を利用するには、専用のプログラミング・モデルが必要な場合があります。このようなモデルの 1 つに、「生産 (producer) - 消費 (consumer)」モデルがあります。このモデルでは、一方のスレッドがデータをデスティネーション (2 次キャッシュが望ましい) に書き込み、同一物理パッケージ内の別のコアで実行される他方のスレッドが、最初のスレッドによって生成されたデータを読み出します。

生産-消費モデルを実装する基本的なアプローチは、一方が生産し他方が消費する 2 つのスレッドを生成することです。生産と消費スレッドは通常、バッファを介して交互に処理を行い、バッファの交替準備が整うと互いに通知します。生産-消費モデルでは、両方でバッファを交替する際にスレッド同期のオーバーヘッドが生じます。コア数に応じて最適なスケーリング保つには、同期のオーバーヘッドを低く抑える必要があります。これは、各増分タスクを終了してバッファを交替する前に、生産と消費のスレッドで同等の時定数を設定することにより実現できます。

例 9-1 に、タスクユニットのシーケンスをシングルスレッドで実行する場合のコーディング構造を示します。この例では、各タスクユニット (生産または消費) がシリアルに実行されています (図 9-2 を参照)。同等のシナリオをマルチスレッドで実行する場合、生産と消費の 1 つのペアを 1 つのスレッド関数としてラップすると、プロセッサ・リソースに 2 つのスレッドを同時にスケジューリングできます。

例 9-1 生産および消費のワークアイテムのシリアル実行

```
for (i = 0; i < number_of_iterations; i++) {
    producer (i, buff); // バッファ・インデックスとアドレスを渡す
    consumer (i, buff);
}
```

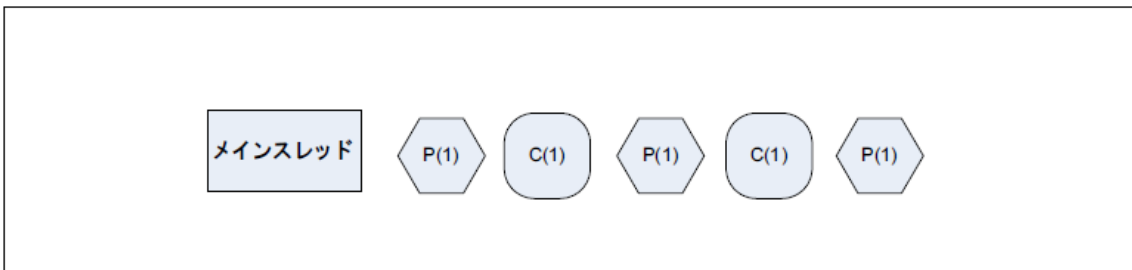


図 9-2 生産-消費スレッド化モデルをシングルスレッドで実行

### 9.2.3.1 生産-消費スレッド化モデル

図 9-3 に、生産スレッドと消費スレッドのペア間の相互作用の基本的な仕組みを示します。水平方向は時間を表します。各ブロックは、タスクユニットを表し、スレッドに割り当てられたバッファを処理します。

各タスク間の隙間は、同期のオーバーヘッドを表します。括弧内の 10 進数は、バッファ・インデックスです。インテル® Core™ Duo プロセッサの場合、生産スレッドがデータを 2 次キャッシュに格納できるため、消費スレッドは最小限のバス・トラフィックで処理を継続できます。



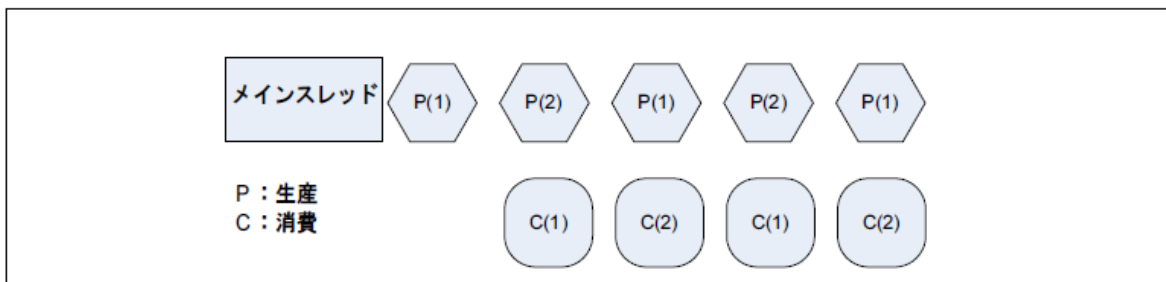


図 9-3 マルチコア・プロセッサ上で生産-消費スレッド化モデルを実行

例 9-2 に、同期によってバッファ・インデックスを通知する生産と消費のスレッド関数を実装する基本構造を示します。

例 9-2 生産および消費スレッドを実装する基本構造

```

(a) 生産スレッド関数の基本構造
void producer_thread()
{ int iter_num = workamount - 1; // ローカルコピーを作成
  int mode1 = 1; // 0 と 1 で 2 つのバッファの利用を監視
  produce(bufs[0],count); // プレースホルダー関数
  while (iter_num-->0) {
    Signal(&signal1,1); // 開始する他のスレッドへ通知
    produce(bufs[mode1],count); // プレースホルダー関数
    WaitForSignal(&end1);
    mode1 = 1 - mode1; // 他のバッファへ切り替え
  }
}

(b) 消費スレッドの基本構造
void consumer_thread()
{ int mode2 = 0; // 最初の反復はバッファ 0 で開始
  int iter_num = workamount - 1;
  while (iter_num-->0) {
    WaitForSignal(&signal1);
    consume(bufs[mode2],count); // プレースホルダー関数
    Signal(&end1,1);
    mode2 = 1 - mode2;
  }
  consume(bufs[mode2],count);
}
    
```

バス・トラフィックを最小限に抑え、共有 2 次キャッシュのないマルチコア・プロセッサでも効果を得られるように、生産と消費モデルをインターレース方式で構造化できます。

このインターレース方式の生産・消費モデルでは、アプリケーション・スレッドの各スケジューリング単位は、生産タスクと消費タスクで構成されます。同じスレッドが 2 つ生成され、並列に実行されます。各スレッドのスケジューリングでは、まず生産タスクが開始され、生産タスクの完了後に消費タスクが開始されます。いずれのタスクも、同じバッファを処理します。タスクの完了ごとに、一方のスレッドが他方のスレッドに信号を送り、特定のバッファを使用するように対応するタスクに通知します。生産タスクと消費タスクはこのようにして、2 つのスレッドで並列実行されます。生産タスクによって生成されるデータが、同一コアの 1 次キャッシュまたは 2 次キャッシュ内に存在する限り、消費タスクはバス・トラフィックを発生させずにそのデータにアクセスできます。図 9-4 に、インターレース方式の生産・消費モデルのスケジューリングを示します。

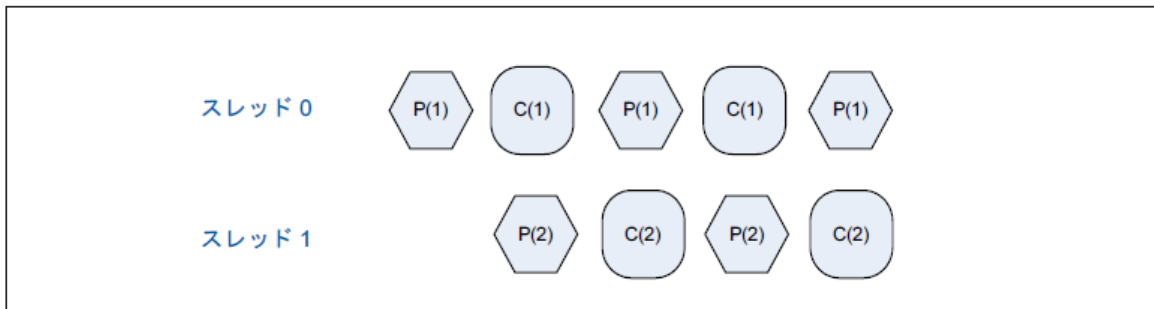


図 9-4 インターレース方式の生産-消費モデル

例 9-3 に、インターレース方式の生産-消費モデルで使用可能なスレッド関数の基本構造を示します。

例 9-3 インターレース方式の生産-消費モデルのスレッド関数

```
// マスタースレッドは最初の反復を開始
// 他のスレッドは1 つの反復を待機
void producer_consumer_thread(int master)
{
    int mode = 1 - master; // スレッドとバッファ・インデックスを追跡
    unsigned int iter_num = workamount >> 1;
    unsigned int i=0;
    iter_num += master & workamount & 1;
    if (master) // マスタースレッドは最初の反復を開始
    {
        produce(bufs[mode],count);
        Signal(sigp[1-mode],1); // 生産タスクは後続のスレッドが開始できることを通知
        consume(bufs[mode],count);
        Signal(sigc[1-mode],1);
        i = 1;
    }
    for (; i < iter_num; i++)
    {
        WaitForSignal(sigp[mode]);
        produce(bufs[mode],count); // 生産タスクは他のスレッドに通知
        Signal(sigp[1-mode],1);
        WaitForSignal(sigc[mode]);
        consume(bufs[mode],count);
        Signal(sigc[1-mode],1);
    }
}
```

## 9.2.4 マルチスレッド・アプリケーション作成用のツール

マルチスレッド用のアプリケーション・プログラミング・インターフェイス (API) を直接使用してプログラミングすることだけがマルチスレッド・アプリケーションを作成する方法ではありません。マルチスレッド・アプリケーションを簡単に作成する機能を備えた (インテル® コンパイラーなどの) 新しいツールが提供されています。

最新のインテル® コンパイラーでは、次の機能が提供されています。

- Y OpenMP\* ディレクティブによるマルチスレッド・コードの生成<sup>8</sup>
- Y ソースコードを変更することなくマルチスレッド・コードを自動生成<sup>9</sup>

<sup>8</sup> OpenMP\* ディレクティブをサポートするのは、インテル® コンパイラー 5.0 以降です。詳細は、<http://software.intel.com> (英語) を参照してください。

<sup>9</sup> 自動並列化をサポートするのは、インテル® コンパイラー 6.0 以降です。

### 9.2.4.1 OpenMP\* ディレクティブによるプログラミング

OpenMP\* は、アプリケーションの共有メモリー並列処理をサポートする Fortran と C/C++ 向けのコンパイラー・ディレクティブを提供します。これらのディレクティブは標準化されており、独自仕様ではなく、移植性があります。OpenMP\*は、ディレクティブ・ベースの処理をサポートします。ディレクティブ・ベースの処理では、特別なプリプロセッサまたはディレクティブの解釈を実装したコンパイラーを使用して、Fortran コメント、または C/C++ プラグマで表現される並列処理が解釈されます。ディレクティブ・ベースの利点を次に示します。

- Y 元のソースを変更せずにコンパイルできます。
- Y コードを段階的に変更することが可能です。これにより、元のコードのアルゴリズムを保持でき、迅速なデバッグが可能となります。
- Y 段階的なコードの変更は、シリアルバージョンの一貫性の維持に役立ちます。あるプロセッサ上でコードを実行すると、未変更のソースコードと同じ結果が得られます。
- Y スレッド・スケジューリングのインバランスを微調整するディレクティブを提供します。
- Y インテルの OpenMP\* ランタイムは、手作業でコーディングされたマルチスレッディングのコードと比較して、スレッディングのオーバーヘッドを最小化できます。

### 9.2.4.2 コードの自動並列化

OpenMP\* ディレクティブでは、シリアル・アプリケーションを迅速に並列アプリケーションに変換できます。ただし、アプリケーション・コード内の並列処理が可能な領域を識別し、コンパイラー・ディレクティブを追加する必要があります。インテル® コンパイラー 6.0 以降では、新しいオプションとして /Qparallel (Windows\*), -parallel (Linux\* および macOS\*) をサポートしています。このオプションでは、並列処理が含まれているループ構造を識別します。コンパイル中、コンパイラーは、並列処理のためにコードシーケンスを別々のスレッドに自動的に分割しようと試みます。プログラマーは、オプションを指定する以外の作業は必要ありません。

### 9.2.4.3 開発ツールのサポート

ソフトウェア開発向けにインテルから提供される各種ツールの詳細については、付録 A の「アプリケーション・パフォーマンス・ツール」をご覧ください。

## 9.3 最適化のガイドライン

この節では、マルチスレッド・アプリケーションをチューニングする最適化のガイドラインについて説明します。重要度の高い順で、次の 5 つの分野を取り上げます。

- Y スレッド間の同期
- Y バスの利用率
- Y メモリー最適化
- Y フロントエンドの最適化
- Y 実行リソースの最適化

ここでは、それぞれの分野に関連する慣例を説明しています。各分野のガイドラインについては、以降の節で詳しく説明します。

コーディングの推奨事項の大部分は、プロセッサ・コアによる性能スケーリングや HT テクノロジーによるスケーリングを改善します。どちらか一方にしか適用されない手法については、その都度注記します。

### 9.3.1 スレッド間の同期の主な慣例

スレッド同期のコストを最小限に抑えるための主な慣例を次に示します。

- Y 高速スピンドループ中に PAUSE 命令を挿入し、ループの反復回数を最小限に抑え、システム全体の性能を向上させます。
- Y 複数のスレッドが取得可能なスピンドルロックを、2 つのスレッドしか 1 つのロックに書き込めないようなパイプライン化されたロックと置き換えます。2 つのスレッドが共有する変数に、1 つのスレッドしか書き込みを行う必要がない場合は、ロックを取得する必要はありません。
- Y スレッド・ブロッキング API を長いアイドルループ内で使用し、プロセッサを解放します。
- Y 2 つのスレッド間で、スレッドごとのデータの「フォルス・シェアリング」を防止します。
- Y 各同期変数を 128 バイトで分離して単独に配置するか、独立したキャッシュラインに格納します。

詳細は、9.4 節「スレッド間の同期」を参照してください。

### 9.3.2 システムバス最適化の主な慣例

バス・トラフィックを管理すると、マルチスレッド・ソフトウェアおよび MP システム全体の性能で大きな効果が得られます。次に、高速なデータ・スループットや応答の実現に向けたシステムバス最適化の主な慣例を示します。

- Y データおよびコードの局所性を改善し、バスコマンド帯域幅を保持します。
- Y ソフトウェア・プリフェッチ命令の過度な使用は避け、自動ハードウェア・プリフェッチを機能させます。ソフトウェア・プリフェッチを過度に使用すると、バス利用が大幅かつ不必要に増加する可能性があります (不適切に使用されている場合)。
- Y オーバーラップする複数の連続したメモリー読み込みにより、実効キャッシュ・ミス・レイテンシーを改善することを検討します。
- Y フルサイズの書き込みトランザクションを使用して、より高いデータ・スループットを達成します。

詳細は、9.5 節「システムバスの最適化」を参照してください。

### 9.3.3 メモリー最適化の主な慣例

次に、メモリー操作を最適化する主な慣例を示します。

- Y キャッシュ・ブロッキングを使用して、データアクセスの局所性を改善します。HT テクノロジー対応のプロセッサが対象の場合は、キャッシュサイズの 4 分の 1 から 2 分の 1 を目標とします。
- Y バスを共有する別々の物理プロセッサ上で実行されるスレッド間で、データの共有を最小限に抑えます。
- Y 各スレッドにおいて 64KB の倍数でオフセットされるデータ・アクセス・パターンを抑えます。
- Y HT テクノロジー対応のプロセッサが対象の場合は、アプリケーション内の各スレッドのプライベート・スタックを調整し、それらのスタック間の挿入間隔が 64KB または 1MB の倍数でオフセットされないようにします (キャッシュラインの不必要な排出を避けるため)。
- Y HT テクノロジー対応のプロセッサが対象の場合は、64KB または 1MB の倍数でオフセットされたメモリーアクセスを避けるため、同じアプリケーションの 2 つのインスタンスがロックステップで実行しているときに、インスタンス 1 つあたりのスタックオフセットを追加します。

詳細は、9.6 節「メモリーの最適化」を参照してください。

### 9.3.4 実行リソース最適化の主な慣例

各物理プロセッサは、専用の実行リソースを持っています。また、HT テクノロジーに対応した物理プロセッサ内の論理プロセッサは、オンチップの特定の実行リソースを共有します。次に、実行リソース最適化の主な慣例を示します。

- Y 各スレッドを最適化し、最初に最適な周波数スケーリングを達成します。
- Y マルチスレッド・アプリケーションを最適化し、物理プロセッサ数に対する最適なスケーリングを達成します。
- Y 同一の物理プロセッサ・パッケージ内で 2 つのスレッドが実行リソースを共有している場合、オンチップの実行リソースを共有します。
- Y 各物理プロセッサ・パッケージ内でハードウェア・リソースの使用率を高めるため、HT テクノロジー対応のプロセッサごとに、機能的に相関のないスレッドを追加することを検討します。

9.8 節「アフィニティーと共有プラットフォーム・リソースの管理」を参照してください。

### 9.3.5 一般性およびパフォーマンスの影響

次の 5 つの節では、各最適化手法について詳しく説明します。各節で説明する推奨事項は、局所的な影響および一般性の評価に関して、重要性の度合いが示されます。

優先度は、主観的で大まかなものです。これは、コーディングの形式やアプリケーション/スレッドのドメインによって変化します。高(H)、中(M)、低(L) という優先度を各推奨事項に含める目的は、推奨事項を適用したときに期待されるパフォーマンス向上の程度を相対的に評価するためです。

多岐にわたるアプリケーションのコード・インスタンスの頻度を予測できないため、影響する優先度をアプリケーション・レベルでパフォーマンス向上に直接相互に関連付けられません。一般性に対する優先度も主観的で大まかなものです。

これらのどのスケール要素にも影響を与えないコーディングの推奨事項は、一般に中 (M) または低 (L) として分類されます。

## 9.4 スレッド間の同期

複数のスレッドを持つアプリケーションは、動作を正しく行うために、同期を使用します。しかし、スレッド同期の実装が不適切であると、性能が大幅に低下する可能性があります。

スレッド同期のオーバーヘッドを削減する最適な慣例としては、アプリケーションの同期を減らすことから始めます。インテル® パフォーマンス・ツールを利用すると、各スレッドの実行タイムラインをプロファイルした上で、頻繁な同期のオーバーヘッドの発生によって性能が影響を受けている箇所を検出できます。

スレッド同期で頻繁に使用されるコーディング手法およびオペレーティング・システム (OS) 呼び出しには、スピンウェイト・ループ、スピンロック、クリティカル・セクションなどがあります。状況に応じて最適な OS 呼び出しを選択し、並列処理を考慮して同期コードを実装することは、スレッド同期の処理コストを最小限に抑えるため重要です。

インテル® SSE3 では、複数のエージェント間でのマルチスレッド・ソフトウェアの同期を改善するように、2 つの命令 (MONITOR と MWAIT) を提供しています。MONITOR と MWAIT の最初の実装では、これらの命令がオペレーティング・システムで利用可能であるため、オペレーティング・システムはさまざまな領域でスレッド同期を最適化できます。例えば、オペレーティング・システムがシステム・アイドル・ループ (C0 ループ) で MONITOR と MWAIT を使用すると、消費電力を削減できます。また、C1 ループで MONITOR と MWAIT を使用すると、C1 ループの応答性を高めることができます。詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 8 章を参照してください。

### 9.4.1 同期プリミティブの選択

スレッド間の同期では、同期プリミティブによって操作を保護しながら共有データを更新する手法が多く用いられます。利用可能なプリミティブは多数あります。同期プリミティブの選択に役立つガイドラインを紹介します。

Y インクリメントや比較/交換など単純なデータ操作のアトミックな更新では、コンパイラー組み関数や OS が提供するインターロック API を優先します。これらは、オーバーヘッドが多く複雑な同期プリミティブよりも効率的です。

各種同期プリミティブの使用に関する詳細については、ホワイトペーパー「マルチスレッド・アプリケーションの開発のためのガイド」(<https://www.isus.jp/products/psxe/intelguide-index/>) を参照してください。

Y 同期構造を実装するプリミティブを選択する際、インテル® パフォーマンス・ツールを使用すると、マルチスレッディング機能の正当性に関する問題や、マルチスレッド実行時におけるパフォーマンスへの影響を判断する上で効果的です。インテル® パフォーマンス・ツールの機能の詳細については、付録 A で説明しています。

表 9-1 は、マルチスレッド・アプリケーションに利用可能な 3 つのカテゴリの同期オブジェクトの機能を比較するのに役立ちます。

表 9-1 同期オブジェクトのプロパティ

特性	オペレーティング・システムの同期オブジェクト	軽量のユーザー同期	MONITOR/MWAIT ベースの同期オブジェクト
取得と解放に要するサイクル (競合する場合)	数千または数万サイクル	数百サイクル	数百サイクル
消費電力	アイドル時はコアまたは論理プロセッサを停止して節電が可能	PAUSE を使用した場合はある程度節電可能	PAUSE よりも節電可能
スケジューリングとコンテキスト・スイッチ	競合がある場合は OS スケジューラーに戻る (スピン・ループ・カウンターの減少によってチューニング可能)	自発的には OS スケジューラーに戻らない	自発的には OS スケジューラーに戻らない
特権レベル	リング 0	リング 3	リング 0
その他	オブジェクトによっては、プロセス内同期やプロセス間通信を提供	複数のスレッドが同時に書き込む場合、同期変数へのアクセスをロックする必要があります。  その他の場合はロックなしで書き込み可能	軽量オブジェクトと同じ。  MONITOR/MWAIT をサポートするシステム上でのみ使用可能
推奨使用条件	Y アクティブなスレッドの数がコア数より多い Y 信号を数千サイクル待機 Y プロセス間で同期	Y アクティブなスレッドの数がコア数以下 Y 競合がまれにしか発生しない Y プロセス間の同期が必要	Y 軽量オブジェクトと同じ Y MONITOR/MWAIT が利用可能

### 9.4.2 短期間の同期

スレッドがほかのスレッドと同期する頻度と継続期間は、アプリケーションの特性によって異なります。同期ループで非常に高速な応答が必要な場合は、スピンウェイト・ループがアプリケーションで使用されます。

一般にスピンウェイト・ループは、あるスレッドが、別のスレッドが同期点に達するまで短時間待つ必要がある場合に使用されます。スピンウェイト・ループは、同期変数と事前定義値を比較するループから成ります。例 9-4(a) を参照してください。

スーパースケaler・スペキュレーティブ・エグゼキューション・エンジンを搭載した最近のマルチプロセッサーでは、このようなループによって、スピン中のスレッドからの同時読み込み要求が複数発行されることがあります。これらの要求はアウトオブオーダーを通常実行し、各読み込み要求は追加のバッファリソースを使用して割り当てられます。進行中のロードにワーカースレッドが書き込みを行ったことが検出されると、プロセッサーは、メモリーオーダー違反が発生しないよう保証する必要があります。未処理のメモリー操作の順序を正しく維持するには、プロセッサーに大きなペナルティーを課す必要があり、すべてのスレッドに影響が及ぶことが避けられません。

このペナルティーは、インテル® Pentium® M プロセッサー、インテル® Core™ Solo プロセッサー、インテル® Core™ Duo プロセッサーで発生します。ただし、これらのプロセッサーでのペナルティーは、インテル® Pentium® 4 プロセッサーやインテル® Xeon® プロセッサーと比べて小さめです。インテル® Pentium® 4 プロセッサー、インテル® Xeon® プロセッサーでは、このループを終了する性能のペナルティーは、約 25 倍以上です。

HT テクノロジー対応のプロセッサーの場合、スピンウェイト・ループは、プロセッサーの実行帯域幅を相当量消費することがあります。スピンウェイト・ループを実行している一方の論理プロセッサーは、もう一方の論理プロセッサーの性能に深刻な影響を与えることがあります。

#### 例 9-4 スピンウェイト・ループと PAUSE 命令

(a) スピンウェイト・ループが最適化されていないと、ループの終了時に性能のペナルティーが発生します。最適化されていないスピンウェイト・ループは、計算処理に貢献せずに実行リソースを消費します。

```
do {
    // このループは最も高速にメモリアクセスできます。
    // 他のワーカースレッドは、このスピンループが解決されるまで
    // sync_var を変更できません
} while( sync_var != constant_value);
```

(b) 高速スピンウェイト・ループに PAUSE 命令を挿入すると、スピニングスレッドおよびワーカースレッドに性能のペナルティーが生じません。

```
do {
    _asm pause
    // このループのパイプラインを分断する。sync_var が参照される間、
    // 1つ以上のロード要求が行われることを防止し、
    // ワーカースレッドが sync_var を更新する時の
    // パフォーマンス・ペナルティーを避けます。スピンスレッドはループを終了します。
}
while( sync_var != constant_value);
```

(c) 同期変数の可用性を判別するために、「テスト、テストおよび設定」手法を使用しているスピンウェイト・ループ。この手法は、インテル® 64 アーキテクチャー・プロセッサーと IA-32 アーキテクチャー・プロセッサー上で動作するスピンウェイト・ループを作成する際に推奨されます。

```
Spin_Lock:
    CMP lockvar, 0 ; // ロックがフリーかチェック
    JE Get_lock
    PAUSE; // 短い遅延
    JMP Spin_Lock;
Get_Lock:
    MOV EAX, 1;
    XCHG EAX, lockvar; // ロックの取得を試みる
    CMP EAX, 0; // 取得できたかテスト
    JNE Spin_Lock;
Critical_Section:
    <クリティカル・セクションのコード>
    MOV lockvar, 0; // ロックを解放
```

**ユーザー/ソース・コーディング規則 18 (影響 M、一般性 H):** 高速スピンループ中に PAUSE 命令を挿入し、ループの反復回数を最小限に抑え、システム全体の性能を向上させます。

Intel NetBurst® マイクロアーキテクチャー・コアを使用するプロセッサーの場合、スピンウェイト・ループからの終了のペナルティーは、PAUSE 命令をループに挿入することで回避できます。PAUSE 命令は、ループ内に若干の遅延

を挿入することで性能を改善し、同期変数へのストアを迅速に検出できる速度でメモリー読み込み要求を発行します。これにより、メモリーオーダー違反による長い遅延の発生を防ぐことができます。

単純なスピンウェイト・ループに PAUSE 命令を挿入する例を、例 9-4(b) に示します。PAUSE 命令は、すべてのインテル® 64 プロセッサおよび IA-32 プロセッサで互換性があります。Intel NetBurst® マイクロアーキテクチャーが採用される以前の IA-32 プロセッサでは、PAUSE 命令は基本的に NOP 命令と同等です。PAUSE 命令を使用してスピンウェイト・ループを最適化するその他の例は、アプリケーション・ノート AP-949 「インテル® Pentium® 4 プロセッサおよびインテル® Xeon® プロセッサにおけるスピンループの使用」 (<https://software.intel.com/sites/default/files/22/30/25602> (英語)) を参照してください。

PAUSE 命令を挿入すると、使用されるシステムリソースが少なくなるため、スピンウェイト中に消費される電力が大幅に減少する利点もあります。

### 9.4.3 スピンロックによる最適化

通常、スピンロックは、複数のスレッドで同期変数を変更する必要がある場合、およびその同期変数が誤って上書きされないようにロックで保護する必要がある場合に使用されます。しかしロックを解放すると、同時に複数のスレッドがそのロックを競って取得しようとすることがあります。こうしたスレッドのロック競合が発生すると、周波数、個別のプロセッサ数、HT テクノロジーに関連するパフォーマンス・ケーリングが大幅に減少する恐れがあります。

パフォーマンスのペナルティーを削減する方法の 1 つは、多数のスレッドが同じロックを競って取得する可能性を減らし、複数のスレッド間で共有する必要のあるデータをソフトウェア・パイプライン手法により処理することです。

複数のスレッドで 1 つのロックを競合させるのではなく、2 つのスレッドしか 1 つのロックに書き込みできないようにする必要があります。アプリケーションでスピンロックを使用する必要がある場合、ウェイトループに PAUSE 命令を挿入します。例 9-4(c) は、スピンウェイト・ループ内のロックの可用性を判別するための「テスト、テストおよび設定」手法の例を示しています。

**ユーザー/ソース・コーディング規則 19 (影響 M、一般性 L):** 複数のスレッドが取得可能なスピンロックを、2 つのスレッドしか 1 つのロックに書き込めないようなパイプライン化されたロックと置き換えます。2 つのスレッドが共有する変数に、1 つのスレッドしか書き込みを行う必要がない場合は、ロックを使用する必要はありません。

### 9.4.4 長期間の同期

長期間ロックを保持するスピンウェイト・ループでは、アプリケーションで次の 2 つのガイドラインに従う必要があります。

- Y スピンウェイト・ループの継続期間を、最小の繰り返し回数になるように維持します。
- Y アプリケーションで OS サービスを使用して、待機中のスレッドをブロックする必要があります。これにより、プロセッサが解放され、その他の実行可能スレッドがプロセッサまたは提供されている実行リソースを利用できるようになります。

HT テクノロジー対応のプロセッサにおいて、一方の論理プロセッサがアクティブで、もう一方の論理プロセッサが非アクティブの場合は、オペレーティング・システムは HLT 命令を使用する必要があります。HLT 命令によって、アイドル状態の論理プロセッサを停止状態に遷移できます。これにより、アクティブな論理プロセッサは、物理パッケージのハードウェア・リソースをすべて利用できるようになります。この手法を持たないオペレーティング・システムは、アイドル状態の論理プロセッサ上で命令を実行する必要があり、繰り返しチェックされます。この「アイドルループ」が発生すると、他方のアクティブな論理プロセッサ上で処理を進行するために使用する実行リソースが消費されます。

アプリケーション・スレッドが長時間アイドル状態を維持する必要がある場合は、そのアプリケーションでスレッド・ブロッキング API、またはその他の手法を使用して、アイドル状態のプロセッサを解放する必要があります。ここで



説明する手法は、従来の MP システムに適用されますが、HT テクノロジーに対応したプロセッサでも、かなり高い効果が得られます。

一般に、オペレーティングシステムは、Sleep(dwMilliseconds)<sup>10</sup> などのタイミングサービスを提供しています。このような機能を使用すると、同期変数が頻繁にチェックされなくなります。

複数のワーカースレッドおよび制御ループを同期するもう 1 つの手法は、OS が提供するスレッド・ブロッキング API を使用することです。スレッド・ブロッキング API を使用すると、スピニングおよびウェイトリング用のプロセッサ・サイクルが、制御スレッドであまり消費されなくなります。これにより、OS は、使用可能なプロセッサ上のワーカースレッドをスケジューリングするより長い時間が得られます。さらに、スレッド・ブロッキング API を使用すると、OS が HLT 命令を使用して実装するシステム・アイドルループの最適化からも利点が得られます。

**ユーザー/ソース・コーディング規則 20 (影響 H、一般性 M):** スレッド・ブロッキング API を長いアイドルループ内で使用し、プロセッサを解放します。

実行可能なスレッドの数が MP システム内のプロセッサの数よりも少ない場合は、スピンウェイトループを従来の MP システムで使用してもあまり問題はありません。しかし、アプリケーション内のスレッド数が、シングルプロセッサ・システムまたはマルチプロセッサのプロセッサ数よりも多くなると予測される場合、スレッド・ブロッキング API を使用してプロセッサ・リソースを解放します。1 つの制御スレッドを使用して複数のワーカースレッドを同期するマルチスレッド・アプリケーションでは、ワーカースレッド数をシステム内のプロセッサ数以下に制限し、制御スレッド内でのスレッド・ブロッキング API の使用を検討する必要があります。

#### 9.4.4.1 スレッド同期におけるコーディングの落とし穴の回避策

複数スレッド間の同期において、個別のプロセッサ数や物理プロセッサあたりの論理プロセッサ数に応じてパフォーマンスのスケールアップを高めるには、設計および実装を慎重に行う必要があります。1 つの手法で、どのような同期状況にも適用できる万能な解決策はありません。

次の例 9-5(a) に示す疑似コードは、制御スレッドに対するポーリングループの実装例を示しています。一般に、実行可能なワーカースレッドが 1 つしか存在しない場合は、Sleep(0) などのタイミングサービス API を呼び出しても、スレッド同期のコストを最小限に抑える効果は得られません。制御スレッドは、依然として、高速スピニンググループのように動作するため、唯一の実行可能なワーカースレッドは、実行リソースをスピンウェイトループと共有する必要があります (HT テクノロジー対応の同一の物理プロセッサ上で両方が実行されている場合)。実行可能なワーカースレッドが複数存在している場合、Sleep(0) などのスレッド・ブロッキング API を呼び出すと、スピンウェイトループを実行しているプロセッサが解放され、スピニンググループの代わりに別のワーカースレッドによってプロセッサが使用されます。

一般に、ワーカースレッドが完了に長時間を要する場合、ワーカースレッドの完了を待機している制御スレッドは、スレッド・ブロッキング API やタイムサービスを使用して、スレッド同期を実装できます。例 9-5(b) は、スレッド同期で、制御スレッドのオーバーヘッドを削減する例を示しています。

<sup>10</sup> Sleep() API は、プロセッサの解放を保証しないため、スレッド・ブロッキングではありません。例 9-5(a) は、Sleep(0) の使用例を示しています。Sleep(0) では、必ずしもプロセッサが別のスレッドに解放されるとは限りません。

例 9-5 スピンウェイト・ループを使用するコーディングの落とし穴

(a) スピンウェイト・ループは、不適切にプロセッサを解放しようとする。唯一のワーカー・スレッドと制御スレッドが同一の物理プロセッサ・パッケージ上で実行されている場合、スピンウェイト・ループでは、パフォーマンスのペナルティが発生します。

```
// 1つのワーカー・スレッドのみが実行されており、
// 制御ループはワーカー・スレッドの終了を待つ
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
    Sleep(0) // 即座にスピンループに戻る
...
}
```

(b) ポーリングループは、プロセッサを正常に解放します。

```
// 1つのワーカー・スレッドを実行し、完了を待機させます。
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
    Sleep(FIVE_MILLISEC)

// このプロセッサはわずかの間解放されます。
// その間他のスレッドがプロセッサを利用できます。
...
}
```

通常、スレッドを同期するときは、OS 関数呼び出しを慎重に使用する必要があります。OS でサポートされるスレッド同期オブジェクト (クリティカル・セクション、ミューテックス、セマフォなど) を使用するときは、クリティカル・セクションなどの最小の同期オーバーヘッドを持つ OS サービスを使用すべきです。

9.4.5 変更されたデータの共有とフォルス・シェアリングの防止

プロセッサ/コアのトポロジーと特定のマイクロアーキテクチャーにおけるキャッシュトポロジーによっては、一方のコアで実行中のスレッドが、他方のコアの 1 次キャッシュに変更状態で存在するデータに対して読み出しまたは書き込みを試みると、変更されたデータの共有によってパフォーマンスのペナルティが発生します。この場合、変更されたキャッシュラインがメモリーに排出され、他方のコアの 1 次キャッシュに再度読み込まれます。このようなキャッシュライン転送のレイテンシーは、1 次キャッシュまたは 2 次キャッシュ内のデータを直接使用する場合よりもはるかに大きくなります。

フォルス・シェアリングが発生するのは、別のスレッドが使用する異なるデータと、スレッドが使用するデータが同じキャッシュライン上に存在する場合です。このような状況では、プラットフォーム上の論理プロセッサ/コアのトポロジーに応じて、パフォーマンス上の遅延が発生することもあります。

異なる物理プロセッサ上の論理プロセッサでスレッドが実行されていると、フォルス・シェアリングによってパフォーマンスのペナルティが発生する場合があります。HT テクノロジー対応のプロセッサでは、異なるコア上、異なる物理プロセッサ上、または物理プロセッサ・パッケージ内の 2 つの論理プロセッサ上で 2 つのスレ

ドが実行されていると、フォルス・シェアリングによってパフォーマンスのペナルティーが発生します。異なるコア上または異なる物理プロセッサ上で 2 つのスレッドが実行されている場合、パフォーマンスのペナルティーは、キャッシュ整合性を維持するためのキャッシュ排出によって生じます。物理プロセッサ・パッケージ内の 2 つの論理プロセッサ上で 2 つのスレッドが実行されている場合、性能のペナルティーは、メモリー・オーダー・マシクリアー条件によって発生します。

マルチスレッド化されたソフトウェアでフォルス・シェアリングのペナルティーを防ぐ一般的な方法は、「フォルス・シェアリングのしきい値」サイズに応じてクリティカルなデータを分離して配置するか、アライメントの粒度でロックします。次のステップで、ソフトウェアは各世代のインテル® プロセッサに適用できる「フォルス・シェアリングのしきい値」を決定できます。

1. プロセッサが CLFLUSH 命令をサポートする場合、つまり PUID.01H:EDX:CLFLUSH[ビット 19]=1 のケース。

CLFLUSH ラインサイズ、CPUID.01H:EBX[15:8] の整数値を「フォルス・シェアリングのしきい値」として使用します。

2. CLFLUSH ラインサイズが利用できない場合、以下に示す CPUID リーフ 4 を使用します。

CPUID リーフ 4 のサブリーフを介して報告される有効なキャッシュタイプで最も大きなシステム・コヒーレンシーのラインサイズを評価することで、「フォルス・シェアリングのしきい値」を決定します。それぞれのサブリーフ n は、システムのコヒーレンシー・ラインサイズに関連する (CPUID.(EAX=4, ECX=n):EBX[11:0] + 1) です。

3. CLFLUSH ラインサイズと CPUID リーフ 4 のどちらも利用できない場合、ソフトウェアは次のいずれかの方法で「フォルス・シェアリングのしきい値」を選択します。
  - a. CPUID リーフ 2 のディスクリプター・テーブルを取得し、利用可能なディスクリプターのエン트리から選択します。
  - b. プラットフォームで利用可能なファミリー/モデル固有メカニズム、または既知のファミリー/モデル固有値。
  - c. 安全なデフォルト値を 64 バイトとします。

**ユーザー/ソース・コーディング規則 21 (影響 H、一般性 M):** キャッシュライン、またはセクター内のフォルス・シェアリングに注意します。「フォルス・シェアリングのしきい値」ほど小さくないアライメントの粒度で分離して、重要なデータやロックを割り当てます。

パラメーターの共通ブロックが親スレッドから複数のワーカー・スレッドに渡される場合、各ワーカー・スレッドが頻繁にアクセスされるプライベート・コピー (それぞれのコピーは「フォルス・シェアリングのしきい値」の倍数に配置) をパラメーター・ブロックに作成することが理想的です。

## 9.4.6 共有同期変数の配置

Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサでは通常、バス読み込みで 128 バイトがフェッチされ、キャッシュに格納されます。そのため、キャッシュデータの排出を最小限に抑えるための最適な挿入間隔は、128 バイトとなります。フォルス・シェアリングを防止するため、同期変数とシステム・オブジェクト (クリティカル・セクションなど) は、128 バイト領域内に単独で存在するように割り当て、128 バイト境界にアライメントする必要があります。

例 9-6 は、MP システムでキャッシュ整合性を維持するために必要なバス・トラフィックを最小限に抑える方法を示しています。この手法は、HT テクノロジー対応または非対応のプロセッサを使用する MP システムにも適用されます。

例 9-6 同期変数およびレギュラー変数の配置

```
int regVar;
int padding[32];
int SynVar[32*NUM_SYNC_VARS];
int AnotherVar;
```

インテル® Pentium® M プロセッサ、インテル® Core™ Solo プロセッサ、インテル® Core™ Duo プロセッサ、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサでは、同期変数を独立したキャッシュラインに単独で格納して、フォルス・シェアリングを回避します。ソフトウェアでは、同期変数がページ境界をまたぐことを許可してはなりません。

**ユーザー/ソース・コーディング規則 22 (影響 M、一般性 ML):** 各同期変数を 128 バイトで分離して単独に配置するか、独立したキャッシュラインに格納します。

**ユーザー/ソース・コーディング規則 23 (影響 H、一般性 L):** スピンロック変数は、キャッシュライン境界をまたいで配置してはなりません。

以下の場合、コードレベルでフォルス・シェアリングに関する考慮が必要です。

- Y 同じキャッシュラインに配置され、異なるスレッドによって書き込まれるグローバルデータ変数とスタティック・データ変数。
- Y 異なるスレッドによって動的に割り当てられるオブジェクトが、キャッシュラインを共有する可能性がある場合。一方のスレッドによって局所的に使用される変数が、他方のスレッドとキャッシュラインを共有しない方法で割り当てられることを確認する必要があります。

同期変数のアライメントを強制し、キャッシュラインの共有を回避するための別の手法としては、データ構造の宣言時にコンパイラ・ディレクティブを使用します。図 9-7 を参照してください。

例 9-7 キャッシュラインを共有しない同期変数の宣言

```
__declspec(align(64)) unsigned __int64 sum;
struct sync_struct {...};
__declspec(align(64)) struct sync_struct sync_var;
```

フォルス・シェアリングを回避するためのその他の手法を示します。

- Y データ構造内の各種の変数を整理します (コンパイラがデータ変数に与えるレイアウトは、ソースコードでの配置と異なる場合があります)。
- Y 各スレッドが変数セットの独自のコピーを使用する必要がある場合は、以下を利用して変数を宣言します。
  - OpenMP\* 使用時は、threadprivate ディレクティブ
  - Microsoft\* コンパイラ使用時は、\_\_declspec(thread) 修飾子
- Y オブジェクトを自動的に割り当てるマネージド環境では、オブジェクト・アロケータとガベージコレクターは、2つのオブジェクトによるフォルス・シェアリングが発生しないように、メモリー内でオブジェクトをレイアウトする必要があります。
- Y 1つのスレッドのみが各オブジェクト・フィールドと近くのオブジェクト・フィールドに書き込めるクラスを提供して、フォルス・シェアリングを回避します。

この節で説明されている推奨事項を、配置がまばらなデータレイアウトの推奨と理解してはいけません。データレイアウトに関する推奨事項は必要時にのみ適用して、ワークセットのサイズが不用意に拡大することを避けるべきです。

## 9.5 システムバスの最適化

システムバスは、バス・エージェント（論理プロセッサなど）からの要求を処理して、メモリー・サブシステムからデータやコードをフェッチします。メモリーからフェッチされるデータ・トラフィックがパフォーマンスに及ぼす影響は、ワークロードの特性、メモリーアクセスにおけるソフトウェア最適化の度合い、ソフトウェア・コードにおける局所性の改善によって異なります。ワークロードのメモリー・トラフィックの特性評価を行う手法については、付録 A で説明します。局所性の改善に関する最適化ガイドラインについては、3.6.11 節「局所性の改善」と 7.5.11 節「ハードウェア・プリフェッチとキャッシュ・ブロッキング手法」で説明しています。

第 3 章と第 7 章で説明した手法を利用すると、バスシステムがシングルスレッド環境に対してサービスを提供しているプラットフォーム上でアプリケーションの性能が向上します。マルチスレッド環境の場合、バスシステムは通常、多数の論理プロセッサにサービスを提供しているため、各論理プロセッサが独立してバス要求を発行することがあります。そのため、局所性の改善、バス帯域幅の保持、間隔の広いキャッシュミスでの遅延の削減に関する手法は、プロセッサ・スケージングのパフォーマンスに大きな影響を与える場合があります。

### 9.5.1 バス帯域幅の保持

マルチスレディング環境では、複数のバス・エージェントから要求されたメモリー・トラフィックによってバス帯域幅が共有される場合があります（バス・エージェントは、複数の論理プロセッサや複数のプロセッサ・コアである場合があります）。バス帯域幅を保持すると、プロセッサのスケージング・パフォーマンスを高めることができます。また、大きなストライドのキャッシュミスが大量にある場合、一般的に実効バス帯域幅が減少します。大きなストライドのキャッシュミス（または DTLB ミス）を減らすと、それに起因する帯域幅が減少する問題が緩和されます。

バスコマンド帯域幅を保持する方法の 1 つは、コードとデータの局所性を改善することです。データの局所性を改善すると、キャッシュラインの排出回数およびデータフェッチの要求回数が減少します。この手法はまた、システムメモリーからの命令フェッチの回数を減少させます。

**ユーザー/ソース・コーディング規則 24 (影響 M、一般性 H):** データおよびコードの局所性を改善し、バスコマンド帯域幅を保持します。

プロファイルに基づく最適化をサポートするコンパイラーを使用して、頻繁に実行されるコードパスをキャッシュ内に留めることで、コードの局所性が改善されます。その結果、命令フェッチの回数が少なくなります。また、ループ・ブロッキングによりデータの局所性も改善できます。その他の局所性を改善する手法は、マルチスレディング環境にも適用してバス帯域幅を保持できます（7.5 節「プリフェッチを使用したメモリーの最適化」を参照してください）。

システムバスは多くのバス・エージェント（論理プロセッサやプロセッサ・コア）間で共有されるため、ソフトウェアのチューニングではバスが飽和状態になる兆候を認識すべきです。効果的な手法の 1 つは、バス読み出しトラフィックのキューの深さを調べることです。バスキューが深い場合、局所性を改善してキャッシュの使用効率を高めると、その他の手法（挿入するソフトウェア・プリフェッチの増加や、重複したバス読み出しによるメモリー・レイテンシーのマスクなど）よりもパフォーマンス上のメリットがあります。バスが飽和しない状態でソフトウェアを動作させる作業ガイドラインとして、バス読み出しキューの深さが 5 を大きく下回っているかどうかを確認する方法があります。

一部の MP プラットフォームとワークステーション・プラットフォームでは、2 つのシステムバスを備え、各バスが 1 つ以上の物理プロセッサに対応するチップセットを搭載しています。バス帯域幅の保持に関する上記のガイドラインは、それぞれのバスドメインにも適用されます。

### 9.5.2 バスとキャッシュとの相互作用について

ワーキングセット全体が 2 次キャッシュに収まらないデータセットや、使用する帯域幅がバスの能力を超えたデータセットがある場合、そのコード領域の並列化には注意を払う必要があります。インテル® Core™ Duo プロセッサでは、1 つのスレッドのみが 2 次キャッシュやバスを使用する場合、そのスレッドの処理は他方のコアによって干渉

されないため、キャッシュやバスシステムを最大限に利用できます。ただし、2 つのスレッドが 2 次キャッシュを同時に使用する場合、以下の条件のいずれかが当てはまると、パフォーマンスが低下する可能性があります。

- Y ワーキングセットの合計が 2 次キャッシュのサイズよりも大きい。
- Y バス使用量の合計がバスの能力よりも多い。
- Y 両者が 2 次キャッシュ内の同じセットに頻繁にアクセスし、少なくとも 1 つのスレッドがキャッシュラインへの書き込みを行う。

マルチスレッド・ソフトウェアでこのような落とし穴を回避するには、一度に 1 つのスレッドのみが 2 次キャッシュにアクセスする並列処理や、2 次キャッシュとバス使用量が制限を超えない並列処理を検討する必要があります。

### 9.5.3 過度なソフトウェア・プリフェッチを避ける

インテル® Pentium® 4 プロセッサとインテル® Xeon® プロセッサは、自動ハードウェア・プリフェッチを備えています。自動ハードウェア・プリフェッチでは、事前の参照パターンに基づいて、データと命令をユニファイド 2 次キャッシュに格納できます。ほとんどの場合、ハードウェア・プリフェッチは、ソフトウェア・プリフェッチの明確な介入がなくても、システム・メモリー・レイテンシーを削減できます。また、コードのデータ・アクセス・パターンを調整し、自動ハードウェア・プリフェッチの特性を活かして、局所性の改善やメモリー・レイテンシーのマスクを行うことが推奨されます。インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサも、先進的なハードウェア・プリフェッチ機構を提供します。旧世代のハードウェア・プリフェッチ機構を利用できるデータ・アクセス・パターンは、通常、新しい世代のハードウェア・プリフェッチ機構にも利用できます。

ソフトウェア・プリフェッチ命令を過度または無計画に使用すると、間違いなくパフォーマンスのペナルティーが生じます。これは、システムバスのコマンドとデータ帯域幅が無駄に使用されるためです。

ソフトウェア・プリフェッチを使用すると、プロセッサ・コアが必要とするデータをハードウェア・プリフェッチがフェッチする時期が遅れます。また、ソフトウェア・プリフェッチは、実行リソースを消費するため、実行がストールします。状況によっては、ソフトウェア・プリフェッチの削減または使用しないことを検討し、ハードウェア・プリフェッチ機構を有効活用の方が効果的です。ソフトウェア・プリフェッチ命令を使用するガイドラインは、第 3 章で説明しています。自動ハードウェア・プリフェッチを活用する手法については、第 7 章で説明しています。

**ユーザー/ソース・コーディング規則 25 (影響 M、一般性 L):** ソフトウェア・プリフェッチ命令の過度な使用は避け、自動ハードウェア・プリフェッチを機能させます。ソフトウェア・プリフェッチを過度に使用すると、バス利用が大幅かつ不必要に増加する可能性があります (不適切に使用されている場合)。

### 9.5.4 キャッシュミスの実効レイテンシーを改善

キャッシュミスによるシステムメモリーのアクセスレイテンシーは、バス・トラフィックの影響を受けます。これは、バス読み込み要求は、その他のバス・トランザクション要求とともに調整されるためです。未処理のバス・トランザクション数を削減すると、実効メモリー・アクセス・レイテンシーが改善されます。

メモリー読み込みトランザクションの実効レイテンシーを改善する手法として、複数のバス読み込みをオーバーラップすることにより、分散した読み込みのレイテンシーを減らす方法があります。データの局所性がほとんどない、またはメモリー読み込みと他のバス・トランザクションを調整する必要がある状況では、複数のメモリー読み込みを連続して発行し、未処理のメモリー読み込みトランザクションをオーバーラップすることにより、分散したメモリー読み込みの実効レイテンシーを改善できます。一般に、連続したバス読み込みの平均レイテンシーは、他のバス・トランザクションとともに間隔を開けて配置された (分散した) 読み込みの平均レイテンシーよりも低くなります。キャッシュミスの完全な遅延が避けられないのは、最初のメモリー読み込みだけです。

**ユーザー/ソース・コーディング規則 26 (影響 M、一般性 M):** オーバーラップする複数の連続したメモリー読み込みにより、実効キャッシュ・ミス・レイテンシーを改善することを検討します。

最終レベルキャッシュで連続するキャッシュミスを引き起こすアクセス間隔が、自動ハードウェア・プリフェッチを引き起こすしきい値の距離よりも狭くなるようにデータ・アクセス・パターンを調整できれば、実効メモリー・レイテンシーを削減するもう 1 つの手法が利用可能となります。7.5.3 節「ハードウェア・プリフェッチで実効レイテンシーを削減する例」を参照してください。

**ユーザー/ソース・コーディング規則 27 (影響 M、一般性 M):** 最終レベルキャッシュの連続するキャッシュミスの間隔が 64 バイトに近づくように、メモリー参照の順序付けを調整することを検討します。

### 9.5.5 フルサイズの手書き込みトランザクションによる高データレートの実現

バスを介してトランザクションが書き込みを行う場合、64 バイトのキャッシュライン・サイズすべて、またはその一部分が物理メモリーへ書き込まれます。後者は、パーシャル書き込みと呼ばれます。一般に、ライトバック (WB) メモリーアドレスではフルサイズの手書き込みとなり、ライトコンバイン (WC) またはキャッシュ不可能 (UC) メモリーアドレスではパーシャル書き込みとなります。キャッシュされた WB ストア操作と WC ストア操作では、6 つの WC バッファー (64 バイト幅) を使用して、これらの書き込みトランザクションのトラフィックが管理されます。WC バッファーへの書き込みがすべて完了する前に、競合するトラフィックによってバッファーがクローズされると、単一の 64 バイト書き込みトランザクションではなく、一連の 8 バイトのパーシャル・バス・トランザクションが発生します。

**ユーザー/ソース・コーディング規則 28 (影響 M、一般性 M):** フルサイズの手書き込みトランザクションを使用して、より高いデータ・スループットを達成します。

多くの場合、WC メモリーへのパーシャル書き込みが複数発生する場合、ソフトウェアによるライト・コンバイニング手法を使用することでそれらをフルサイズの手書き込みに結合し、WB ストア・トラフィックとの競合から WC ストア操作を分離できます。ソフトウェアによるライト・コンバイニングを実装するには、WC 属性が指定されたメモリーへのキャッシュ不可能書き込みが、1 次データキャッシュに収まる小さな一時バッファー (WB タイプ) に書き込まれるようにします。一時バッファーが一杯になると、アプリケーションは最終的な WC デスティネーションに一時バッファーの内容をコピーします。

バス上でパーシャル書き込みを処理すると、システムメモリーに対する有効なデータレートが、システムバス帯域幅の 1/8 に削減されます。

## 9.6 メモリー最適化

効率的なキャッシュ利用は、メモリー最適化の重要な要素です。キャッシュを効率的に利用するには、以下の課題を考慮する必要があります。

- Y キャッシュ・ブロッキング
- Y 共有メモリー最適化
- Y 64KB エイリアス・データ・アクセスの排除
- Y 1 次キャッシュの過度な排出防止

### 9.6.1 キャッシュ・ブロッキングのテクニック

ループ・ブロッキングは、キャッシュミスの削減とメモリーアクセスのパフォーマンス向上に役立ちます。ループ・ブロッキング手法を適用する場合、ブロックサイズを適切に選択することが重要です。ループ・ブロッキングは、HT テクノロジー対応または非対応のプロセッサ上で動作するマルチスレッド・アプリケーションだけではなく、シングルスレッド・アプリケーションにも適用できます。ループ・ブロッキングでは、メモリー・アクセス・パターンを目的のキャッシュサイズに効果的に収まるブロックに変換します。

HT テクノロジー対応のインテル® プロセッサの場合、ユニファイド・キャッシュ向けのループ・ブロッキング手法では、目的のキャッシュサイズの 2 分の 1 を超えないブロックサイズを選択します (2 つの論理プロセッサがキャッシュを共有している場合)。ループ・ブロッキングのブロックサイズの上限は、目的のキャッシュサイズを、物理プ

ロセッサ・パッケージで利用可能な論理プロセッサ数で割ることで決定します。一般に、キャッシュ・ブロッキングで使用するソースまたはターゲットバッファに含まれないデータにアクセスするには、複数のキャッシュラインが必要となります。そのため、ブロックサイズは、ターゲットのキャッシュの 4 分の 1 から 2 分の 1 の間で選択します (第 3 章「一般的な最適化ガイドライン」を参照)。

ソフトウェアは、CPUID のキャッシュ・パラメーター・リーフを参照して、特定のキャッシュをどの論理プロセッサのサブセットが共有しているか検出できます (第 7 章「キャッシュ利用の最適化」を参照)。したがって、特定のキャッシュをアクセスする論理プロセッサ数でそのキャッシュの合計サイズを割ったものを、ブロックサイズの上限に設定することで、上記のガイドラインの適用を拡張できます。その結果、特定のキャッシュをアクセスするすべての論理プロセッサがそのキャッシュを同時に使用できるようになります。この手法は、マルチタスク・ワークロードの一部として使用されるシングルスレッド・アプリケーションにも適用できます。

**ユーザー/ソース・コーディング規則 29 (影響 H, 一般性 H):** キャッシュ・ブロッキングを使用して、データアクセスの局所性を改善します。HT テクノロジー対応のインテル®プロセッサが対象の場合、キャッシュサイズの 4 分の 1 から 2 分の 1 を目標とします。または、特定のキャッシュをアクセスするすべての論理プロセッサがそのキャッシュを同時に共有可能なブロックサイズを目標とします。

## 9.6.2 メモリー最適化

多くの場合、個々のプロセッサ間でキャッシュの整合性を維持するには、プロセッサ周波数より大幅に低いクロックレートで動作するバスを介して、データを転送する必要があります。

### 9.6.2.1 物理プロセッサ間でのデータ共有の最小化

一般に、2 つのスレッドが 2 つの物理プロセッサ上でデータを共有して実行する場合、共有データに対して読み込みまたは書き込みを行うには、いくつかのバス・トランザクション (スヌーピング、所有権の変更要求、バスを介したデータのフェッチなど) が必要となります。そのため、大量の共有メモリーにアクセスするスレッドは、プロセッサのスケールン・パフォーマンスを低下させる可能性があります。

**ユーザー/ソース・コーディング規則 30 (影響 H, 一般性 M):** バスを共有する別々のバス・エージェント上で実行されるスレッド間で、データの共有を最小限に抑えます。プラットフォームが複数のバスドメインで構成されている場合、バスドメイン間でのデータ共有も最小限に抑える必要があります。

データの共有を最小化する手法の 1 つは、ローカルスタック変数にデータをコピーすることです (長時間繰り返しアクセスされる場合)。必要であれば、共有メモリー・ロケーションに書き戻すとき、複数のスレッドからの結果を結合します。この方法では、共有データアクセスの同期で使用する時間を最小限に抑えられます。

### 9.6.2.2 バッチ方式の生産-消費モデル

スレッド化された生産-消費モデル (図 9-5 を参照) のメリットは、共有 2 次キャッシュを使って生産と消費との間でデータを共有しながら、バス・トラフィックを最小限に抑えられることです。インテル® Core™ Duo プロセッサでワークバッファが 1 次キャッシュ内に収まる場合、最適なパフォーマンスを得るには生産タスクと消費タスクの再順序付けが必要となります。これは、L2 から L1 にデータをフェッチする方が、一方のコアのキャッシュラインを無効化してバスからフェッチするよりも、はるかに高速であるためです。

図 9-5 に示すバッチ方式の生産-消費モデルを利用すると、標準の生産-消費モデルで小型のワークバッファを使用する際の欠点も克服できます。バッチ方式の生産-消費モデルでは、各スケジューリング単位で 2 つまたは 3 つの生産タスクがまとめられ、それぞれ指定のバッファで処理されます。まとめられるタスクの数は、ワーキングセット全体が 1 次キャッシュより大きく、2 次キャッシュよりも小さくなることを基準として決定されます。



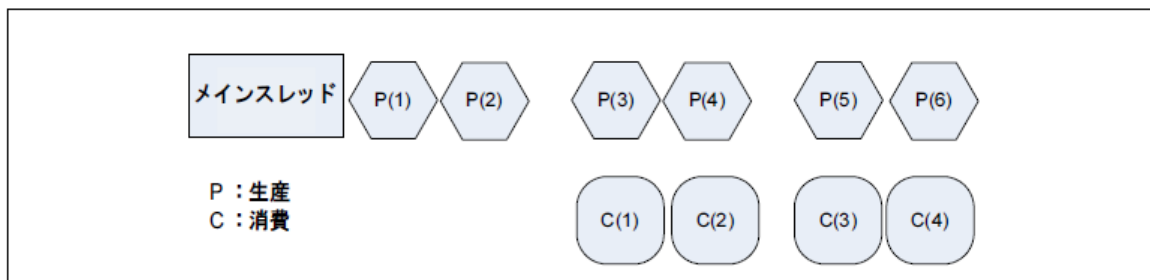


図 9-5 バッチ方式の生産-消費モデル

例 9-8 に、生産および消費のスレッド機能のバッチ方式での実装を示します。

例 9-8 生産および消費スレッドのバッチ方式での実装

```

void producer_thread()
{ int iter_num = workamount - batchsize;
  int model;
for (model=0; model < batchsize; model++)
{ produce(bufs[model],count); }
  while (iter_num--)
  { Signal(&signal1,1);
    produce(bufs[model],count); // プレースホルダー関数
    WaitForSignal(&end1);
    model++;
    if (model > batchsize)
      model = 0;
  }
}

void consumer_thread()
{ int mode2 = 0;
  int iter_num = workamount - batchsize;
  while (iter_num--)
  { WaitForSignal(&signal1);
    consume(bufs[mode2],count); // プレースホルダー関数
    Signal(&end1,1);
    mode2++;
    if (mode2 > batchsize)
      mode2 = 0;
  }
  for (i=0;i<batchsize;i++)
  { consume(bufs[mode2],count);
    mode2++;
    if (mode2 > batchsize)
      mode2 = 0;
  }
}
    
```

### 9.6.3 64KB エイリアスのデータアクセスを排除

64KB エイリアスの条件については、第 3 章で詳しく説明しています。64KB エイリアスの条件に該当するメモリーアクセスでは、1 次データキャッシュが過度に排出される可能性があります。一般に、各スレッドに起因する 64KB エイリアス・データ・アクセスを排除すると、周波数スケーリングの改善に役立ちます。また、アプリケーションで HT テクノロジーを活用している場合、1 次データキャッシュを効率良く利用できます。

**ユーザー/ソース・コーディング規則 31 (影響 H、一般性 H):** 各スレッドで 64KB の倍数でオフセットされるデータ・アクセス・パターンを最小限にします。

インテル® Pentium® 4 プロセッサのパフォーマンス監視イベントを使用すると、64KB エイリアス・データ・アクセスを検出できます。インテル® Pentium® 4 プロセッサのパフォーマンス・メトリックは、付録 B で説明します。これらのメトリックは、インテル® VTune™ Amplifier を使用してアクセスされるイベントに基づいています。

64KB エイリアスに関連するパフォーマンスのペナルティーは、主に HT テクノロジーまたは Intel NetBurst® マイクロアーキテクチャーを実装したプロセッサに適用されます。次の節では、HT テクノロジー対応のプロセッサ上で動作するマルチスレッド・アプリケーションに適用されるメモリー最適化手法について説明します。

## 9.7 フロントエンドの最適化

ユニファイド 2 次キャッシュが 2 つのプロセッサ・コアによって共有されるデュアルコア・プロセッサ (インテル® Core™ Duo プロセッサおよびインテル® Core™ マイクロアーキテクチャー・ベースのプロセッサ) の場合、マルチスレッド・ソフトウェアでは、ユニファイド・キャッシュからコードをフェッチする 2 つのスレッドが原因でコード・ワーキング・セットが増加するため、フロントエンドとキャッシュの最適化において考慮する必要があります。インテル® Core™ マイクロアーキテクチャー・ベースのクアドコア・プロセッサの場合、インテル® Core™2 Duo プロセッサに適用される考慮事項がクアドコア・プロセッサにも適用されます。

### 9.7.1 過度なループアンロールの回避

ループをアンロールすると分岐の数が少なくなり、アプリケーション・コードの分岐予測が改善されます。ループアンロールの詳細については、第 3 章で説明しています。ループアンロールは慎重に行う必要があります。分岐予測が改善することの利点、およびループストリーム検出器 (LSD) の利用率が低下するコストについて検討する必要があります。

**ユーザー/ソース・コーディング規則 32 (影響 M、一般性 L):** 過度なループアンロールを回避し、LSD が効率良く動作するようにします。

## 9.8 アフィニティーと共有プラットフォーム・リソースの管理

現代の OS は、論理プロセッサや NUMA (Non-Uniform Memory Access) メモリー・サブシステムなど特定の共有リソースをアプリケーションが管理できるように API やデータ構造 (アフィニティー・マスクなど) を提供しています。

マルチスレッド・ソフトウェアでは、表 9-2 の推奨事項を考慮した上で、アフィニティー API の使用を検討すべきです。

表 9-2 設計時におけるリソース管理の選択

ランタイム環境	スレッド・スケジューリング/プロセッサ・アフィニティの考慮事項	メモリー・アフィニティの考慮事項
シングルスレッド・アプリケーション	OS スケジューラーにスケジュールを管理させることで、システムの応答性とスループットに関する OS スケジューラーの目標をサポートします。OS は、エンドユーザーがランタイム固有の環境を最適化する機能を提供します。	該当しません。OS に任せます。
次の条件に当てはまるマルチスレッド・アプリケーション: i) システム上のすべてのプロセッサ・リソースを使用するわけではない。 ii) システムリソースをほかの並列アプリケーションと共有する。 iii) ほかの並列アプリケーションのほうが優先度が高い。	OS のデフォルト・スケジューラー・ポリシーに依存します。  ハードコードされたアフィニティ・バインディングは、システムの応答性やスループットに悪影響を与えることがあり、場合によってはアプリケーションのパフォーマンスにも影響します。	OS のデフォルト・スケジューラー・ポリシーに依存します。  明示的に NUMA を管理することなく透過的に NUMA のメリットを提供できる API を使用します。
次の条件に当てはまるマルチスレッド・アプリケーション: i) フォアグラウンドで実行され優先度が高い。 ii) システム上のすべてのプロセッサ・リソースを使用するわけではない。 iii) システムリソースをほかの並列アプリケーションと共有する。 iv) ただし、ほかの並列アプリケーションのほうが優先度が低い。	アプリケーション向けにカスタマイズされたスレッド・バインディング・ポリシーについて検討する場合、ハードコードされたスレッド・アフィニティ・バインディング・ポリシーではなく、OS スケジューラーを使用した協調的アプローチを採用すべきです。例えば、SetThreadIdealProcessor() を使用すると、局所性が最適化されたアプリケーション・バインディング・ポリシーに次のフリーコア・バインディング・ポリシーを固定するフローティング・ベースを提供して、デフォルト OS ポリシーと協調できます。	明示的に NUMA を管理することなく透過的に NUMA のメリットを提供できる API を使用します。  デフォルト OS ポリシーが原因でパフォーマンス上の問題が発生する場合、パフォーマンス・イベントを使用してローカルでないメモリアクセスの問題を診断します。
フォアグラウンドで実行されるマルチスレッド・アプリケーションであり、システム上のすべてのプロセッサ・リソースを必要とし、システムリソースを並列アプリケーションと共有しない、MPI ベースのマルチスレッディング。	アプリケーション向けにカスタマイズされたスレッド・バインディング・ポリシーのほうがデフォルト OS ポリシーよりも効果的な場合があります。パフォーマンス・イベントを使用して、局所性とキャッシュ転送の可能性を最適化します。  独自の明示的なスレッド・アフィニティ・バインディング・ポリシーを使用するマルチスレッド・アプリケーションは、エンドユーザーまたは管理者に何らかの形式のオプトイン選択肢を提供し、許可された場合に導入すべきです。例えば、明示的なスレッド・アフィニティ・バインディング・ポリシーを導入する権限は、インストール後に権限が与えられてから有効化できるようにします。	アプリケーション向けにカスタマイズされたメモリー・アフィニティ・バインディング・ポリシーのほうがデフォルト OS ポリシーよりも効果的な場合があります。パフォーマンス・イベントを使用して、OS またはカスタムポリシーに関するローカルでないメモリアクセスの問題を診断します。

### 9.8.1 トポロジー共有リソースの列挙

マルチスレッド・ソフトウェアが OS のスケジューリング・ポリシーに基づいて実行される場合でも、カスタマイズされたリソース管理向けにアフィニティ API の使用を必要とする場合でも、共有プラットフォーム・リソースのトポロジーを理解することは重要です。CPUID によって提供される情報から、プラットフォーム上の論理プロセッサ (SMT)、プロセッサ・コア、物理プロセッサのプロセッサ・トポロジーを判断できます。これに関して、『インテル® 64 およ

び IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A<sub>2</sub> の第 8 章「Multiple-Processor Management」を参照してください。インテルからはホワイトペーパーとサンプルコードも提供されています。

## 9.8.2 NUMA (Non-Uniform Memory Access)

インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのインテル® Xeon® プロセッサが複数搭載されたプラットフォームでは、各物理プロセッサが独自のローカル・メモリー・コントローラーを備えているため、Non-Uniform Memory Access (NUMA) トポロジーがサポートされています。NUMA が提供するシステムメモリー帯域幅は、物理プロセッサ数に合わせて拡張されます。システムメモリーのレイテンシーは、メモリー・トランザクションが同じソケット内でローカルに発生するか、別のソケットからリモートで発生するかに応じて、非対称な動作となります。また、OS 固有の構造や実装による動作により、API レベルの複雑性が増すため、マルチスレッド・ソフトウェアでは NUMA 環境におけるメモリー割り当て/初期化に注意を払う必要があります。

一般に、レイテンシーに拘束されるワークロードでは、リモートよりもローカルのメモリー・トラフィックを維持することが望まれます。複数のスレッドがバッファを共有する場合、プログラマーは、NUMA システムでのメモリー割り当て/初期化に関する OS 固有の動作に注意する必要があります。

帯域幅に拘束されるワークロードでは、データ編成スレディング・モデルを採用すると利便性が高くなります。各ソケットで実行されるアプリケーション・スレッドをまとめて、ソケットごとでローカル・トラフィックを優先することで、物理プロセッサ数に合わせて帯域幅全体を拡張できるようになります。

ローカルとリモートの NUMA トラフィックを管理するプログラミング・インターフェイスを提供する OS 構造は、メモリー・アフィニティと呼ばれます。OS は、物理アドレス (システム RAM によって割り当てられる) とリニアアドレス (アプリケーション・ソフトウェアによってアクセスされる) のマッピングを管理しており、ページングは物理ページの動的再割り当てで異なるリニアアドレスへの動的な割り当てを可能にします。このような理由から、メモリー・アフィニティを適切に使用するには、OS 固有の知識が求められます。

アプリケーション・プログラミングを簡素化するため、OS では特定の API と物理/リニア・アドレス・マッピングを実装し、一部の状況で NUMA の特性を透過的に利用できます。一般的な手法の 1 つとして、アプリケーション・スレッドがリニアアドレス空間で物理メモリーページの最初のメモリー参照にアクセスするまで、OS による同ページの割り当てのコミットを遅延します。この場合、メモリー割り当て API がプログラムに戻る際にどのソケットがローカル・メモリー・トラフィックを処理するかは、アプリケーション・スレッドによるリニアアドレス空間でのメモリー・バッファ割り当てによって決まるとは限りません。ただし、このレベルの NUMA 透過性をサポートするメモリー割り当て API は、OS によって異なります。例えば、Linux\* 上では移植性に優れた C 言語 API 「malloc」がある程度の透過性を提供しますが、Windows\* 上では API 「VirtualAlloc」が同様の透過性を提供します。OS によっては、明示的な NUMA 情報を必要とするメモリー割り当て API も提供されており、リニアアドレスとローカル/リモート・メモリー・トラフィックとのマッピングが割り当て時に固定されます。

例 9-9 では、マルチスレッド・アプリケーションが最小限の処理で OS 固有の API を扱い、NUMA ハードウェア機能を利用しています。メモリー・バッファを初期化するこの並列アプローチは、NUMA システム上で各ワークスレッドにローカルのメモリー・トラフィックを維持するのに役立ちます。

例 9-9 OpenMP\* と NUMA を使用した並列メモリの初期化手法

```

#ifdef _LINUX // Linux* の malloc の実装では、最初に参照/ アクセスしたときに物理ページがコミットされます
    buf1 = (char *) malloc(DIM*(sizeof (double))+1024);
    buf2 = (char *) malloc(DIM*(sizeof (double))+1024);
    buf3 = (char *) malloc(DIM*(sizeof (double))+1024);
#endif
#ifdef windows
// Windows* の malloc の実装では割り当て時に物理ページがコミットされるので、
// 代わりに VirtualAlloc を使用します
    buf1 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
    buf2 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
    buf3 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
#endif
(続き)
    a = (double *) buf1;
    b = (double *) buf2;
    c = (double *) buf3;
#pragma omp parallel
{ // ループの各反復を OpenMP* スレッドで処理します
// OpenMP* のスレッド数はデフォルトか、もしくは環境変数で指定できます
#pragma omp for private(num)
// 各ループ反復を実行するため、プライベートなイテレーターを持つ異なる OpenMP* スレッドに割り当てられます
for(num=0;num<len;num++)
{ // 各スレッドは最初にアクセスした時に、メモリアドレスのサブセットを取得し、
// 物理ページは該当するスレッドのローカル・メモリー・コントローラーへマップされます
    a[num]=10.;
    b[num]=10.;
    c[num]=10.;
}
}
}

```

例 9-9 では、OpenMP\* によって生成されたワーカー・スレッドが終了した後に、メモリー・バッファーが解放されることに注意してください。ここでは、異なるアプリケーション・スレッド間で malloc と解放が繰り返し使用される問題を回避しています。ローカルメモリーがあるスレッドによって初期化され、その後別のスレッドによって解放された場合、OS では NUMA トポロジーに関連するリニアアドレス空間のメモリー・プールを追跡/再割り当てすることが困難になります。Linux\* では、別の API 「numa\_local\_alloc」が使用できます。

### 9.9 その他の共有リソースの最適化

マルチスレッド・アプリケーションのリソースの最適化は、プロセッサ・トポロジー階層内のキャッシュトポロジーと関連する実行リソースに依存します。プロセッサ・トポロジーと、ソフトウェアによるプロセッサ・トポロジーの識別アルゴリズムについては、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3A』の第 8 章で説明されています。

共有バスを備えたプラットフォームでは、バスシステムは、SMT レベルとプロセッサ・トポロジーのプロセッサ・コア・レベルで複数のエージェントによって共有されます。そのため、マルチスレッド・アプリケーションの設計は、同じバスリンクを共有する複数のプロセッサ・エージェント間で利用可能なバス帯域幅を公平に管理することから始めるべきです。これは、個々のアプリケーション・スレッドのデータ局所性を改善するか、2 つのスレッドが共有 2 次キャッシュを利用することで (そのような共有キャッシュトポロジーを利用可能な場合) 実現できます。

マルチスレッド・アプリケーションのビルディング・ブロックの最適化は通常、個々のスレッドから開始できます。第 3 章から第 10 章のガイドラインは、主にマルチスレッドの最適化に適用されます。

**チューニングの推奨事項 2:** 実行スループットを最大化するため、最初にシングルスレッド・コードを最適化します。

**チューニングの推奨事項 3:** 効率良いスレッディング・モデルを採用し、入手可能なツール (インテル® スレッディング・ビルディング・ブロック、インテル® パフォーマンス・ツールなど) を活用して、物理プロセッサやプロセッサ・コア数に応じた最適なプロセッサ・スケーリングを達成します。

### 9.9.1 HT テクノロジー最適化の可能性を拡大

インテル® マイクロアーキテクチャー開発コード名 Nehalem のハイパースレッディング (HT) テクノロジーは、前世代の HT テクノロジーの実装とは異なっています。幅広いアプリケーションで、マルチスレッド・ソフトウェアが HT テクノロジーを利用してシステム・スループットを高められる可能性が拡大しています。ここでは、ヒューリスティック的な推奨事項をいくつか紹介し、インテル® マイクロアーキテクチャー開発コード名 Nehalem で提供される HT テクノロジーの最適化の可能性について説明します。

第 2 章「インテル® 64 アーキテクチャーと IA-32 アーキテクチャー」では、HT テクノロジーにおけるマイクロアーキテクチャー上の機能強化について説明しています。これらの機能強化の多くは、複数のスレッド・コンテキストで使用可能なハードウェア・リソースの共有に関するマルチスレッド・ソフトウェアの基本要件を中心に行われています。

ソフトウェア・アルゴリズムやワークロード特性が異なると、複数の論理プロセッサ間で共有されるマイクロアーキテクチャー・リソースの要件から、パフォーマンス特性も異なります。表 9-3 では、HT テクノロジー向けのソフトウェア・チューニングで重要な役割を果たす各種マイクロアーキテクチャー・サブシステムを簡単に比較します。

表 9-3 HT テクノロジーのマイクロアーキテクチャー・リソースの比較

マイクロアーキテクチャー・サブシステム	インテル® マイクロアーキテクチャー開発コード名 Nehalem	Intel NetBurst® マイクロアーキテクチャー
	06_1AH	0F_02H, 0F_03H, 0F_04H, 0F_06H
発行ポート、実行ユニット	3 つの発行ポート (0, 1, 5) を ALU、SIMD、FP の各演算向けに分散	ポートのバランスが取れていない。高速 ALU、SIMD、FP で同じポート (ポート 1) を共有
バッファリング	適度なパイプラインの深さを確保しながら、ROB、RS、フィルバッファなどのエンタリを増加	バッファ・エンタリとパイプラインの深さとのバランスレベルが低い
分期予測とアライメントされないメモリアクセス	スペキュレーティブ・エグゼキューションが強化され、予測ミスの直後に再利用が可能。キャッシュ分割を効率的に処理	マイクロアーキテクチャー・ハザードが多く発生し、両方のスレッドでパイプラインがクリアされる
キャッシュ階層	大容量かつ効率的	回避すべきマイクロアーキテクチャー・ハザードが多い
メモリーと帯域幅	NUMA、DDR3 に対してソケットごとに 3 チャンネル、ソケットごとに最大 32GB/秒	SMP、FSB またはデュアル FSB、FSB ごとに最大 12.8GB/秒

演算量の多いワークロードの場合、Intel NetBurst® マイクロアーキテクチャーの HT テクノロジーでは、比較的高い CPI (連続した命令がリタイアするまでの平均サイクル) で実行されるスレッド・コンテキストが優先される傾向にあります。ハードウェア・レベルでは、高速 ALU、低速 ALU (負荷の高い整数演算)、SIMD、FP の各演算によってポート 1 が共有されるという、マイクロアーキテクチャーにおける発行ポートのインバランスがこの原因です。ソフトウェア・レベルでは、HT テクノロジーのメリットをもたらす高い CPI の要因として、レイテンシーの長い命令 (ポート 1)、一部の L2 ヒット、時折発生する分岐予測ミスなどがあります。ただし、Intel NetBurst® マイクロアーキテクチャーのパイプライン長は、ハードウェア内部の制約を増加させ、ソフトウェアによる HT テクノロジーの利用を制限する傾向があります。

表 9-3 に示したマイクロアーキテクチャーの強化点は、演算量の多いワークロードにおいてソフトウェア最適化の可能性を高めることが期待されます。ただし、演算量の多い 2 つのスレッドが同じ実行ユニット内で競合することが、データ分解スレッドモデルよりも機能分解スレッドモデルを選択する上での課題となる可能性があります。プログラマーによる最適なスレッド分解モデルの選択をサポートするには、インテル® マイクロアーキテクチャー開発コード名 Nehalem のほうが適しています。

大量のメモリーを必要とするワークロードは、完全に並列なメモリー・トラフィック (Stream の例に見られるような、システムメモリー帯域幅の飽和)、メモリー・レイテンシーに左右されるメモリー・トラフィック、演算操作といずれかの種類のメモリー・トラフィックとの各種組み合わせなど、幅広いパフォーマンス特性が見られます。

Intel NetBurst® マイクロアーキテクチャーの HT テクノロジーは、後者の 2 種類のワークロード特性の一部においてメリットが得られます。インテル® マイクロアーキテクチャー開発コード名 Nehalem の HT テクノロジーを利用すると、NUMA のサポート、効率的なリンクプロトコル、物理プロセッサ数に合わせて拡張可能なシステムメモリー帯域幅により、後者の 2 種類のワークロード特性における処理範囲を広げ、システム・スループットの向上を図ることが可能です。

キャッシュ階層中の一部のキャッシュレベルは、複数の論理プロセッサによって共有されることがあります。キャッシュ階層の利用は、ソフトウェアによってメモリー・トラフィックを効率化し、システムメモリー帯域幅の飽和を回避する重要な手段です。キャッシュ・ブロッキング手法が採用されたマルチスレッド・アプリケーションでは、HT テクノロジーを利用するためにターゲット・キャッシュ・レベルの分割が必要となることがあります。また、同じ L1 キャッシュと L2 キャッシュを共有する 2 つの論理プロセッサや、L3 キャッシュを共有する論理プロセッサは、それぞれに関連するトポロジーに応じた共有リソース管理を必要とすることがあります。プロセッサ・トポロジーの列挙およびキャッシュトポロジーの列挙とそのサンプルコードについては、ホワイトペーパーが公開されています (第 1 章の最後にある「関連情報」をご覧ください)。

## 10.1 はじめに

この章では、64 ビット・モードで動作するように作成されたアプリケーション・ソフトウェア向けのコーディング・ガイドラインについて説明します。コーディングの推奨事項の一部は、第 3 章で触れています。この章のガイドラインは、第 3 章から第 9 章で説明したコーディング・ガイドラインへの補遺であると言えます。

互換モードまたはレガシーの非 64 ビット・モードで実行されるソフトウェアは、第 3 章から第 9 章で説明するガイドラインに従わなければなりません。

## 10.2 64 ビット・モードに影響するコーディング規則

### 10.2.1 データサイズが 32 ビットの場合はレガシーの 32 ビット命令を使用

64 ビット・モードでは、アプリケーションは 16 個の汎用 64 ビット・レジスターを利用できます。アプリケーションのデータサイズが 32 ビットである場合、64 ビット・レジスターや 64 ビット演算を使用する必要はありません。

ほとんどの命令では、デフォルトのオペランドサイズは 32 ビットです。このような命令では、上位 32 ビットをすべてゼロにします。例えば、レジスターをゼロにする場合、以下の 2 つの命令は同じように動作しますが、32 ビット版では 1 つの命令バイトのみ保存されます。

32 ビット版

`xor eax, eax`; 下位 32 ビットが xor され、上位 32 ビットにはゼロを設定

64 ビット版

`xor rax, rax`; 64 ビットすべてに xor を実行

この最適化は、8 つの汎用レジスター (EAX, ECX, EBX, EDX, ESP, EBP, ESI, EDI) に適用されます。レジスター R9 ~ R15 のデータをアクセスするには、REX プリフィクスが必要です。この場合、32 ビット形式を使用してもコードサイズは減少しません。

**アセンブリー/コンパイラー・コーディング規則 63 (影響 H、一般性 M):** 64 ビット・モードでは、64 ビット・データや追加レジスターへのアクセスに 64 ビット版の命令が必要な場合を除き、32 ビット版の命令を使うことでコードサイズを削減します。

### 10.2.2 追加のレジスターを使用してレジスターへの負荷を削減

64 ビット・モードでは、アプリケーションは 8 つの追加 64 ビット汎用レジスターと 8 つの追加 XMM レジスターを利用できます。追加されたレジスターにアクセスするには、1 バイトの REX プリフィクスが必要です。8 つの追加レジスターを使用することで、コンパイラーは値をスタックに退避させる必要がなくなります。

REX プリフィクスによってコードサイズが増加すると、キャッシュミスが増える可能性があるため注意が必要です。これは、追加のレジスターを使ってデータにアクセスするメリットに悪影響をもたらす恐れがあります。アルゴリズムで使用するレジスターが 8 つで十分なら、REX プリフィクスが必要なレジスターは使用してはいけません。これにより、コードサイズが小さくなります。



**アセンブリー/コンパイラー・コーディング規則 64 (影響 H、一般性 MH):** レジスターへの負荷削減に必要であれば、整数コードには 8 つの追加汎用レジスターと、浮動小数点および SIMD コードには 8 つの追加 XMM レジスターを使用します。

### 10.2.3 64 ビット値同士の乗算を有効活用

64 ビット・オペランド同士の整数乗算は、128 ビット幅の結果を生成する可能性があります。128 ビットの結果の上位 64 ビットは、下位 64 ビットよりも計算結果を生成するのにさらに数サイクル分必要な場合があります。128 ビットよりも幅が広い整数の加算の依存関係チェーンでは、最適なソフトウェア・パイプライン処理を行う上で、乗算結果の上位 64 ビットへのアクセスが、下位 64 ビットと比べて遅くなります。

コンパイラーは、コンパイル時に乗算結果が 64 ビットを超えないことを判断できる場合、64 ビットの結果を生じる乗算命令を生成すべきです。乗算結果が 64 ビットよりも小さくなることをコンパイラーやアセンブリー・プログラマーが判断できない場合、128 ビットの結果を生成する乗算が必要です。

**アセンブリー/コンパイラー・コーディング規則 65 (影響 ML、一般性 M):** 128 ビットの結果を生成する乗算よりも、64 ビットの結果を生成する 64 ビット同士の整数乗算を優先します。

**アセンブリー/コンパイラー・コーディング規則 66 (影響 ML、一般性 M):** 128 ビット乗算の下位 64 ビットの結果にアクセスした後に、上位 64 ビットの結果にアクセスしないようにします。

インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge では、128 ビットの乗算の下位 64 ビットの結果は 3 サイクル後に使用できるのに対し、上位 64 ビットの結果は下位 64 ビットの結果から 1 サイクル遅れて使用可能になります。これにより、大きな整数の整数乗算および除算の計算速度を向上できます。

### 10.2.4 128 ビット整数除算を 128 ビット乗算で置き換え

現在のコンパイラーは、定数除数を含む高級言語コードの整数除算数式において、IDIV/DIV 命令を IMUL/MUL 命令に置換してアセンブリー・シーケンスを生成できます。通常、コンパイラーが除数値を置換するのは、この除数値が 32 ビットの範囲内であり、かつコンパイル時に既知である場合です。除数値がコンパイル時に不明である場合、または除数が 32 ビット表現よりも大きい場合は、DIV または IDIV が生成されます。

128 ビットの被除数を含む DIV 命令のレイテンシーはかなり長くなります。被除数値が 64 ビットよりも大きい場合のレイテンシーは 70 ~ 90 サイクルとなります。

整数除算を 128 ビットの整数乗算に変換する際にコンパイラーが利用する基本的な手法は、モジュラー計算の合同原則に基づきます。これは大きな除数値の処理まで簡単に拡張でき、128 ビットの高速な IMUL/MUL 操作を利用することができます。

整数式:

$$\text{被除数} = Q * \text{除数} + R$$

または

$$Q = \text{下限}(\text{被除数}/\text{除数}), R = \text{被除数} - Q * \text{除数}$$

実数ドメインに変換:

$$\text{下限}(\text{被除数}/\text{除数}) = \text{被除数}/\text{除数} - R/\text{除数}, \text{は以下と等しくなります}$$

$$Q * C2 = \text{被除数} * (C2 / \text{除数}) + R * C2 / \text{除数}$$

最後の項の丸めを制御するため、 $C2 = 2^N$  を選択できます。この場合、次のようになります。

$$Q = ((\text{被除数} * (C2 / \text{除数})) \gg N) + ((R * C2 / \text{除数}) \gg N)$$

「除数」がコンパイル時に既知である場合、(C2/ 除数) は合同定数  $Cx = \text{上限}(\text{C2/ 除数})$  として事前に計算でき、商は整数倍数によって算出した後にシフトできます。

$$Q = (\text{被除数} * Cx) \gg N;$$

$$R = \text{被除数} - ((\text{被除数} * Cx) \gg N) * \text{除数}$$

128 ビットの IDIV/DIV 命令は、数値例外の発生を避けるため、除数、商、剰余の範囲を 64 ビット内に制限します。これは、この 3 つの値のいずれかが 64 ビットの上限に近い場合、および被除数値が 128 ビットの上限に近い場合に問題となります。

この問題は、大きなシフトカウント N を選択し、(被除数 \* Cx) 演算を 128 ビット範囲から次に計算効率の良い範囲に拡張することで解決できます。例えば、(被除数 \* Cx) が 128 ビットよりも大きく、N が 63 ビットよりも大きい場合、192 ビットの乗算を実装することなく、128 ビットの MUL を使用して、192 ビットの結果の計算ビット 191:64 を利用できます。

合同定数 Cx を選択する方法は以下のとおりです。

- Y 被除数の範囲が 64 ビット内の場合:  $N_{\min} = \text{BSR}(\text{除数}) + 63$
- Y 除数の範囲と比べたときに商/剰余の動的な範囲が異なる場合、商/剰余が効率的に算出できるよう、適切に N を生成します。

除数が  $10^{16}$  で、符号なしの被除数が 64 ビットの範囲に近い商/剰余の計算について考えてみます。例 10-1 に、「MUL r64」命令を使用した、64 ビットの除数による 64 ビットの被除数の処理を示します。

例 10-1 64 ビットの除数による 64 ビットの商および剰余の計算

```

_Cx10to16: ; シフトカウント N=117 の合同定数 10^16
    DD 0c44de15ch ; floor((2^117 / 10^16) + 1)
    DD 0e69594beh ; 128 ビットの倍数で減るように Cx の長さを最適化
_tento16: ; 10^16
    DD 6fc10000h
    DD 002386f2h

    mov r9, qword ptr [rcx] ; 64 ビットの被除数をロード
    mov rax, r9
    mov rsi, _Cx10to16 ; シフトカウント N=117 の合同定数 10^16
    mul [rsi] ; 128 ビット乗算
    mov r10, qword ptr 8[rsi] ; 除数 10^16 のロード
    shr rdx, 53; ;
    mov r8, rdx

    mov rax, r8
    mul r10 ; 128 ビット乗算
    sub r9, rax; ;
    jae remain
    sub r8, 1 ; 丸めによって 1 つずれているかもしれない
    mov rax, r8
    mul r10 ; 128 ビット乗算
    sub r9, rax; ;
remain:
    mov rdx, r8 ; 商
    mov rax, r9 ; 余り
    
```

例 10-2 に、同様の手法を使用した 64 ビットの除数による 128 ビットの被除数の処理を示します。

例 10-2 64 ビットの除数による 128 ビットの商および剰余の計算

```

mov rax, qword ptr [rcx] ; メモリーから 128 ビット被除数の 63:0 ビットをロード
mov rsi, _Cx10to16 ; シフトカウント N=117 の合同定数 10^16
mov r9, qword ptr [rsi] ; 合同定数をロード
mul r9 ; 128 ビット乗算
xor r11, r11 ; 積算器のクリア
mov rax, qword ptr 8[rcx] ; 128 ビット被除数の 127:64 ビットをロード
shr rdx, 53 ; ;
mov r10, rdx ; 192 ビット結果の 127:64 ビットを初期化
mul r9 ; 191:128 ビットを積算
add rax, r10 ; ;
adc rdx, r11 ; ;
shr rax, 53 ; ;
shl rdx, 11 ; ;
or rdx, rax ; ;
mov r8, qword ptr 8[rsi] ; 除数 10^16 をロード
mov r9, rdx ; おおよその商、1 つずれているかもしれない
mov rax, r8
mul r9 ; 商 * 除数 > 被除数 ?
sub rdx, qword ptr 8[rcx] ;
sbb rax, qword ptr [rcx] ;

    jb remain
sub r9, 1 ; これは丸めによって 1 つずれているかもしれない
mov rax, r8 ; 除数 10^16 の復元
mul r9 ; 最終的な商 * 除数
sub rax, qword ptr [rcx] ;
sbb rdx, qword ptr 8[rcx] ;
remain:
    mov rdx, r9 ; 商
    neg rax ; 余り
    
```

例 10-1 および例 10-2 に示す手法により、128 ビットの被除数の剰余/商の計算速度を、32 ビットの整数除算のコストと同じかそれ以下に向上させることができます。

上記の手法を 64 ビットよりも大きい除数の処理に拡張するのは比較的容易です。シフトカウント  $N > 128$  ビットを選択することは、考慮に値する 1 つの最適化です。これによって、(被除数 \* Cx) の関連する上位ビットを計算するために必要となる 128 ビットの MUL の数を減らすことができます。

### 10.2.5 完全な 64 ビットへの符号拡張

Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサは、64 ビット・モードで単一のマイクロオペレーション (μop) により 64 ビットへの符号拡張を行うことができます。64 ビット・モードでは、デスティネーションが 32 ビットの場合、上位 32 ビットをゼロにする必要があります。

上位 32 ビットをゼロにするには、追加のマイクロオペレーション (μop) が必要になり、64 ビットへの符号拡張よりも効率が悪くなります。64 ビットに符号拡張すると、命令が 1 バイト長くなりますが、Intel NetBurst® マイクロアーキテクチャー・ベースのプロセッサでは、トレースキャッシュにストアしなければならないマイクロオペレーション (μop) が減少するので、パフォーマンスが向上します。

例えば、1 バイトを ESI に符号拡張するには、以下を使用します。

```
movsx rsi, BYTE PTR[rax]
```

以下は使用すべきではありません。

```
movsx esi, BYTE PTR[rax]
```

次の命令で 32 ビット形式の ESI レジスターを使用する場合でも、結果は同じです。この最適化は、予期しない依存関係を解消するのにも利用できます。例えば、プログラムが 16 ビット値をレジスターに書き込んでから、8 ビット値をそのレジスターに書き込む場合、デスティネーションのビット 15:8 が不要であれば、符号拡張を伴う書き込みを利用します（利用可能であれば）。

次に例を示します。

```
mov r8w, r9w; ビット 63:15 をマージして保存する場合
mov r8b, r10b; ビット 63:8 をマージして保存する場合
```

これは以下に置き換えられます。

```
movsx r8, r9w ; ビット 63:8 を保存する必要がない場合
movsx r8, r10b ; ビット 63:8 を保存する必要がない場合
```

上記の例では、R8W への移動および R8B への移動は、マージによってレジスター内の残りのビットを保持する必要があります。「MOV R8W, R9W」と「MOV R8B, R10B」の間には、R8 に対する暗黙的な依存関係があります。MOVSX を使用すると、実際の依存関係が解消され、出力の依存関係のみが残ります。この依存関係は、プロセッサがリネームによって排除できます。

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサでは、上位 32 ビットをゼロにすることは、64 ビットへの符号拡張よりも高速です。インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサでは、上位ビットをゼロにすること、または符号拡張することは単一のマイクロオペレーション（`pop`）で行われます。

## 10.3 64 ビット・モード向けの代替えコーディング規則

### 10.3.1 64 ビット演算結果では 2 つの 32 ビット・レジスターの代わりに 64 ビット・レジスターを使用

レガシーの 32 ビット・モードでは、拡張精度整数演算（64 ビット演算など）をサポートする機能が提供されています。ただし、64 ビット・モードでは、64 ビット演算をネイティブにサポートしています。64 ビットの整数が必要な場合は、64 ビット形式の演算命令を使用します。

32 ビットのレガシーモードでは、32 ビット同士の整数乗算から 64 ビットの結果を得るのに 3 つのレジスターが必要となり、結果は、EDX:EAX のペアにストアされます。64 ビット・モードの命令が利用可能な場合、64 ビットの結果が必要なときに 32 ビット命令を使用するのは最適な実装とはいえません。拡張レジスターを使用します。

例えば、以下のコードシーケンスでは、64 ビット・レジスターに符号拡張された 32 ビット値をロードして、乗算を行います。

```
movsx rax, DWORD PTR[x]
movsx rcx, DWORD PTR[y]
imul rax, rcx
```

上記の 64 ビット版の方が、以下の 32 ビット版よりも効率的です。

```
mov eax, DWORD PTR[x]
mov ecx, DWORD PTR[y]
imul ecx
```

上記の 32 ビット版では、EAX がソースである必要があります。結果は、単一の 64 ビット・レジスターではなく、EDX:EAX のペアに格納されます。

**アセンブリ/コンパイラ・コーディング規則 67 (影響 ML、一般性 M):** 64 ビットの結果が必要な 32 ビットの整数乗算には、64 ビット版の乗算を使用します。

32 ビットのレガシーモードで 2 つの 64 ビット値を加算するには、add 命令の後に adc 命令を続けて使用します。例えば、2 つの 64 ビット変数 (X および Y) を加算するには、以下の 4 つの命令を使用できます。

```
mov eax, DWORD PTR[X]
mov edx, DWORD PTR[X+4]
add eax, DWORD PTR[Y]
adc edx, DWORD PTR[Y+4]
```

結果は、2 つのレジスターペア EDX:EAX に格納されます。

64 ビット・モードの場合、上記のシーケンスを以下のように短縮できます。

```
mov rax, QWORD PTR[X]
add rax, QWORD PTR[Y]
```

結果は RAX に格納されます。必要なレジスターは、2 つではなく 1 つです。

**アセンブリ/コンパイラ・コーディング規則 68 (影響 ML、一般性 M):** 64 ビットに加算には、64 ビット版の加算を使用します。

## 10.3.2 CVTSI2SS と CVTSI2SD

インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサ以降では、CVTSI2SS 命令と CVTSI2SD 命令のレイテンシーとスループットが Intel NetBurst® マイクロアーキテクチャーよりも大幅に向上しています。これは、64 ビット版にも 32 ビット版にも適用されます。

## 10.3.3 ソフトウェア・プリフェッチの使用

ソフトウェア・プリフェッチを使用せず、ハードウェア・プリフェッチを利用できるようにデータ・アクセス・パターンを編成する手法を検討する際は、第 3 章と第 7 章の推奨事項に従うことを推奨します。

**アセンブリ/コンパイラ・コーディング規則 69 (影響 L、一般性 L):** ソフトウェア・プリフェッチ命令が必要な場合は、インテル® SSE 命令セットで提供されているプリフェッチ命令を使用します。

インテル® アドバンスド・ベクトル・エクステンション (インテル® AVX) は、インテル® アーキテクチャーへの重要な拡張機能です。これまでの世代の 128 ビットのインテル® SSE ベクトル命令の機能を拡張し、256 ビット操作をサポートするようにベクトルレジスター幅を拡大しています。インテル® AVX 命令セット・アーキテクチャー (ISA) 拡張は浮動小数点命令に注目します。一部の 256 ビット整数ベクトルは、浮動小数点から整数および整数から浮動小数点への変換を行うことでサポートされます。

インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge におけるインテル® AVX 命令は、ほとんどが 256 ビット・ハードウェアで実装されています。したがって、それぞれのコアは 256 ビット浮動小数点加算と乗算ユニットを備えています。除算と平方根ユニットは 256 ビットへ強化されていません。それゆえ、インテル® AVX 命令は、128 ビット・ハードウェアを使用した 2 つのステップで 256 ビット操作を完了します。

これまでの世代のインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の大部分は 2 オペランド構文です (オペランドの 1 つがソースとデスティネーションとして機能)。インテル® AVX 命令は、ベクトル長をエンコードするビット・フィールドを含む VEX プリフィクス付きでエンコードされ、3 オペランド構文をサポートします。ほとんどの命令は 2 つのソースと 1 つのデスティネーションを持ちます。VBLENDVPS や VBLENDVPD のような 4 オペランド命令もあります。追加されたオペランドにより非破壊ソース (NDS) が可能となり、MOVAPS 命令を使用してレジスターを複製する必要がなくなります。インテル® MMX® 命令を除く、ほとんどすべてのレガシーインテル® SSE 128 ビット命令には、3 オペランド構文をサポートする等価なインテル® AVX 命令があります。256 ビットのインテル® AVX 命令は 3 オペランド (いくつかは 4 オペランド構文) を採用しています。

256 ビット・ベクトル・レジスター YMM は、128 ビットの XMM レジスターを 256 ビットに拡張したものです。そのため YMM の下位 128 ビットは、レガシー XMM レジスターにエイリアスされます。256 ビットのインテル® AVX 命令は 256 ビットの結果を YMM に書き込むのに対し、128 ビットのインテル® AVX 命令は 128 ビットの結果を XMM に書き込んで YMM の上位 128 ビットにはゼロを設定します。64 ビット・モードでは 16 個のベクトルレジスターが利用できます。64 ビット・モード以外 (32 ビット・モード) では、下位 8 個のベクトルレジスターのみが利用できます。

ソフトウェアは、レガシーインテル® SSE コード、128 ビットのインテル® AVX コード、そして 256 ビットのインテル® AVX コードをどのような組み合わせでも使用できます。この節では、ベクトル長が混在するコードモジュールがインテル® SSE とインテル® AVX コード間の移行の遅延を被ることなく最適なパフォーマンスを達成するガイドラインをカバーしています。128 ビットのインテル® AVX コードと 256 ビットのインテル® AVX コードの混在では移行の遅延は発生しません。

メモリーに保存されるインテル® AVX ベクトルの最適なメモリー・アライメントは、32 バイトです。256 ビットのインテル® AVX 命令のデータ移動のいくつかは 32 バイト・アライメントを強制します。メモリーオペランドが適切にアライメントされていないと #GP フォルトが発生します。ほとんどの 256 ビットのインテル® AVX 命令はアドレス・アライメントを要求しません。これらの命令では一般にロードと計算操作が組み合わされており、命令はどのようなアライメントのメモリーアドレスも使用できます。

最高のパフォーマンスを達成するには、ソフトウェアは可能な限り 32 バイトにアライメントされたロードとストアを使用すべきです。

インテル® AVX 命令とレガシーインテル® SSE 命令を使用する上での主な違いを表 12-1 に示します。

表 12-1 256 ビットのインテル® AVX、128 ビットのインテル® AVX、およびレガシーインテル® SSE 拡張の機能比較

機能	256 ビットのインテル® AVX	128 ビットのインテル® AVX	レガシーインテル® SSE - インテル® AES-NI
機能範囲	浮動小数点演算、データ移動	レガシー SIMD ISA と一致  (インテル® MMX® 命令を除く)	128 ビット FP と整数 SIMD ISA
レジスターオペランド	YMM	XMM	XMM
オペランド・シンタックス	4 つまで、ソースを非破壊	4 つまで、ソースを非破壊	2 オペランド・シンタックス、ソースを破壊
メモリー・アライメント	ロード操作のセマンティクスでアライメントは不要	ロード操作のセマンティクスでアライメントは不要	常に 16B アライメントを強制
アライメントされた移動命令	32 バイト・アライメント	16 バイト・アライメント	16 バイト・アライメント
非破壊ソースオペランド	はい	はい	いいえ
レジスターストールの扱い	ビット 255:0 を更新	127:0 を更新、128 以上はゼロビット	127:0 を更新、128 以上は変更しない
組込み関数サポート	新しい 256 ビット・データ型  プロモートされた機能のための <code>_mm256</code> プレフィクス  新しい機能のための新しい組込み関数	既存のデータ型  既存の機能のための同じプロトタイプ の継続  新しい VEX-128 の機能には “ <code>_mm</code> ” プリフィクスを使用	ベースラインのデータ型とプロトタイプの定義
128 ビット・レーン	ほとんどの 256 ビット操作に適用	1 つの 128 ビット・レーン	1 つの 128 ビット・レーン
混在したコードの扱い	移行のペナルティーを避けるため <code>VZEROUPPER</code> を使用	移行のペナルティーなし	256 ビットのインテル® AVX コード実行後の移行のペナルティー

## 12.1 インテル® AVX 組込み関数のコーディング

256 ビットのインテル® AVX 命令では新しい組込み関数が定義されています。特に、既存のインテル® SSE の機能を 256 ビット・ベクトルへ拡張した 256 ビットのインテル® AVX 命令では、プロトタイプ化された “\_mm” プリフィックスの代わりに “\_mm256” プリフィックスによって、256 ビット操作向けに定義された新しいデータ型を使用します。256 ビットのインテル® AVX 命令の新しい機能には、全く新しいプロトタイプが定義されています。

レガシー SIMD ISA を継続する 128 ビットのインテル® AVX 命令には、これまでと同じプロトタイプを使用します。256 ビットと 128 ビットのインテル® AVX 命令の新しい機能は、それぞれ “\_mm256” と “\_mm” プリフィックス付きのプロトタイプです。

そのため、組込み関数で記述されたレガシー SIMD コードを 256 ビットのインテル® AVX へ移植する労力はそれほど掛かりません。

次のガイドラインは、インテル® SSE コードのシーケンスからインテル® AVX へ簡単に組込み関数を書き換える方法を示しています。

インテル® AVX:

- Y 静的および動的に割り当てられているバッファーを 32 バイトにアライメントします。
- Y バッファーサイズは倍にする必要があるでしょう。
- Y 「\_mm\_組込み関数名」のプリフィックスを \_\_mm256 に変更します。
- Y 変数のデータ型名を \_\_m128 から \_\_m256 へ変更します。
- Y ループの反復回数を半分にします (または倍のストライド長に)。

以下に示す例 (デカルト座標変換) は、インテル® AVX 命令形式、32 バイト YMM レジスター、32 バイト・データ境界での静的および動的メモリーの割り当て、および YMM レジスター内の 8 つの浮動小数点を表現する C のデータ型を使用しています。



例 12-1 組み込み関数を使用したデカルト座標変換

<pre>// インテル® SSE 組み込み関数を使用 #include "wmmintrin.h" int main() { int len = 3200; // 16 バイト境界で動的にメモリーを割り当て // float* pInVector = (float*) _mm_malloc(len*sizeof(float), 16); float* pOutVector = (float*) _mm_malloc(len*sizeof(float), 16); //データの初期化 for(int i=0; i&lt;len; i++) pInVector[i] = 1; float cos_teta = 0.8660254037; float sin_teta = 0.5; // 4 つの float を 16 バイトのアライメントで静的にメモリー割 り当て __declspec(align(16)) float cos_sin_teta_vec[4] = {cos_teta, sin_teta, cos_teta, sin_teta}; __declspec(align(16)) float sin_cos_teta_vec[4] = {sin_teta, cos_teta, sin_teta, cos_teta}; // __m128 データ型は、xmm レジスターの // 4 つの float 要素を表します __m128 Xmm_cos_sin = _mm_load_ps(cos_sin_teta_vec); // インテル® SSE の 128 ビットのパックド単精度ロード __m128 Xmm_sin_cos = _mm_load_ps(sin_cos_teta_vec); __m128 Xmm0, Xmm1, Xmm2, Xmm3 // 2 回分アンロールされたループで 8 つの要素を処理 for(int i=0; i&lt;len; i+=8) {     Xmm0 = _mm_load_ps(pInVector+i);     Xmm1 = _mm_moveldup_ps(Xmm0);     Xmm2 = _mm_movehdup_ps(Xmm0);     Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);     Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);     Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);     _mm_store_ps(pOutVector + i, Xmm3);     Xmm0 = _mm_load_ps(pInVector+i+4);     Xmm1 = _mm_moveldup_ps(Xmm0);     Xmm2 = _mm_movehdup_ps(Xmm0);     Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);     Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);     Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);     _mm_store_ps(pOutVector+i+4, Xmm3); } _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>	<pre>// インテル® AVX 組み込み関数を使用 #include "immintrin.h" int main() { int len = 3200; // 16 バイト境界で動的にメモリーを割り当て // float* pInVector = (float*) _mm_malloc(len*sizeof(float), 32); float* pOutVector = (float*) _mm_malloc(len*sizeof(float), 32); //データの初期化 for(int i=0; i&lt;len; i++) pInVector[i] = 1; float cos_teta = 0.8660254037; float sin_teta = 0.5; // 8 つの float を 32 バイトのアライメントで静的にメモリー割 り当て __declspec(align(32)) float cos_sin_teta_vec[8] = {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta}; __declspec(align(32)) float sin_cos_teta_vec[8] = {sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta }; // __m256 データ型は 8 つの float 要素を格納 __m256 Ymm_cos_sin = _mm256_load_ps(cos_sin_teta_vec); // インテル® AVX の 256 ビットのパックド単精度ロード __m256 Ymm_sin_cos = _mm256_load_ps(sin_cos_teta_vec); __m256 Ymm0, Ymm1, Ymm2, Ymm3; // 2 回分アンロールされたループで 8 つの要素を処理 for(int i=0; i&lt;len; i+=16) {     Ymm0 = _mm256_load_ps(pInVector+i);     Ymm1 = _mm256_moveldup_ps(Ymm0);     Ymm2 = _mm256_movehdup_ps(Ymm0);     Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);     Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);     Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);     _mm256_store_ps(pOutVector + i, Ymm3);     Ymm0 = _mm256_load_ps(pInVector+i+8);     Ymm1 = _mm256_moveldup_ps(Ymm0);     Ymm2 = _mm256_movehdup_ps(Ymm0);     Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);     Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);     Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);     _mm256_store_ps(pOutVector+i+8, Ymm3); } _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>
---	--

## 12.1.1 インテル® AVX アセンブリのコーディング

組込み関数の移植のガイドラインと同様に、アセンブリで記述されたコードを移植するガイドラインを以下に示します。

- Y 静的および動的に割り当てられているバッファを 32 バイトにアライメントします。
- Y 必要であれば補足のバッファサイズを 2 倍にします。
- Y 命令ニーモニックの前に “v” プリフィクスを追加します。
- Y レジスター名を xmm から ymm へ変更します。
- Y インテル® AVX 演算命令にデスティネーションレジスターを追加します。
- Y ループの反復回数を半分にします (またはストライド長を倍にします)。

例 12-2 アセンブリを使用したデカルト座標変換

<pre>// インテル® SSE アセンブリを使用 int main() {     int len = 3200;     // 16 バイト境界で動的にメモリーを割り当て     //     float* pInVector = (float*)     _mm_malloc(len*sizeof(float), 16);     float* pOutVector = (float*)     _mm_malloc(len*sizeof(float), 16);     //データの初期化     for(int i=0; i&lt;len; i++)     pInVector[i] = 1;     // 4 つの float を 16 バイトのアライメントで静的にメモリー割     // り当て      float cos_teta = 0.8660254037;     float sin_teta = 0.5;     __declspec(align(16)) float cos_sin_teta_vec[4] =     {cos_teta, sin_teta, cos_teta, sin_teta};     __declspec(align(16)) float sin_cos_teta_vec[4] =     {sin_teta, cos_teta, sin_teta, cos_teta};     // 2 回分アンロールされたループで 8 つの要素を処理     __asm     {         mov eax, pInVector         mov ebx, pOutVector         // ymm レジスターに 16 バイトをロード         movups xmm3,         xmmword ptr[cos_sin_teta_vec]         movups xmm4,         xmmword ptr[sin_cos_teta_vec]         mov edx, len         xor ecx, ecx         loop1:         movsldup xmm0, [eax+ecx]         movshdup xmm1, [eax+ecx]         // 例: mulps は 2 オペランド         mulps xmm0, xmm3         mulps xmm1, xmm4         addsubps xmm0, xmm1</pre>	<pre>// インテル® AVX アセンブリを使用 int main() {     int len = 3200;     // 32 バイト境界で動的にメモリーを割り当て     //     float* pInVector = (float*)     _mm_malloc(len*sizeof(float), 32);     float* pOutVector = (float*)     _mm_malloc(len*sizeof(float), 32);     //データの初期化     for(int i=0; i&lt;len; i++)     pInVector[i] = 1;     // 8 つの float を 32 バイト境界で静的にメモリー割り当て      float cos_teta = 0.8660254037;     float sin_teta = 0.5;     __declspec(align(32)) float cos_sin_teta_vec[8] =     {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta,     sin_teta, cos_teta, sin_teta};     __declspec(align(32)) float sin_cos_teta_vec[8] =     {sin_teta, cos_teta, sin_teta, cos_teta, sin_teta,     cos_teta, sin_teta, cos_teta};     // 2 回分アンロールされたループで 16 つの要素を処理     __asm     {         mov eax, pInVector         mov ebx, pOutVector         // ymm レジスターに 32 バイトをロード         vmovups ymm3,         ymmword ptr[cos_sin_teta_vec]         vmovups ymm4,         ymmword ptr[sin_cos_teta_vec]         mov edx, len         xor ecx, ecx         loop1:         vmovsldup ymm0, [eax+ecx]         vmovshdup ymm1, [eax+ecx]         // 例: vmulps は 3 オペランド         vmulps ymm0, ymm0, ymm3         vmulps ymm1, ymm1, ymm4</pre>
---	--

<pre>// xmm レジスターから 16v バイトのストア movaps [ebx+ecx], xmm0 movsldup xmm0, [eax+ecx+16] movshdup xmm1, [eax+ecx+16] mulps xmm0, xmm3 mulps xmm1, xmm4 addsubps xmm0, xmm1 // 前のストアからの 16 バイトのオフセット movaps [ebx+ecx+16], xmm0 // このループで処理された 32 バイト // (コードは 2 回アンロール) add ecx, 32 cmp ecx, edx j1 loop1 } _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>	<pre>vaddsubps ymm0, ymm0, ymm1 // ymm レジスターから 32 バイトのストア vmovaps [ebx+ecx], ymm0 vmovsldup ymm0, [eax+ecx+32] vmovshdup ymm1, [eax+ecx+32] vmulps ymm0, ymm0, ymm3 vmulps ymm1, ymm1, ymm4 vaddsubps ymm0, ymm0, ymm1 // 前のストアからの 32 バイトのオフセット vmovaps [ebx+ecx+32], ymm0 // このループで処理された 64 バイト // (コードは 2 回アンロール) add ecx, 64 cmp ecx, edx j1 loop1 } _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>
---	--

## 12.2 非破壊ソース (NDS)

ほとんどのインテル® AVX 命令は 3 つのオペランドで構成されています。一般的な命令は 2 つのソースと 1 つのデスティネーションを持ち、ソースオペランドは命令によって変更されることはありません。この節では NDS の機能を使用して、レジスターの複製を回避し、命令数と  $\mu\text{op}$  数を減らしてパフォーマンスを向上する方法を説明します。この例では、インテル® AVX のコードはインテル® SSE のコードよりも 2 倍高速です。

次の例はベクトル化された多項式  $A^3 + A^2 + A$  の計算を行っています。多項式計算の疑似コードを示します。

```
While (i<len)
{
    B[i] := A[i]3 + A[i]2 + A[i]
    i++
}
```

例 12-3 の左側のリストはインテル® SSE アセンブリーを使用したベクトル化の実装を示しています。このコードでは、A は追加のロード命令によってメモリーからレジスターへコピーされ、A2 はレジスター間の割り当てによってコピーされています。コードは 4 つの要素を処理するため  $10 \mu\text{op}$  を要します。

中央の例では 128 ビットのインテル® AVX を使用し NDS の利点を活用しています。追加のロードとレジスターのコピーは排除されています。このコードは 4 つの要素を  $8 \mu\text{op}$  で処理し、左のベースラインよりも 30% ほど高速化されています。

右側の例は 256 ビットのインテル® AVX 命令を使用しています。8 つの要素を処理するため  $8 \mu\text{op}$  を要します。NDS の機能と 2 倍のベクトル幅により、ベースラインよりも 2 倍以上スピードアップされています。

例 12-3 直接多項式の計算

インテル® SSE コード (ベースライン)	128 ビットのインテル® AVX	256 ビットのインテル® AVX
<pre>float* pA = InputBuffer; float* pB = OutputBuffer; int len = miBufferWidth-4; __asm { mov rax, pA mov rbx, pB movsxd r8, len loop1: // ロード A movupsxmm0, [rax+r8*4] // コピー A movupsxmm1, [rax+r8*4] //A^2 mulpsxmm1, xmm1 //コピー A^2 movupsxmm2, xmm1 //A^3 mulps xmm2, xmm0 //A + A^2 addps xmm0, xmm1 //A + A^2 + A^3 addps xmm0, xmm2 // 結果をストア movups[rbx+r8*4], xmm0 sub r8, 4 jge loop1 }</pre>	<pre>float* pA = InputBuffer1; float* pB = OutputBuffer1; int len = miBufferWidth-4; __asm { mov rax, pA mov rbx, pB movsxd r8, len loop1: // ロード A vmovups xmm0, [rax+r8*4] //A^2 vmulps xmm1, xmm0, xmm0 //A^3 vmulps xmm2, xmm1, xmm0 //A+A^2 vaddps xmm0, xmm0, xmm1 //A+A^2+A^3 vaddps xmm0, xmm0, xmm2 // 結果をストア vmovups[rbx+r8*4], xmm0 sub r8, 4 jge loop1 }</pre>	<pre>float* pA = InputBuffer1; float* pB = OutputBuffer1; int len = miBufferWidth-8; __asm { mov rax, pA mov rbx, pB movsxd r8, len loop1: // ロード A vmovups ymm0, [rax+r8*4] //A^2 vmulpsymm1, ymm0, ymm0 //A^3 vmulps ymm2, ymm1, ymm0 //A+A^2 vaddps ymm0, ymm0, ymm1 //A+A^2+A^3 vaddps ymm0, ymm0, ymm2 // 結果をストア vmovups [rbx+r8*4], ymm0 sub r8, 8 jge loop1 }</pre>

### 12.3 インテル® AVX コードとインテル® SSE コードの混在

インテル® AVX 命令セット・アーキテクチャーは、プログラマーが大規模なコードベースを段階的に移植することを可能にしますが、その結果インテル® AVX コードとインテル® SSE コードが混在することになります。コードにインテル® AVX とインテル® SSE が含まれる場合、次のことを考慮してください。

- Y インテル® SSE コードをインテル® コンパイラーで “/QxAVX” (Windows\*) または “-xAVX” (Linux\*) オプションを指定して再コンパイルします。これにより、すべてのインテル® SSE 命令は 128 ビットのインテル® AVX 命令へ自動的に変換されます。これは、アセンブリーと組込み関数コードに適用されます。“GCC -c -mAVX” はインテル® AVX コード (アセンブリー・ファイルを含む) を生成します。GCC アセンブラーもまた、インテル® SSE からインテル® AVX コードを生成する “-msse2AVX” オプションをサポートしています。
- Y インテル® AVX とインテル® SSE コードは共存して実行することができます。インテル® SSE コードを含むサードパーティー・ライブラリーを使用するアプリケーションが、インテル® SSE コードを実行する他のモジュールを呼び出すインテル® AVX を実装する新しい DLL を展開する場合や、アプリケーション全体を再コンパイルできない場合が該当するでしょう。この場合、インテル® AVX コードは VZEROUPPER 命令を使用して インテル® AVX/インテル® SSE 間の移行のペナルティーを避ける必要があります。

インテル® AVX 命令は YMM レジスターの上位ビットを常に更新しますが、インテル® SSE 命令はこの上位ビットを変更しません。ハードウェアの観点からは、YMM レジスターの上位ビットは次の 3 つのステートのいずれかであると考えられます。

- Y クリーン (Clean): YMM の上位ビットはゼロ。これはプロセッサが RESET から開始された状態です。
- Y 変更後未保存 (Modified and Unsaved) (表 11-2 では M/U と省略されます): インテル® AVX 命令 (256 ビットまたは 128 ビット) の実行により、YMM デスティネーションの上位ビットが変更されています。これは YMM の上位がダーティーな状態と呼ばれます。この状態では、YMM レジスターのビット 255:128 とビット 127:0 は、そのレジスターを操作した最後の (256 ビットまたは 128 ビット) インテル® AVX 命令に関連付けられています。
- Y 保存された/Non\_INIT 上位状態 (Preserved/Non\_INIT Upper State) (表 11-2 では P/N と省略されます): YMM の上位ビットは非ゼロです。YMM レジスターの上位 128 ビットと下位 128 ビットは、XRSTOR によって YMM の上位がダーティーなイメージがセーブされたため、最後に実行されたインテル® AVX 命令に関連していない可能性があります。

ソフトウェアが適切に VZEROUPPER を使用せずにインテル® AVX/インテル® SSE 命令を交互に混在させると、インテル® AVX/インテル® SSE 移行のペナルティを被ることになります。インテル® SSE、インテル® AVX、または XSAVE/XRSTOR/VZEROUPPER/VZEROALL を使用した YMM ステート管理による実行の様子を図 12-1 に示します。移行またはプロセッサ・ステート “変更後未保存” に関連するペナルティは、マイクロアーキテクチャーに依存する実装固有のもので、

図 12-1 は、Broadwell<sup>†</sup> マイクロアーキテクチャーを含むインテル® AVX をサポートする最近の世代のマイクロアーキテクチャーにおける、ステート移行のペナルティが生じる可能性を示しています。A と B の移行のペナルティは、移行を引き起こす命令が実行された時に発生します。これは、YMM ステートの全体を内部ストレージへコピーする大きなコストです。

“保存された/Non\_INIT 上位状態” に関連する YMM のステート移行の発生を最小限にするため、YMM のステートをセーブ/リストアする命令 XSAVE/XRSTOR を使用するソフトウェアは、メモリーの XSAVE 領域に上位が “クリーン” な YMM ステートを書き込む必要があります。メモリーからダーティーな YMM イメージを YMM レジスターにリストアするとペナルティを被ります。この様子を図 12-1 に示します。

Skylake<sup>†</sup> マイクロアーキテクチャーでは、インテル® SSE とインテル® AVX 命令の混在による YMM ステートの移行を管理するため、前の世代とは異なるステートマシンを実装しています。“変更後未保存 (Modified and Unsaved)” ステートでインテル® SSE 命令が実行されると、すべての YMM ステートの上位をセーブせず、個々のレジスターの上位ビットのみをセーブします。その結果、インテル® SSE とインテル® AVX 命令を混在させると、デスティネーション・レジスターの部分レジスターの依存性に関するペナルティと、デスティネーション・レジスターの上位ビットを操作する追加の混合命令が引き起こされます。図 12-2 に Skylake<sup>†</sup> マイクロアーキテクチャーに適用される移行のペナルティを示します。

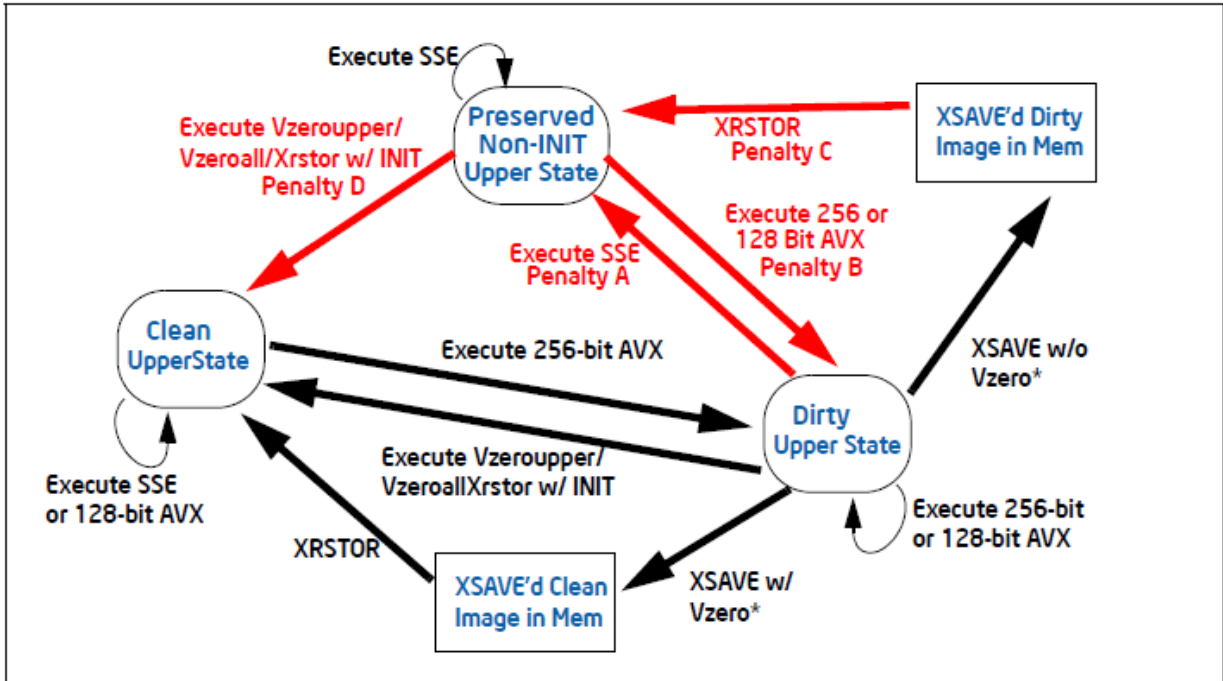


図 12-1 Broadwell+ およびそれ以前のマイクロアーキテクチャーのインテル® AVX-インテル® SSE の移行

表 12-2 に、インテル® AVX とインテル® SSE コードが混在する場合の影響を示します。最下部の行は、最初の YMM ステート（'開始' と示される行）に依存して引き起こされる可能性があるペナルティーのタイプを示しています。また、表 12-2 はメモリーにストアされたダーティーな YMM イメージに関連する移行のペナルティー（タイプ C と D）の影響を示します。

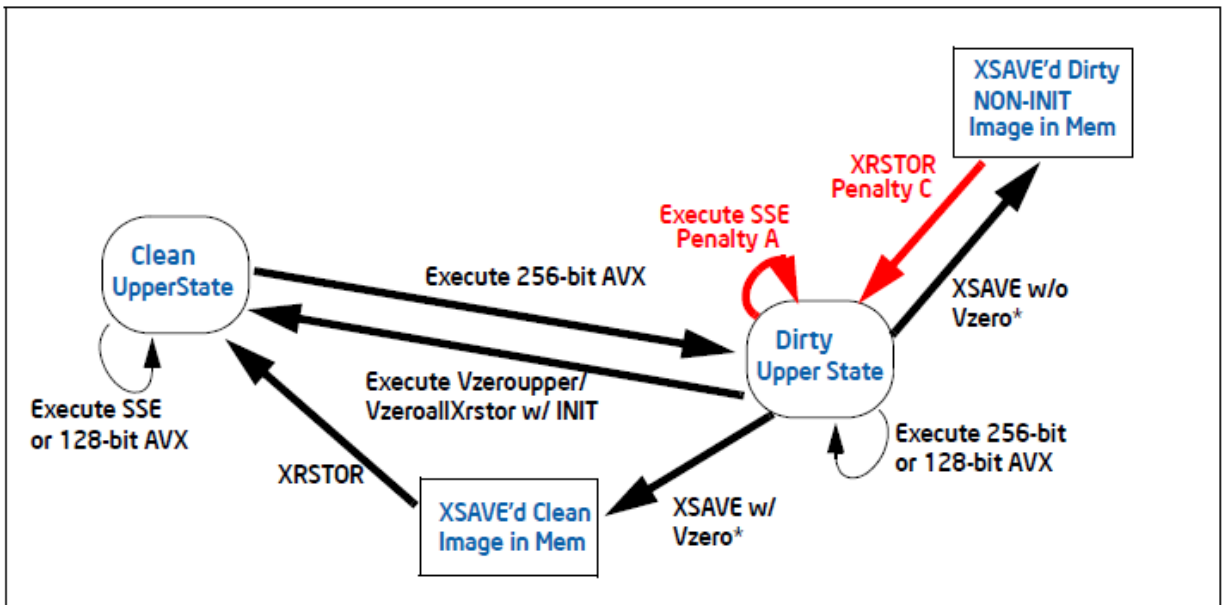


図 12-2 Skylake+ マイクロアーキテクチャーにおけるインテル® AVX-インテル® SSE の移行

表 12-2 インテル® AVX とインテル® SSE コードが混在したステート移行

	インテル® SSE を実行			128 ビットのインテル® AVX を実行			256 ビットのインテル® AVX を実行			VZeroupper		XRSTOR
	クリーン	M/U	P/N	クリーン	M/U	P/S	クリーン	P/N	P/N	P/N	ダーティーイメージ	クリーンイメージ
開始	クリーン	M/U	P/N	クリーン	M/U	P/S	クリーン	P/N	P/N	P/N	ダーティーイメージ	クリーンイメージ
終了	クリーン	P/N	P/N	クリーン	M/U	M/U	M/U	M/U	M/U	クリーン	P/N	クリーン
ペナルティー	なし	A	なし	なし	なし	B	なし	なし	B	D	C	なし

それぞれの移行ペナルティーのタイプの影響度は、マイクロアーキテクチャーによって異なります。Skylake<sup>†</sup> マイクロアーキテクチャーではいくつかの移行のペナルティーが軽減されています。図 12-2 に移行のダイアグラムと関連するペナルティーを示します。表 12-3 に、最近のマイクロアーキテクチャーにおける異なる移行ペナルティー・タイプの大まかな影響度を示しています。

表 12-3 異なるマイクロアーキテクチャーにおけるインテル® AVX-インテル® SSE 移行ペナルティーの影響度概算

タイプ	Haswell <sup>†</sup> マイクロアーキテクチャー	Broadwell <sup>†</sup> マイクロアーキテクチャー	Skylake <sup>†</sup> マイクロアーキテクチャー
A	XSAVE	XSAVE	部分的なレジスター依存性 + ブレンド
B	XSAVE	XSAVE	N/A
C	XSAVE の仮数	XSAVE の仮数	XSAVE
D	XSAVE	XSAVE	N/A

256 ビットのインテル® AVX とインテル® SSE コードブロック間の高速な移行を可能にするには、インテル® SSE コードの実行との切り替えが必要なインテル® AVX コードブロックの前後で VZEROUPPER 命令を使用します。VZEROUPPER 命令はインテル® AVX レジスターの上位 128 ビットをリセットします。この命令のレイテンシーはゼロサイクルです。さらに、プロセッサはインテル® SSE 命令やインテル® AVX 命令の実行後、クリーンステートに戻ります。今後の世代のマイクロアーキテクチャーでは移行のペナルティーはなくなります。Skylake<sup>†</sup> マイクロアーキテクチャーでは、インテル® SSE コードブロックはクリーンステートから上位ビットの依存性とブレンド操作のペナルティーなしで実行されます。

128 ビットのインテル® AVX 命令のデスティネーション・レジスターの上位 128 ビットは常にゼロです。そのため、128 ビットと 256 ビットのインテル® AVX 命令はペナルティーなしで混在できます。

**アセンブリー/コンパイラー・コーディング規則 70 (影響 H、一般性 H):** 256 ビットのインテル® AVX コードブロックと 128 ビットのインテル® SSE コードブロックが連続して実行する場合は、次のコードブロックを実行する“クリーン”ステートへの移行を容易にするため常に VZEROUPPER 命令を使用します。

### 12.3.1 関数呼び出しでインテル® AVX とインテル® SSE を混在させる

インテル® AVX とインテル® SSE 間の移行は、関数呼び出しや関数からのリターンにおいて意図せずに発生する可能性があります。例えば、256 ビットのインテル® AVX 関数がほかの関数を呼び出す場合、呼び出される関数がインテル® SSE コードを使用していることがあります。同様に、256 ビットのインテル® AVX 関数からリターンする場合に、呼び出し元がインテル® SSE コードを実行していることも考えられます。

**アセンブリー/コンパイラー・コーディング規則 71 (影響 H、一般性 H):** 256 ビットのインテル® AVX 命令が実行された後と、インテル® SSE コードが実行される関数呼び出しの前には、VZEROUPPER 命令を追加します。さらに、256 ビットのインテル® AVX 命令を使用するすべての関数の終わりに VZEROUPPER 命令を追加します。

例 12-4 関数呼び出しとインテル® AVX/インテル® SSE の移行

<pre> __attribute__((noinline)) void SSE_function() { __asm addps xmm1, xmm2 __asm xorps xmm3, xmm4 } __attribute__((noinline)) void AVX_function_no_zeroupper() { __asm vaddps ymm1, ymm2, ymm3 __asm vxorps ymm4, ymm5, ymm6 } __attribute__((noinline)) void AVX_function_with_zeroupper() { __asm vaddps ymm1, ymm2, ymm3 __asm vxorps ymm4, ymm5, ymm6 // インテル® AVX 関数からリターンする時は vzeroupper を追加 __asm vzeroupper } </pre>	
<pre> // コードは移行のペナルティーを被る __asm vaddps ymm1, ymm2, ymm3 .. // ペナルティー SSE_function(); AVX_function_no_zeroupper(); // ペナルティー __asm addps xmm1, xmm2 </pre>	<pre> // コードの移行ペナルティーを軽減 __asm vaddps ymm1, ymm2, ymm3 // インテル® AVX コードからインテル® SSE 関数を // 呼び出す前に vzeroupper を追加 __asm vzeroupper //no penalty SSE_function(); AVX_function_with_zeroupper(); // ペナルティーなし __asm addps xmm1, xmm2 </pre>

表 12-2 は、インテル® AVX コード実装とインテル® SSE コード間を移行する関数呼び出しにおいて VZERoupper を使用する場合と、使用しない場合のパフォーマンスへの影響のヒューリスティックをまとめたものです。

表 12-4 インテル® AVX とインテル® SSE コードの関数間の呼び出しにおける VZERoupper の影響

関数間の呼び出し	以前のマイクロアーキテクチャー	Skylake <sup>†</sup> マイクロアーキテクチャー
VZERoupper を使用	1X (ベースライン)	1
VZERoupper なし	< 0.1X	ベースラインの仮数

## 12.4 128 ビット・レーン操作とインテル® AVX

インテル® AVX における 256 ビット演算は、通常 128 ビット・レーンを 2 つ使用して行われます。ほとんどの 256 ビットのインテル® AVX 命令はレーンとして定義されます。各レーンのデスティネーション要素は、同じレーンのソース要素のみを使用して計算されます。以下で説明するように、レーン間で操作を行う命令もわずかにあります。

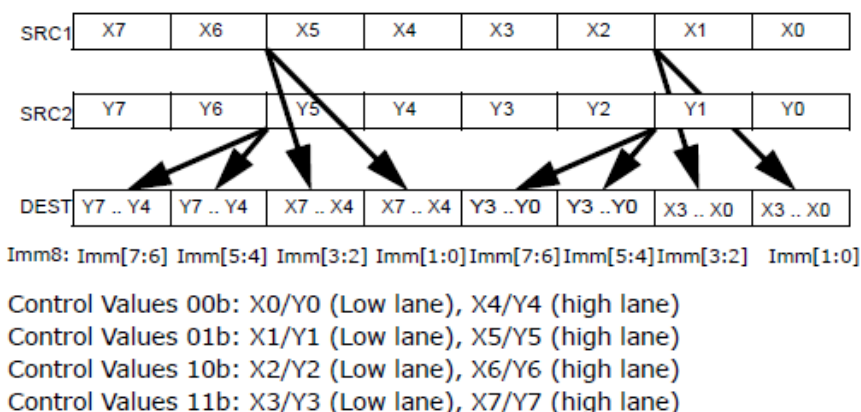
インテル® SSE 演算命令の大部分は、各データ要素の垂直位置に沿って計算を行います。128 ビット・レーンは、128 ビット・コードを 256 ビットのインテル® AVX コードへ移植する際には影響しません。VADDPS 命令はこの例の 1 つです。

多くの 128 ビットのインテル® SSE 命令はデータ要素を水平方向に移動します。例えば、SHUFPS は imm8 バイトを使用してデータ要素の水平移動を制御します。

インテル® AVX は、下位 128 ビット・レーンと上位 128 ビット・レーン内で同じ制御フィールドを使用することで、水平方向の 128 ビット SIMD 命令のレーンを 256 ビット操作に適用します。例えば 256 ビット VSHUFPS 命



令は、128 ビット・レーン内の各デスティネーション要素のソースの位置を選択するため、4 つの制御値を含む制御バイトを使用します。以下にその様子を示します。



## 12.4.1 レーンの概念とプログラミング

レーンの概念を導入しインテル® SSE 命令セットを実装したアルゴリズムは、256 ビットのインテル® AVX に容易に変換できます。反復 0 から n を実行するインテル® SSE のアルゴリズムは、反復 i を下位のレーンで計算し反復 i+k を上位のレーンで計算するように変換できます。連続する反復 k は 1 に相当します。

インテル® SSE 命令で実装されたいくつかのベクトル・アルゴリズムには、前述した簡単な変換が適用できないものがあります。例えば、16 バイト内で要素を移動するシャッフルは、32 バイト・シャッフルではレーンをまたげないため、32 バイトでシャッフルするようにそのまま変換することはできません。

レーン进行操作するビルディング・ブロックとして、次の命令を使用できます。

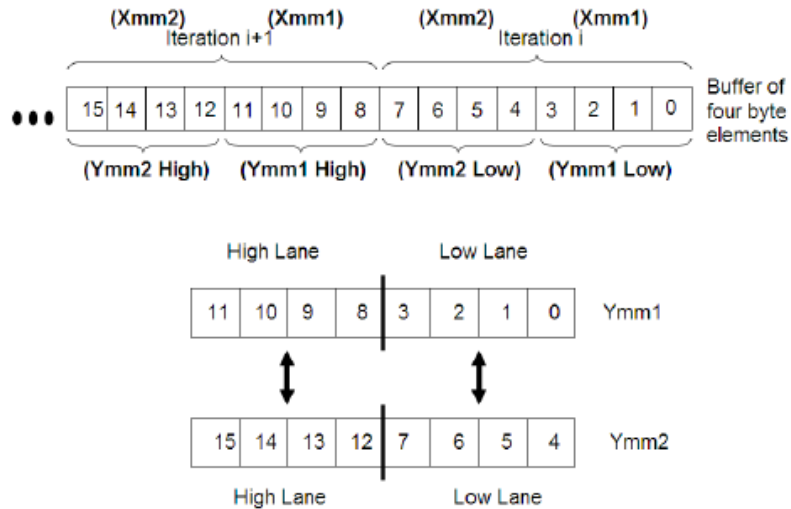
- Y VINSERTF128 - パックド浮動小数点値を挿入。
- Y VEXTRACTF128 - パックド浮動小数点値を抽出。
- Y VPERM2F128 - 浮動小数点値の並べ替え (permute)。
- Y VBROADCAST - ブロードキャストをロード。

次の節では 2 つの手法を示します: スライドロードとレジスター間のオーバーラップ。これらの方法は前述のレーン内のデータ配置を実装し、レーンを交差する計算を必要とする多くのアルゴリズムで役立ちます。

## 12.4.2 スライドロードの手法

スライドロードの手法は、インテル® AVX 命令を使用するプログラミング方式であり、サポートされないレーン間のシャッフルを伴うアルゴリズムに適しています。

レーン間のシャッフルを避けてデータを配置する方法を示します。考え方としては、対応するインテル® SSE アルゴリズムを模倣する方法で 128 ビット・ロードを使用するため、下位レーンのループ反復 i と上位レーンの反復 i+k を実行する 256 ビットのインテル® AVX 命令を有効にします。次の例において k は 1 に相当します。



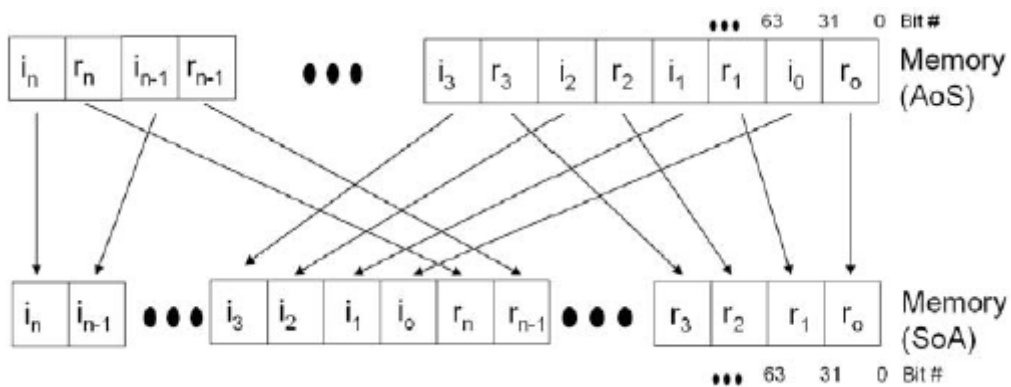
図中の Ymm1 と Ymm2 の下位レーンの値は、インテル® SSE 実装の反復  $i$  に相当します。同様に、Ymm1 と Ymm2 の上位レーンの値は、反復  $i + 1$  に相当します。

以下に、構造体配列 (AoS) から配列構造体 (SoA) への変換におけるストライドロード方式の例を示します。この例では、入力バッファーには AoS 形式の複素数値が含まれています。それぞれの複素数は実数と float 値の虚数で構成されます。出力バッファーは SoA として配置されます。複素数のすべての実数コンポーネントは、出力バッファーの前半に配置され、すべての虚数コンポーネントはバッファーの後半に格納されます。次の疑似コードと図は変換の様子を示しています。

例 12-5 C コードにおける複素数の AoS-SoA 変換

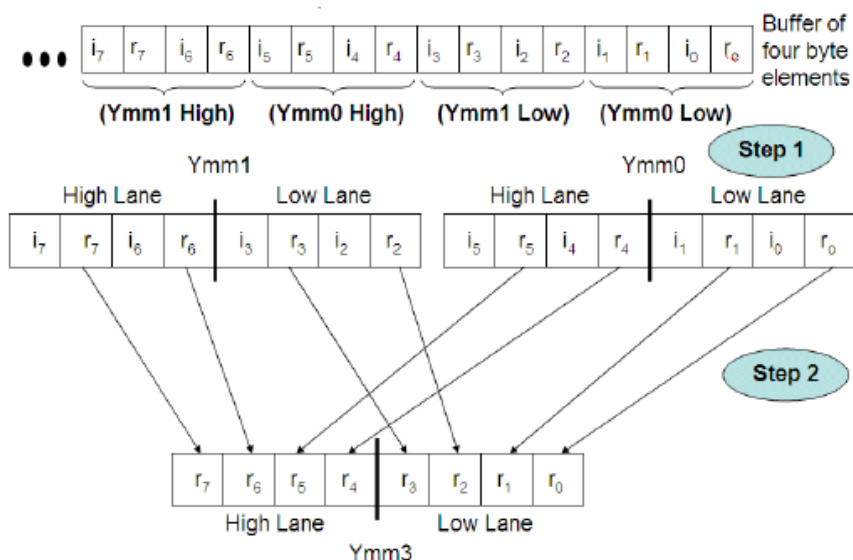
```

lfor (i=0 I < N)
{
    Real[i] =Complex[i].Real
    Imaginary[i] =Complex[i].Imaginary
}
    
```



以下の図に示すように、インテル® SSE アルゴリズムの 16 バイトから 32 バイトへの簡単な操作の拡張では、レーンを交差するデータの移動が必要になります。しかし、これはインテル® AVX 命令セット・アーキテクチャーでは不可能であるため、異なる手法が必要とされます。

レーンを交差するシャッフルの課題は、インテル® AVX による AoS から SoA への変換で解決できます。VINSERTF128 を使用して YMM レジスターの適切なレーンへ 16 バイトをロードすることで、レーンを交差するシャッフルの必要性を排除できます。ステップ 1 で YMM レジスターにデータを適切に配置されると、ステップ 2 でレーン内のデータを移動するため 32 バイト VSHUFPS を使用できます。



次のコードは、インテル® SSE 実装と 256 ビットのインテル® AVX 実装の AoS から SoA への変換の比較とパフォーマンス上のゲインを示しています。

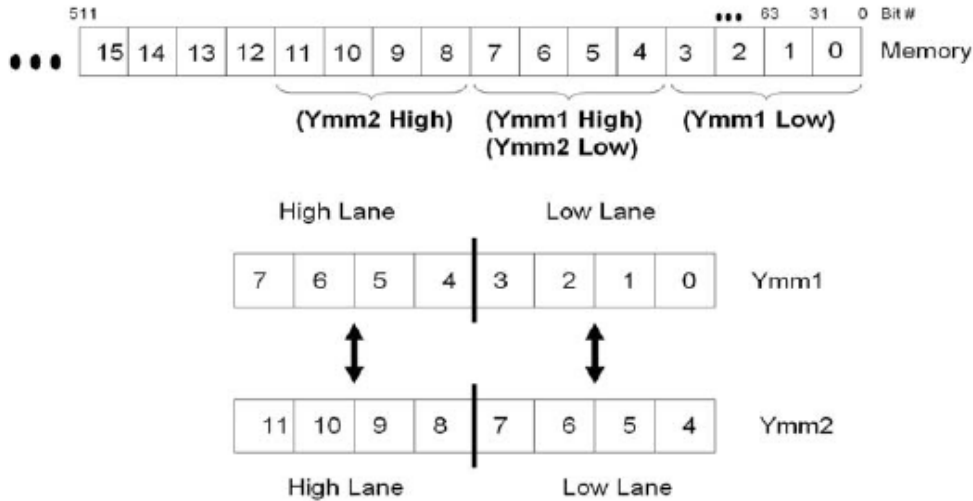
例 12-6 インテル® AVX を使用した複素数の AoS から SoA への変換

インテル® SSE コード	インテル® AVX コード
<pre> xor rbx, rbx xor rdx, rdx mov rcx, len mov rdi, inPtr mov rsi, outPtr1 mov rax, outPtr2 loop1: movups xmm0, [rdi+rbx] //i1 r1 i0 r0 movups xmm1, [rdi+rbx+16] // i3 r3 i2 r2 movups xmm2, xmm0  shufps xmm0, xmm1, 0xdd //i3 i2 i1 i0 shufps xmm2, xmm1, 0x88 //r3 r2 r1 r0  movups [rax+rdx], xmm0 movups [rsi+rdx], xmm2 add rdx, 16 add rbx, 32 cmp rcx, rbx jnz loop1                     </pre>	<pre> xor rbx, rbx xor rdx, rdx mov rcx, len mov rdi, inPtr mov rsi, outPtr1 mov rax, outPtr2 loop1: vmovups xmm0, [rdi+rbx] //i1 r1 i0 r0 vmovups xmm1, [rdi+rbx+16] // i3 r3 i2 r2 vinsertf128 ymm0, ymm0, [rdi+rbx+32], 1 //i5 r5 i4 r4; i1 r1 i0 r0 vinsertf128 ymm1, ymm1, [rdi+rbx+48], 1 //i7 r7 i6 r6; i3 r3 i2 r2 vshufps ymm2, ymm0, ymm1, 0xdd //i7 i6 i5 i4; i3 i2 i1 i0 vshufps ymm3, ymm0, ymm1, 0x88 //r7 r6 r5 r4; r3 r2 r1 r0  vmovups [rax+rdx], ymm2 vmovups [rsi+rdx], ymm3 add rdx, 32 add rbx, 64 cmp rcx, rbx jnz loop1                     </pre>

### 12.4.3 レジスター・オーバーラップの手法

レジスター・オーバーラップの手法は、シャッフルを使用するアルゴリズムにおいて有効です。ストライドロードの手法と同様に、レジスター・オーバーラップの手法はレーンを交差するシャッフルを避けてデータを配置します。

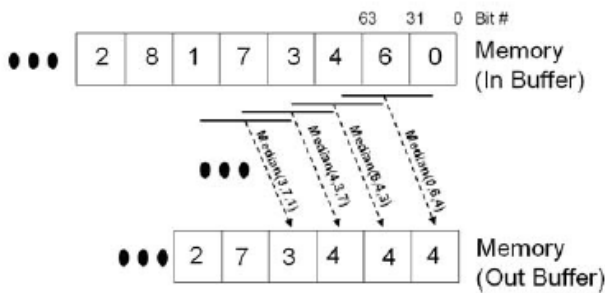
この手法は、シーケンシャルな反復によって部分的に共有される連続したデータを処理するアルゴリズムに役立ちます。以下の図は期待されるデータ配置を示します。これは、256 ビット・ロードをオーバーラップするか、VPERM2F128 命令を使用することで可能になります。



次の Median3 のサンプルコードは、レジスター・オーバーラップの手法を示しています。Median3 の手法では、ベクトル中の 3 つの連続した要素ごとの中央値を計算します。

```
Y[i] = Median( X[i], X[i+1], X[i+2] )
```

ここで Y は出力ベクトル、X は入力ベクトル。次の図はメディアン・アルゴリズムによる計算の様子を示しています。



Median3 アルゴリズムの 3 つの実装例を示します。リスト 1 はインテル® SSE の実装で、リスト 2 と 3 の実装はレジスター・オーバーラップの手法を使用した 2 つの方法です。リスト 2 では、入力バッファーから YMM レジスターへ、オーバーラップした 256 ビットのロード命令を使用してデータをロードしています。リスト 3 は、入力バッファーから YMM レジスターへ、256 ビットのロードと VPERM2F128 命令を使用してデータをロードしています。リスト 2 と 3 は広いベクトル幅によるパフォーマンスの利点を得ています。

例 12-7 3 つの数値の中央値をもとめるレジスター・オーバーラップの方法

1: インテル® SSE コード	2: オーバーラップ・ロードと 256 ビットのインテル® AVX	3: VPERM2F128 と 256 ビットの インテル® AVX
<pre>xor ebx, ebx mov rcx, len mov rdi, inPtr mov rsi, outPtr movaps xmm0, [rdi]  loop_start: movaps xmm4, [rdi+16] movaps xmm2, [rdi] movaps xmm1, [rdi] movaps xmm3, [rdi]  add rdi, 16 add rbx, 4 shufps xmm2, xmm4, 0x4e shufps xmm1, xmm2, 0x99 minps xmm3, xmm1 maxps xmm0, xmm1 minps xmm0, xmm2 maxps xmm0, xmm3 movaps [rsi], xmm0 movaps xmm0, xmm4 add rsi, 16 cmp rbx, rcx jnl loop_start</pre>	<pre>xor ebx, ebx mov rcx, len mov rdi, inPtr mov rsi, outPtr vmovaps ymm0, [rdi]  loop_start: vshufps ymm2, ymm0, [rdi+16], 0x4E vshufps ymm1, ymm0, ymm2, 0x99  add rbx, 8 add rdi, 32  vminps ymm4, ymm0, ymm1 vmaxps ymm0, ymm0, ymm1 vminps ymm3, ymm0, ymm2 vmaxps ymm5, ymm3, ymm4 vmovaps [rsi], ymm5 add rsi, 32 vmovaps ymm0, [rdi] cmp rbx, rcx jnl loop_start</pre>	<pre>xor ebx, ebx mov rcx, len mov rdi, inPtr mov rsi, outPtr vmovaps ymm0, [rdi]  loop_start: add rdi, 32 vmovaps ymm6, [rdi] vperm2f128 ymm1, ymm0, ymm6, 0x21 vshufps ymm3, ymm0, ymm1, 0x4E  vshufps ymm2, ymm0, ymm3, 0x99 add rbx, 8 vminps ymm5, ymm0, ymm2 vmaxps ymm0, ymm0, ymm2 vminps ymm4, ymm0, ymm3 vmaxps ymm7, ymm4, ymm5 vmovaps ymm0, ymm6 vmovaps [rsi], ymm7 add rsi, 32 cmp rbx, rcx jnl loop_start</pre>

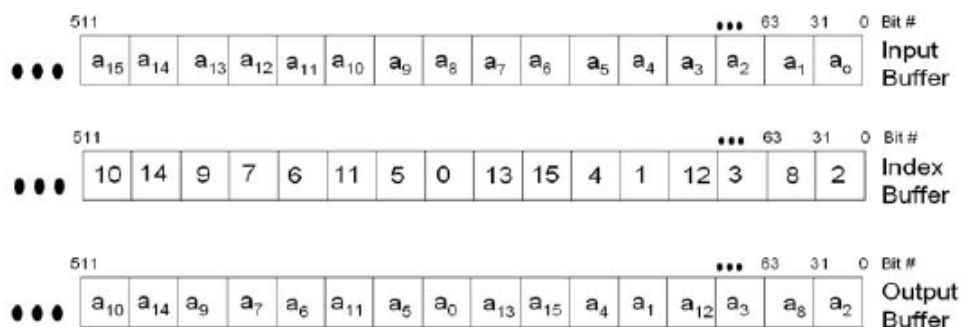
## 12.5 データのギャザーとスカッター

この節では、インテル® AVX 命令を使用したデータのギャザー（集約）とスカッター（散布）の実装手法を説明します。

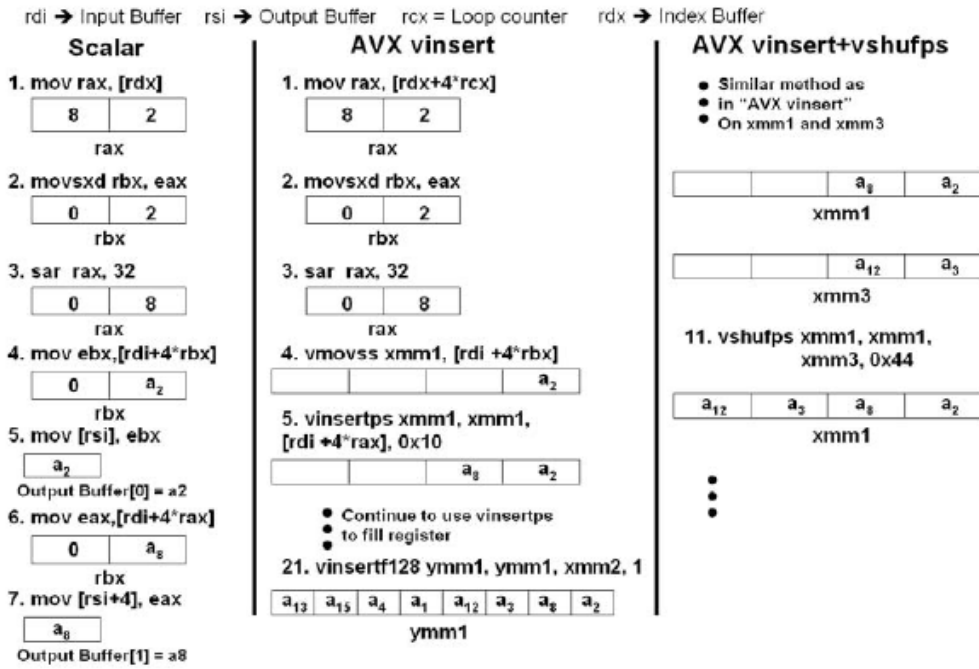
### 12.5.1 データギャザー

ギャザー操作はインデックス・バッファーで指定されたインデックスに基づいて入力バッファーから要素を読み取ります。ギャザーされた要素は出力バッファーへ書き込まれます。次の図はギャザー操作の様子を示しています。

$$\text{Output}[i] = \text{Input}[\text{Index}[i]]$$



次の 3 つの実装は 4 バイト要素の配列からのギャザー操作を行っています。リスト 1 は、汎用レジスターを使用したスカラー実装です。リスト 2 と 3 ではインテル® AVX 命令を使用しています。図には、前の図でデータの最初の反復を実行すると想定される、コードの一部を示しています (例 12-8 から抜粋)。



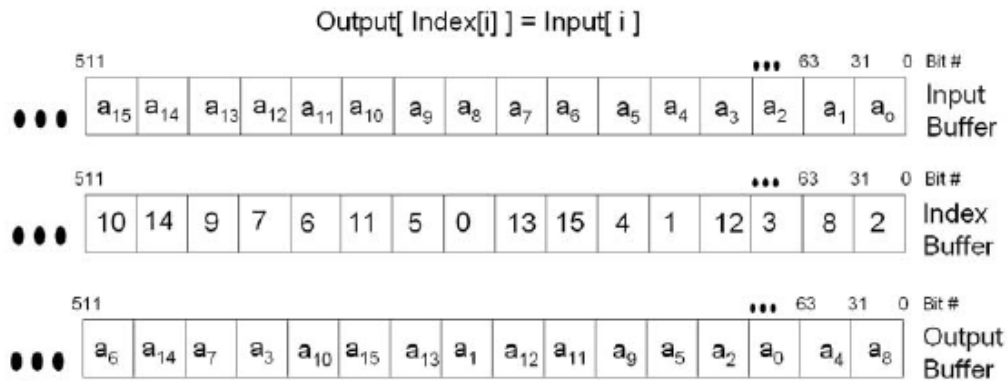
インテル® AVX 例のパフォーマンスは、対応するインテル® SSE 実装のパフォーマンスと同等です。次の表に 3 つのギャザーの実装を示します。

例 12-8 データギャザー – インテル® AVX とスカラーコード

1: スカラーコード	2: VINSRT を使用したインテル® AVX	3: VINSRT+VSHUFPS
<pre> movr di, InBuf mov rsi, OutBuf mov rdx, Index xor rcx, rcx loop1: mov rax, [rdx] movsxd rbx, eax sar rax, 32 mov ebx,[rdi+4*rbx] mov [rsi], ebx mov eax,[rdi+4*rax] mov [rsi+4], eax  mov rax, [rdx+8] movsxd rbx, eax sar rax, 32 mov ebx, [rdi+4*rbx] mov [rsi+8], ebx mov eax,[rdi+4*rax] mov [rsi+12], eax  mov rax, [rdx+16] movsxd rbx, eax sar rax, 32 mov ebx, [rdi+4*rbx] mov [rsi+16], ebx mov eax, [rdi+4*rax] mov [rsi+20], eax  mov rax, [rdx+24] movsxd rbx, eax sar rax, 32 mov ebx, [rdi+4*rbx] mov [rsi+24], ebx mov eax, [rdi+4*rax] mov [rsi+28], eax  addrsi, 32 addrdx, 32 addrcx, 8 cmprcx, len jl loop1 </pre>	<pre> movr di, InBuf mov rsi, OutBuf mov rdx, Index xor rcx, rcx loop1: mov rax, [rdx+4*rcx] movsxd rbx, eax sar rax, 32 vmovss xmm1, [rdi +4*rbx] vinsertps xmm1, xmm1,     [rdi +4*rax], 0x10  mov rax, [rdx + 8+4*rcx] movsxd rbx, eax sar rax, 32 vinsertps xmm1, xmm1,     [rdi +4*rbx], 0x20 vinsertps xmm1, xmm1,     [rdi +4*rax], 0x30  mov rax, [rdx + 16+4*rcx] movsxd rbx, eax sar rax, 32 vmovss xmm2, [rdi +4*rbx] vinsertps xmm2, xmm2,     [rdi +4*rax ], 0x10  mov rax,[rdx + 24+4*rcx] movsxd rbx, eax sar rax, 32 vinsertps xmm2, xmm2,     [rdi +4*rbx], 0x20 vinsertps xmm2, xmm2,     [rdi +4*rax], 0x30 vinsertf128 ymm1, ymm1,     xmm2, 1 vmovaps [rsi+4*rcx], ymm1 add rcx, 8 cmp rcx, len jl loop1 </pre>	<pre> movr di, InBuf mov rsi, OutBuf mov rdx, Index xor rcx, rcx loop1: mov rax, [rdx+4*rcx] movsxd rbx, eax sar rax, 32 vmovss xmm1, [rdi +4*rbx] vinsertps xmm1, xmm1,     [rdi +4*rax], 0x10  mov rax, [rdx + 8+4*rcx] movsxd rbx, eax sar rax, 32 vmovss xmm3, [rdi +4*rbx] vinsertps xmm3, xmm3,     [rdi +4*rax], 0x10 vshufps xmm1, xmm1,xmm3, 0x44  mov rax, [rdx + 16+4*rcx] movsxd rbx, eax sar rax, 32 vmovss xmm2, [rdi +4*rbx] vinsertps xmm2, xmm2,     [rdi +4*rax ], 0x10  mov rax,[rdx + 24+4*rcx] movsxd rbx, eax sar rax, 32 vmovss xmm4, [rdi +4*rbx] vinsertps xmm4, xmm4,     [rdi +4*rax], 0x10 vshufpsxmm2, xmm2,     xmm4, 0x44 vinsertf128 ymm1, ymm1,     xmm2, 1 vmovaps [rsi+4*rcx], ymm1 add rcx, 8 cmp rcx, len jl loop1 </pre>

## 12.5.2 データ・スカッター

スカッター操作は入力バッファーから要素をシーケンシャルに取り出します。そして、インデックス・バッファーで指定されたインデックスに基づいて出力バッファーへ書き込みます。次の図はスカッター操作の様子を示しています。



次の例には、スカラー実装とインテル® AVX によるスカッター・シーケンスの実装が示されています。インテル® AVX の例は、主に 128 ビットのインテル® AVX 命令で構成されます。インテル® AVX 例のパフォーマンスは、対応するインテル® SSE 実装のパフォーマンスと同等です。



例 12-9 インテル® AVX を使用したスカッター操作

スカラーコード	インテル® AVX コード
<pre> mov rdi, InBuf mov rsi, OutBuf mov rdx, Index xor  cx, rcx  loop1: movsxd rax, [rdx] mov  ebx, [rdi] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 4] mov  ebx, [rdi + 4] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 8]  mov  ebx, [rdi + 8] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 12] mov  ebx, [rdi + 12] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 16] mov  ebx, [rdi + 16] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 20] mov  ebx, [rdi + 20] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 24] mov  ebx, [rdi + 24] mov  [rsi + 4*rax], ebx movsxd rax, [rdx + 28] mov  ebx, [rdi + 28] mov  [rsi + 4*rax], ebx add  rdi, 32 add  rdx, 32 add  rcx, 8 cmp  rcx, len jl  loop1 </pre>	<pre> mov rdi, InBuf mov rsi, OutBuf mov rdx, Index xor  cx, rcx  loop1: vmovaps ymm0, [rdi + 4*rcx] movsxd rax, [rdx + 4*rcx] movsxd rbx, [rdx + 4*rcx + 4] vmovss [rsi + 4*rax], xmm0 movsxd rax, [rdx + 4*rcx + 8] vpsalignr xmm1, xmm0, xmm0, 4  vmovss [rsi + 4*rbx], xmm1 movsxd rbx, [rdx + 4*rcx + 12] vpsalignr xmm2, xmm0, xmm0, 8 vmovss [rsi + 4*rax], xmm2 movsxd rax, [rdx + 4*rcx + 16] vpsalignr xmm3, xmm0, xmm0, 12 vmovss [rsi + 4*rbx], xmm3 movsxd rbx, [rdx + 4*rcx + 20] vextractf128 xmm0, ymm0, 1 vmovss [rsi + 4*rax], xmm0 movsxd rax, [rdx + 4*rcx + 24] vpsalignr xmm1, xmm0, xmm0, 4 vmovss [rsi + 4*rbx], xmm1 movsxd rbx, [rdx + 4*rcx + 28] vpsalignr xmm2, xmm0, xmm0, 8 vmovss [rsi + 4*rax], xmm2 vpsalignr xmm3, xmm0, xmm0, 12 vmovss [rsi + 4*rbx], xmm3 add  rcx, 8 cmp  rcx, len jl  loop1 </pre>

## 12.6 インテル® AVX 向けのデータ・アライメント

この節では、インテル® AVX 命令におけるアライメントされたデータの利点を説明し、アライメントができない場合にパフォーマンスを改善するいくつかの方法を紹介します。この節では、いくつかの SAXPY カーネル例を使用して説明を行います。SAXPY はスカラーの「Alpha \* X + Y」アルゴリズムです。

以下の C コードは SAXPY の C 実装です。

```

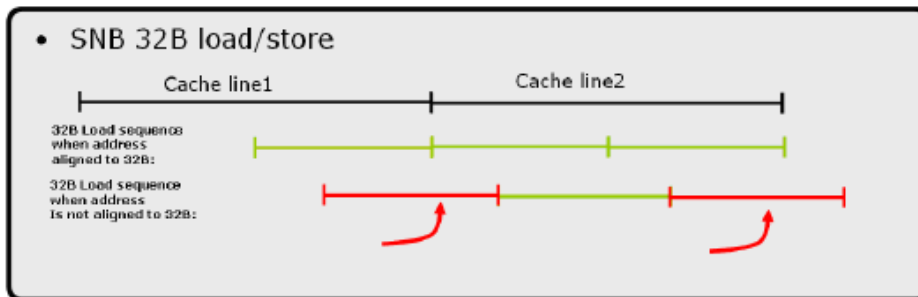
for (int i = 0; i < n; i++)
{ c[i] = alpha * a[i] + b[i]; }

```

## 12.6.1 32 バイトにデータをアライメント

ベクトル長に合わせてデータをアライメントすることを推奨します。16 バイト SIMD 命令を使用する場合、ロードされるデータは 16 バイトにアライメントされているべきです。同様に、インテル® AVX 命令で 32 バイトレジスターを使用する場合の最良の結果は、32 バイトにアライメントされたデータから得られます。

インテル® AVX でアライメントされていない 32 バイトベクトルを使用すると、2 回に一度のロードがキャッシュライン分割を引き起こします (キャッシュラインが 64 バイトであるため)。16 バイトのベクトルを使用するインテル® SSE と比べて、キャッシュライン分割の頻度は倍になります。インテル® マイクロアーキテクチャー開発コード名 Nehalem 以降、キャッシュライン分割のペナルティーは劇的に少なくなりましたが、キャッシュライン分割の頻度が高いとメモリー集約型のコードではパフォーマンスの低下の原因となります。



例 12-10 インテル® AVX を使用した SAXPY

```

mov rax, src1
mov rbx, src2
mov rcx, dst
mov rdx, len
xor rdi, rdi
vbroadcastss ymm0, alpha

start_loop:
  vmovups ymm1, [ rax + rdi ]
  vmulps ymm1, ymm1, ymm0
  vmovups ymm2, [ rbx + rdi ]
  vaddps ymm1, ymm1, ymm2
  vmovups [ rcx + rdi ], ymm1
  vmovups ymm1, [ rax + rdi + 32 ]
  vmulps ymm1, ymm1, ymm0
  vmovups ymm2, [ rbx + rdi + 32 ]
  vaddps ymm1, ymm1, ymm2
  vmovups [ rcx + rdi + 32 ], ymm1
  add rdi, 64
  cmp rdi, rdx
  jl start_loop
    
```

SAXPY は、データアライメントの重要性が高いメモリー集約型のカーネルです。最適なパフォーマンスを達成するには、両方のソースアドレスとデスティネーション・アドレスが 32 バイトにアライメントされる必要があります。3 つのアドレスのうち 1 つでも 32 バイト境界にアライメントされていないと、パフォーマンスは半減します。また、3 つのアドレスがすべて 32 バイトにアライメントされていなければ、パフォーマンスはさらに低下します。いくつかのケースでは、アライメントされていないインテル® AVX コードによるアクセスのパフォーマンスはインテル® SSE よりも低くなる場合があります。他のインテル® AVX カーネルは、通常より計算集約型であるためデータアライメントのペナルティーの影響を軽減できます。

**アセンブリ/コンパイラ・コーディング規則 72 (影響 H、一般性 M):** 可能な限りデータを 32 バイトにアライメントします。ロードのアライメントよりもストアのアライメントの方が重要です。

インテル® コンパイラで提供される `_mm_malloc` 組込み関数や Microsoft\* コンパイラの `_aligned_malloc` を使用して動的なデータ割り当てをアライメントできます。次に例を示します。

```
// 2048 float 要素のバッファを 32 バイトのアライメントで動的に割り当て、
InputBuffer = (float*)_mm_malloc (2048*sizeof(float), 32);
```

`__declspec(align(32))` を使用して静的なデータ割り当てをアライメントできます。次に例を示します。

```
// 2048 float 要素のバッファを 32 バイトのアライメントで静的に割り当て、
__declspec(align(32)) float InputBuffer[2048];
```

## 12.6.2 メモリーがアライメントされていない場合に 16 バイトのメモリーアクセスを考慮する

インテル® AVX の 32 バイト・ロードを使用して最良の結果を得るには、データを 32 バイト境界にアライメントします。しかし、データをアライメントできない場合や、データのアライメントが不明であることもあります。この状況は、ライブラリー関数を記述していて、入力データのアライメントが不明である場合に起こりえます。この場合、16 バイト・メモリー・アクセスを使用するのが最も良い手段であると考えられます。次の方法では、32 バイト YMM レジスターの恩恵を得ながら 16 バイト・ロードを使用します。

VMOVUPS、VINSERTF128 および VEXTRACTF128 命令の組み合わせを使用して、アライメントされていない 32 バイト・メモリー・アクセスを置き換えることを検討してください。

### 例 12-11 アライメントされていない 32 バイト・メモリー操作向けに使用する 16 バイト・メモリー操作

32 バイト・ロードを次のように置き換えます。

```
vmovups ymm0, mem -> vmovups xmm0, mem
vinsertf128 ymm0, ymm0, mem+16, 1
```

32 バイト・ストアを次のように置き換えます。

```
vmovups mem, ymm0 -> vmovups mem, xmm0
vextractf128 mem+16, ymm0, 1
```

次の組込み関数は、16 バイト・メモリー・アクセスを使用してアライメントされていない 32 バイト・メモリー操作を扱うことができます。

```
_mm256_loadu2_m128 ( float const * addr_hi, float const * addr_lo);
_mm256_loadu2_m128d ( double const * addr_hi, double const * addr_lo);
_mm256_loadu2_m128i ( __m128i const * addr_hi, __m128i const * addr_lo);
_mm256_storeu2_m128 ( float * addr_hi, float * addr_lo, __m256 a);
_mm256_storeu2_m128d ( double * addr_hi, double * addr_lo, __m256d a);
_mm256_storeu2_m128i ( __m128i * addr_hi, __m128i * addr_lo, __m256i a);
```

例 12-12 は、アライメントされていないアドレスによる SAXPY 向けの 2 つの実装を示しています。リスト 1 では 32 バイト・ロードを使用し、リスト 2 では 16 バイト・ロードを使用しています。これらのコードサンプルは、32 バイト境界からの 4 バイトのオフセットを持つ 2 つのソースバッファ (src1, src2) と、32 バイトにアライメントされたデスティネーション・バッファ (DST) を使用して実行します。32 バイトのメモリーアクセスの代わりに、2 つの 16 バイト・メモリー操作を行う方が高速です。

例 12-12 アライメントされていないデータアクセス向けの SAXPY 実装

インテル® AVX (32 バイト・メモリー操作)	インテル® AVX (2 つの 16 バイト・メモリー操作)
<pre> mov rax, src1 mov rbx, src2 mov rcx, dst mov rdx, len xor rdi, rdi vbroadcastss ymm0, alpha start_loop: vmovups ymm1, [rax + rdi] vmulps ymm1, ymm1, ymm0 vmovups ymm2, [rbx + rdi] vaddps ymm1, ymm1, ymm2 vmovups [rcx + rdi], ymm1  vmovups ymm1, [rax+rdi+32] vmulps ymm1, ymm1, ymm0 vmovups ymm2, [rbx+rdi+32] vaddps ymm1, ymm1, ymm2 vmovups [rcx+rdi+32], ymm1 add rdi, 64 cmp rdi, rdx j1 start_loop </pre>	<pre> mov rax, src1 mov rbx, src2 mov rcx, dst mov rdx, len xor rdi, rdi vbroadcastss ymm0, alpha start_loop: vmovups xmm2, [rax+rdi] vinsertf128 ymm2, ymm2, [rax+rdi+16], 1 vmulps ymm1, ymm0, ymm2 vmovups xmm2, [rbx + rdi] vinsertf128 ymm2, ymm2, [rbx+rdi+16], 1 vaddps ymm1, ymm1, ymm2 vaddps ymm1, ymm1, ymm2 vmovaps [rcx+rdi], ymm1 vmovups xmm2, [rax+rdi+32] vinsertf128 ymm2, ymm2, [rax+rdi+48], 1 vmulps ymm1, ymm0, ymm2 vmovups xmm2, [rbx+rdi+32] vinsertf128 ymm2, ymm2, [rbx+rdi+48], 1 vaddps ymm1, ymm1, ymm2 vmovups [rcx+rdi+32], ymm1 add rdi, 64 cmp rdi, rdx j1 start_loop </pre>

**アセンブリー/コンパイラー・コーディング規則 73 (影響 M、一般性 H):** 可能な限りデータを 32 バイトにアライメントします。ロードのアライメントよりもストアのアライメントの方が重要です。

### 12.6.3 ロードのアライメントよりもストアのアライメントが重要

処理されたデータバッファーの一部だけが、アライメントされていることがあります。そのような場合、ストア操作で使用されるアライメントされたバッファーは、ロード操作がアライメントされたバッファーを使用するよりも良いパフォーマンスをもたらします。

アライメントされていないストアは、ページをまたがるキャッシュライン分割では非常に高いペナルティーを被るため、アライメントされていないロードに比べパフォーマンスは大幅に低下します。このペナルティーは、150 サイクルと推測されます。ページ境界をまたがるロードはリタイアメント時に実行されます。例 12-12 において、アライメントされていないストアアドレスを使用すると、3 つのアライメントされていないアドレスの SAXPY のパフォーマンスは、アライメントされた場合の約 4 分の 1 になります。

## 12.7 L1D キャッシュラインの置き換え

L1D キャッシュのロードミスが発生すると、要求されたデータを含むキャッシュラインが上位メモリー階層レベルから転送されます。L1D キャッシュが常にアクティブであるメモリー集約型のコードでは、L1D キャッシュのキャッシュライン入れ替えは他のロードを遅延させる恐れがあります。インテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> と Ivy Bridge<sup>†</sup> では、32 バイト・ロードのペナルティーは 16 バイト・ロードのペナルティーよりも大きくなります。そのため、32 バイト・ロードと L1D キャッシュよりも大きなデータセットを使用するメモリー集約型のインテル® AVX コードは、同等の 16 バイト・ロードのコードよりも低速になります。

例 12-12 においてデータセットが L2 キャッシュに収まり、16 バイト・メモリー・アクセスが実装されていると、32 バイト・メモリー操作を行う場合に比べ劇的に高速化されます。

16 バイト・メモリー・アクセスと 32 バイト・メモリー・アクセスの相対的なパフォーマンスの差は、マイクロアーキテクチャーの世代による実装固有であることに注意してください。

インテル® マイクロアーキテクチャー開発コード名 Haswell では、L1D キャッシュは各サイクルで 2 つの 32 バイト・フェッチをサポートするため、ここで述べたキャッシュラインの入れ換えの問題は適用されません。

## 12.8 4K エイリアシング

4K バイト・メモリー・エイリアシングは、コードがメモリー位置にデータをストアし、その直後に 4K バイト・オフセット離れたメモリー位置からロードを行うような場合に発生します。例えば、リニアアドレス 0x400020 からのロードが、リニアアドレス 0x401020 へのストアに続く場合、ロードとストアアドレスの 5-11 ビットは同じ値であり、アクセスされるバイト・オフセットは部分的、または完全に一致します。

4K エイリアシングはロードのレイテンシーに 5 サイクルのペナルティーを加算します。4K エイリアシングが繰り返し発生し、ロードがクリティカル・パスにある場合、このペナルティーは重大である可能性があります。ロードが 2 つのキャッシュラインにまたがっている場合、衝突するストアがキャッシュにコミットされるまでロードは遅延されます。そのため、アライメントされていないインテル® AVX のロードで繰り返し 4K エイリアシングが発生すると、高いパフォーマンス・ペナルティーを被ることになります。

4K エイリアシングを検出するには LD\_BLOCKS\_PARTIAL\_ADDRESS\_ALIAS イベントを使用します。このイベントはインテル® AVX のロードが 4K エイリアシングによってブロックされた回数をカウントします。

4K エイリアシングを回避するには、次の手法を順番に試してください。

- Y データを 32 バイトにアライメントします。
- Y 可能であれば、入力と出力バッファー間のオフセットを変更します。
- Y 32 バイトにアライメントされていないメモリーには、16 バイトのメモリー・アクセスを使用します。

## 12.9 条件付き SIMD パックドロードとストア

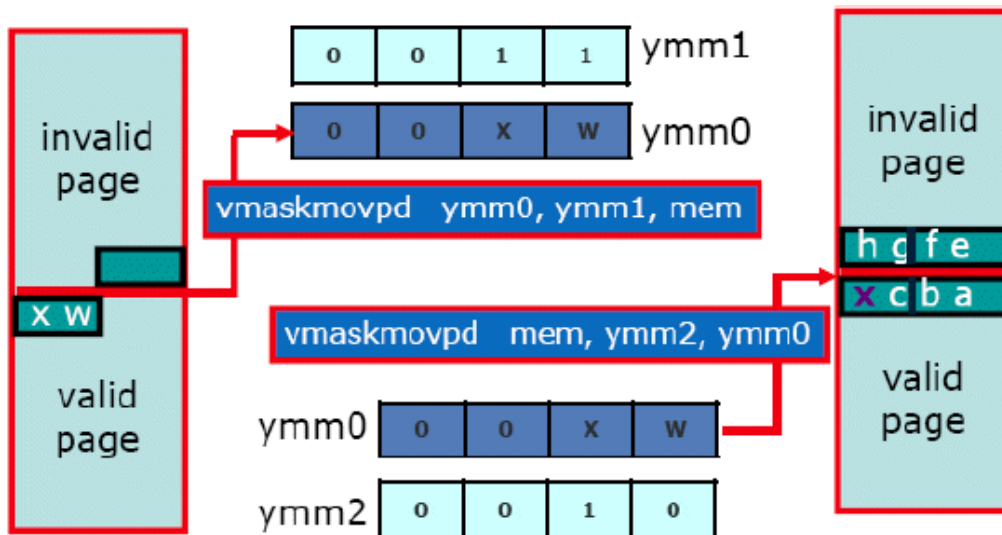
VMASKMOV 命令は、各データ要素に関連付けられたマスクビットに応じて、メモリー間との条件付きパックドデータの移動を行います。各データ要素のマスクビットは、マスクレジスター内の対応する要素の最上位ビットです。

マスク付きロードを行う場合、マスク値 0 に対応する要素には 0 が返されます。マスク付きストアは、マスク値が 1 に対応する要素のみをメモリーに書き込みます。マスク値が 0 に対応するメモリー上の要素は保持されます。メモリー・アクセスがマスクされるとフォルトが発生する可能性があります。メモリー位置に対応するマスクビットがゼロである場合、メモリー位置を参照することでフォルトは発生しません。例えば、マスクビットがすべてゼロであればフォルトは検出されません。

次の図は、フォルトを発生しないマスク付きロードとストアの例を示しています。この例では、ロード操作のためのマスクレジスターは ymm1 であり、ストア操作のマスクレジスターは ymm2 です。

マスク付きロードやストアを使用する場合次のことを留意してください。

- Y VMASKMOV ストアのアドレスは、マスクが判明した後にのみ解決されると考えられるべきです。マスク付きストアに続くロードは、マスク値が判明するまでブロックされます (メモリー・ディスアンピゲーションによって解決されない限り)。
- Y マスクがすべて 1 または 0 でないロードはマスク付きストアに依存し、ストアデータがキャッシュに書き込まれるまで待機します。マスクがすべて 1 のデータは、マスク付きストアから依存関係のあるロードへフォワードされます。マスクがすべて 0 のロードはマスク付きストアと依存関係はありません。



不正なアドレス範囲を含むマスク付きロードは、範囲がゼロ以下のマスク値であれば例外を発生しません。しかし、プロセッサは不正な範囲のどの部分にもマスク値 1 がないことを決定するため、数百サイクルの“アシスト”を必要とすることがあります。このアシストは、マスクが“ゼロ”であってもプログラマーにとってロードを実行すべきでないことが明白な場合に発生する可能性があります。

VMASKMOV を使用する際には次のことを考慮してください。

- Y VMOVUPS が使用できない場合のみ VMASKMOV を使用します。
- Y 可能であれば、32 バイトにアライメントされたアドレスで VMASKMOV を使用します。
- Y 不正な部分がゼロでマスクされていても、可能な限り有効な範囲でマスク付きロードを使用します。
- Y できるだけ早くマスクを決定します。
- Y できれば VMASKMOV ストアより前でロードを実行することで生じるストアフォワードの問題を回避します。
- Y マスク値によって VMASKMOV 命令がアシストを必要とすることがあることに注意してください (アシストが必要になるとデータをロードする VMASKMOV のレイテンシーは劇的に増加します)。
  - 不正なアドレスから 0 個の要素を選択するマスク値で VMASKMOV を使用するロードは、アシストを必要とします。
  - 不正なアドレスから特定のアドレス形式 ([base+index] や disp[base+index]) で 0 個の要素を選択するマスク値で VMASKMOV を使用するロードは、アシストを必要とします。

Skylake<sup>+</sup> マイクロアーキテクチャー・ベースのプロセッサでは、VMASKMOV 命令のパフォーマンス特性に次のような注目すべきことがあります。

- Y マスク付きストアに続くロードは、マスク値が不明であってもブロックされません。
- Y 不正なアドレスへ 0 個の要素を書き込むマスク値で VMASKMOV を使用するストアデータは、アシストを必要とします。

### 12.9.1 条件付きループ

VMASKMOV は条件式を含むループのベクトル化を可能にします。スカラー実装で VMASKMOV を使用すると 2 つの利点があります。

- Y VMASKMOV コードはベクトル化されます。
- Y 分岐予測ミスは排除されます。

次に条件付きループの C のコードを示します。

例 12-13 条件式とループ

```
for(int i = 0; i < miBufferWidth; i++)
{
    if(A[i]>0)
    {
        B[i] = (E[i]*C[i]);
    }
    else
    {
        B[i] = (E[i]*D[i]);
    }
}
```

例 12-14 VMASKMOV でループ条件を処理

スカラー	VMASKMOV を使用したインテル® AVX
<pre>float* pA = A; float* pB = B; float* pC = C; float* pD = D; float* pE = E; uint64 len = (uint64) (miBuffer- Width)*sizeof(float); __asm {     mov rax, pA     mov rbx, pB     mov rcx, pC     mov rdx, pD     mov rsi, pE     mov r8, len //xmm8 はすべてゼロ     vxorps xmm8, xmm8, xmm8     xor r9,r9 loop1:     vmovss xmm1, [rax+r9]     vcomiss xmm1, xmm8     jbe a_le a_gt:     vmovss xmm4, [rcx+r9]     jmp mul a_le:     vmovss xmm4, [rdx+r9] mul:     vmulss xmm4, xmm4, [rsi+r9]     vmovss [rbx+r9], xmm4     add r9, 4     cmp r9, r8     jl loop1 }</pre>	<pre>float* pA = A; float* pB = B; float* pC = C; float* pD = D; float* pE = E; uint64 len = (uint64) (miBufferWidth)*sizeof(float); __asm {     mov rax, pA     mov rbx, pB     mov rcx, pC     mov rdx, pD     mov rsi, pE     mov r8, len //ymm8 はすべてゼロ     vxorps ymm8, ymm8, ymm8 //ymm9 はすべて 1     vcmppps ymm9, ymm8, ymm8, 0     xor r9,r9 loop1:     vmovups ymm1, [rax+r9]     vcmppps ymm2, ymm8, ymm1, 1     vmaskmovps ymm4, ymm2, [rcx+r9]     vxorps ymm2, ymm2, ymm9     vmaskmovps ymm5, ymm2, [rdx+r9]     vorps ymm4, ymm4, ymm5     vmulps ymm4,ymm4, [rsi+r9]     vmovups [rbx+r9], ymm4     add r9, 32     cmp r9, r8     jl loop1 }</pre>

例 12-14 の左のリストのパフォーマンスは分岐予測ミスに影響され、右のデータ依存の分岐がない VMASKMOV の例より 1 桁遅くなることがあります。

## 12.10 整数と浮動小数点コードの混在

インテル® AVX 命令の整数 SIMD の機能性は 128 ビットに制限されます。整数 SIMD と浮動小数点 SIMD 命令を混在するアルゴリズムもあります。そのようなレガシー 128 ビット・コードを 256 ビットのインテル® AVX コードへ移植するには、特別な配慮が必要です。

例えば、PALINGR (Packed Align Right) は、整数と浮動小数点コードでデータ要素の配置を行う際に役立つ整数 SIMD 命令ですが、インテル® AVX には対応する 256 ビットの VPALINGR 命令がありません。

大半が浮動小数点で一部が整数操作を含むレガシーコードを 256 ビットのインテル® AVX コードへ移植する場合、検討すべき 2 つの手法があります。

- Y 重要な 128 ビットの整数 SIMD 命令に代わる 256 ビットのインテル® AVX 命令 (存在する場合) を見つけます。これは、データ要素を再配置する整数 SIMD 命令に当てはまる傾向があります。
- Y 128 ビットのインテル® AVX 命令と 256 ビットのインテル® AVX 命令を混在させます。

この 2 つの手法による性能の向上は変動する可能性があります。256 ビットのベクトル幅をフルに利用するには、できる限り最初の方法を使用します。

コードの大半が整数の場合は、128 ビットのインテル® SSE 命令から 128 ビットのインテル® AVX 命令にコードを変換し、非破壊ソース(NDS) から利点を得ることを検討します。

### 例 12-15 C コードでの 3 タップフィルター

```
for(int i = 0; i < len -2; i++)
{
    pOut[i] = A[i]*coeff[0]+A[i+1]*coeff[1]+A[i+2]*coeff[2];{B[i] = (E[i]*D[i]);
}
```



例 12-16 128 ビットの整数と FP が混在した SIMD の 3 タップフィルター

```

xor ebx, ebx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
mov r15, coeffs
movss xmm2, [r15] // coeff 0 をロード
shufps xmm2, xmm2, 0 // coeff 0 をブロードキャスト
movss xmm1, [r15+4] // coeff 1 をロード
shufps xmm1, xmm1, 0 // coeff 1 をブロードキャスト
movss xmm0, [r15+8] // coeff 2 をロード
shufps xmm0, xmm0, 0 // coeff 2 をブロードキャスト
movaps xmm5, [rdi] //xmm5={A[n+3],A[n+2],A[n+1],A[n]}
loop_start:
movaps xmm6, [rdi+16] //xmm6={A[n+7],A[n+6],A[n+5],A[n+4]}
movaps xmm7, xmm6
movaps xmm8, xmm6
add rdi, 16 //inPtr+=32
add rbx, 4 //ループカウンター
palignr xmm7, xmm5, 4 //xmm7={A[n+4],A[n+3],A[n+2],A[n+1]}
palignr xmm8, xmm5, 8 //xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
mulps xmm5, xmm2 //xmm5={C0*A[n+3],C0*A[n+2],C0*A[n+1], C0*A[n]}

mulps xmm7, xmm1 //xmm7={C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
mulps xmm8, xmm0 //xmm8={C2*A[n+5],C2*A[n+4], C2*A[n+3],C2*A[n+2]}
addps xmm7, xmm5
addps xmm7, xmm8
movaps [rsi], xmm7
movaps xmm5, xmm6
add rsi, 16 //outPtr+=16
cmp rbx, rcx
jle loop_start

```

例 12-17 VSHUFPS を使用した 256 ビットのインテル® AVX の 3 タップフィルター

```

xor ebx, ebx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
mov r15, coeffs
vbroadcastss ymm2, [r15] //coeff 0 のロードとブロードキャスト
vbroadcastss ymm1, [r15+4] //coeff 1 のロードとブロードキャスト
vbroadcastss ymm0, [r15+8] //coeff 2 のロードとブロードキャスト
loop_start:
vmovaps ymm5, [rdi] // Ymm5={A[n+7],A[n+6],A[n+5],A[n+4];
// A[n+3],A[n+2],A[n+1] , A[n]}
vshufps ymm6,ymm5,[rdi+16],0x4e // ymm6={A[n+9],A[n+8],A[n+7],A[n+6];
// A[n+5],A[n+4],A[n+3],A[n+2]}
vshufps ymm7,ymm5,ymm6,0x99 // ymm7={A[n+8],A[n+7],A[n+6],A[n+5];
// A[n+4],A[n+3],A[n+2],A[n+1]}
vmulps ymm3,ymm5,ymm2 // ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5],C0*A[n+4];
// C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
vmulps ymm9,ymm7,ymm1 // ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
// C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
vmulps ymm4,ymm6,ymm0 // ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
// C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
vaddps ymm8 ,ymm3,ymm4
vaddps ymm10, ymm8, ymm9
vmovaps [rsi], ymm10
add rdi, 32 //inPtr+=32
add rbx, 8 //ループカウンター
add rsi, 32 //outPtr+=32
cmp rbx, rcx
jle loop_start

```

例 12-18 256 ビットのインテル® AVX と128 ビットのインテル® AVX コードが混在した 3 タップフィルター

```

xor ebx, ebx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
mov r15, coeffs
vbroadcastss ymm2, [r15] //coeff 0 のロードとブロードキャスト
vbroadcastss ymm1, [r15+4] //coeff 1 のロードとブロードキャスト
vbroadcastss ymm0, [r15+8] //coeff 2 のロードとブロードキャスト
vmovaps xmm3, [rdi] //xmm3={A[n+3],A[n+2],A[n+1],A[n]}
vmovaps xmm4, [rdi+16] //xmm4={A[n+7],A[n+6],A[n+5],A[n+4]}
vmovaps xmm5, [rdi+32] //xmm5={A[n+11], A[n+10],A[n+9],A[n+8]}
loop_start:
vinsertf128 ymm3, ymm3, xmm4, 1 // ymm3={A[n+7],A[n+6],A[n+5],A[n+4];
// A[n+3], A[n+2],A[n+1],A[n]}
vpsalignr xmm6, xmm4, xmm3, 4 // xmm6={A[n+4],A[n+3],A[n+2],A[n+1]}
vpsalignr xmm7, xmm5, xmm4, 4 // xmm7={A[n+8],A[n+7],A[n+6],A[n+5]}
vinsertf128 ymm6, ymm6, xmm7, 1 // ymm6={A[n+8],A[n+7],A[n+6],A[n+5];
// A[n+4],A[n+3],A[n+2],A[n+1]}
vpsalignr xmm8, xmm4, xmm3, 8 // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
vpsalignr xmm9, xmm5, xmm4, 8 // xmm9={A[n+9],A[n+8],A[n+7],A[n+6]}
vinsertf128 ymm8, ymm8, xmm9, 1 // ymm8={A[n+9],A[n+8],A[n+7],A[n+6];
// A[n+5],A[n+4],A[n+3],A[n+2]}
vmulps ymm3, ymm3, ymm2 // Ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5], C0*A[n+4];
// C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
vmulps ymm6, ymm6, ymm1 // Ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
// C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
vmulps ymm8, ymm8, ymm0 // Ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
// C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
vaddps ymm3, ymm3, ymm6
vaddps ymm3, ymm3, ymm8
vmovaps [rsi], ymm3
vmovaps xmm3, xmm5
add rdi, 32 //inPtr+=32
add rbx, 8 //ループカウンター
add rsi, 32 //outPtr+=32
cmp rbx, rcx
jl loop_start
    
```

例 12-17 では、256 ビットの VSHUFPS で 128 ビットのインテル® SSE が混在したコード内の PALIGNR を置換しています。これにより、例 12-16 の 128 ビットのインテル® SSE が混在したコードよりもおよそ 70% 速度が向上し、例 12-18 を多少上回ります。

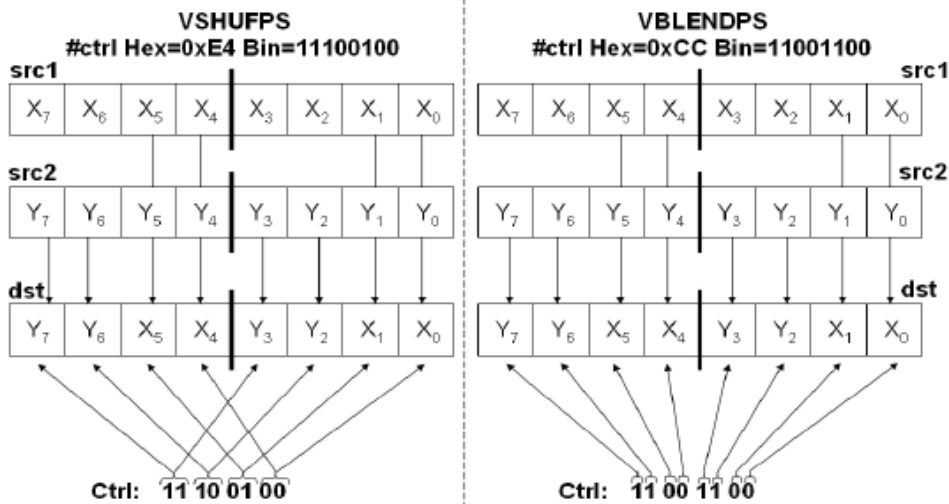
整数命令を含み、256 ビットのインテル® AVX 命令で記述されたコードでは、整数命令を、同等の機能とパフォーマンスを持つ浮動小数点命令に置き換えます。同等の浮動小数点命令がない場合は、128 ビットのインテル® AVX 命令を使用して、必要な整数操作を行うことを検討してください。

## 12.11 ポート 5 への負荷の考慮

インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge のポート 5 にはシャッフルユニットが含まれますが、これがしばしばパフォーマンスのボトルネックになることがあります。ポート 5 にのみディスパッチされるシャッフル命令を別の命令に置き換えて、ポート 5 の負荷を軽減することでパフォーマンスを向上できる場合があります。詳細は、表 2-16 を参照してください。

### 12.11.1 シャッフルをブレンドに置き換える

VSHUFPS や VPERM2F128 などのシャッフルをブレンド命令に置き換えられる場合があります。インテル® AVX のシャッフルはポート 5 でしか実行できないのに対し、ブレンド命令はポート 0 でも実行できます。そのため、シャッフルをブレンド命令に置き換えるとポート 5 の負荷を軽減できます。以下の図に、VBLENDPS を使用した VSHUFPS 実装の置き換え方法を示します。



次の例は、8x8 行列転置の 2 つの実装を示しています。どちらのケースでも、ボトルネックはポート 5 への負荷です。リスト 1 は、ポート 5 でしか実行できない 12 個の vshufps 命令を使用しています。リスト 2 では、この vshufps 命令を、ポート 0 でも実行できる vblendps 命令に置き換えています。

例 12-19 8x8 行列転置 - シャッフルをブレンドに置き換え

1) VSHUFPS を使用する 256 ビットのインテル® AVX	2) VSHUFPS を VBLENDPS で置き換えたインテル® AVX
<pre> movrcx, inpBuf movrdx, outBuf movr10, NumOfLoops movrbx, rdx loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vshufps ymm10, ymm6, ymm3, 0xE4 vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vshufps ymm11, ymm0, ymm6, 0xE4 vshufps ymm12, ymm3, ymm1, 0xE4 vperm2f128 ymm3, ymm10, ymm11, 0x20 vmovaps [rdx], ymm3 vunpckhps ymm5, ymm13, ymm14 vshufps ymm13, ymm6, ymm9, 0xE4 vunpckhps ymm4, ymm15, ymm2 vperm2f128 ymm2, ymm12, ymm13, 0x20 vmovaps 32[rdx], ymm2 vshufps ymm14, ymm8, ymm7, 0x4 vshufps ymm15, ymm14, ymm7, 0xE4 vshufps ymm7, ymm5, ymm4, 0x4E vshufps ymm8, ymm8, ymm14, 0xE4 vshufps ymm5, ymm5, ymm7, 0xE4 vperm2f128 ymm6, ymm8, ymm5, 0x20 vmovaps 64[rdx], ymm6 vshufps ymm4, ymm7, ymm4, 0xE4 vperm2f128 ymm7, ymm15, ymm4, 0x20 vmovaps 96[rdx], ymm7 vperm2f128 ymm1, ymm10, ymm11, 0x31 vperm2f128 ymm0, ymm12, ymm13, 0x31 vmovaps 128[rdx], ymm1 vperm2f128 ymm5, ymm8, ymm5, 0x31 vperm2f128 ymm4, ymm15, ymm4, 0x31 vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 decr10 jnz loop1 </pre>	<pre> movrcx, inpBuf movrdx, outBuf movr10, NumOfLoops movrbx, rdx loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vblendps ymm10, ymm6, ymm3, 0xCC vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vblendps ymm11, ymm0, ymm6, 0xCC vblendps ymm12, ymm3, ymm1, 0xCC vperm2f128 ymm3, ymm10, ymm11, 0x20 vmovaps [rdx], ymm3 vunpckhps ymm5, ymm13, ymm14 vblendps ymm13, ymm6, ymm9, 0xCC vunpckhps ymm4, ymm15, ymm2 vperm2f128 ymm2, ymm12, ymm13, 0x20 vmovaps 32[rdx], ymm2 vshufps ymm14, ymm8, ymm7, 0x4E vblendps ymm15, ymm14, ymm7, 0xCC vshufps ymm7, ymm5, ymm4, 0x4E vblendps ymm8, ymm8, ymm14, 0xCC vblendps ymm5, ymm5, ymm7, 0xCC vperm2f128 ymm6, ymm8, ymm5, 0x20 vmovaps 64[rdx], ymm6 vblendps ymm4, ymm7, ymm4, 0xCC vperm2f128 ymm7, ymm15, ymm4, 0x20 vmovaps 96[rdx], ymm7 vperm2f128 ymm1, ymm10, ymm11, 0x31 vperm2f128 ymm0, ymm12, ymm13, 0x31 vmovaps 128[rdx], ymm1 vperm2f128 ymm5, ymm8, ymm5, 0x31 vperm2f128 ymm4, ymm15, ymm4, 0x31 vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 dec r10 jnz loop1 </pre>

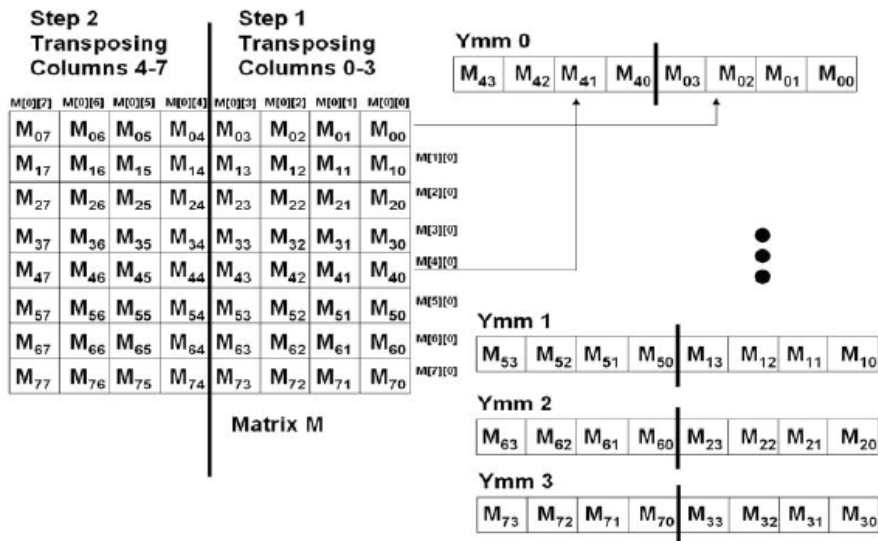
例 12-19 では、VSHUFPS を VBLENDPS に置き換えることでポート 5 の負荷を軽減し、およそ 40% スピードアップを達成しています。

**アセンブリ/コンパイラ・コーディング規則 74 (影響 M、一般性 M):** インテル® AVX のシャッフル命令の代わりに、可能であればブレンド命令を使用します。

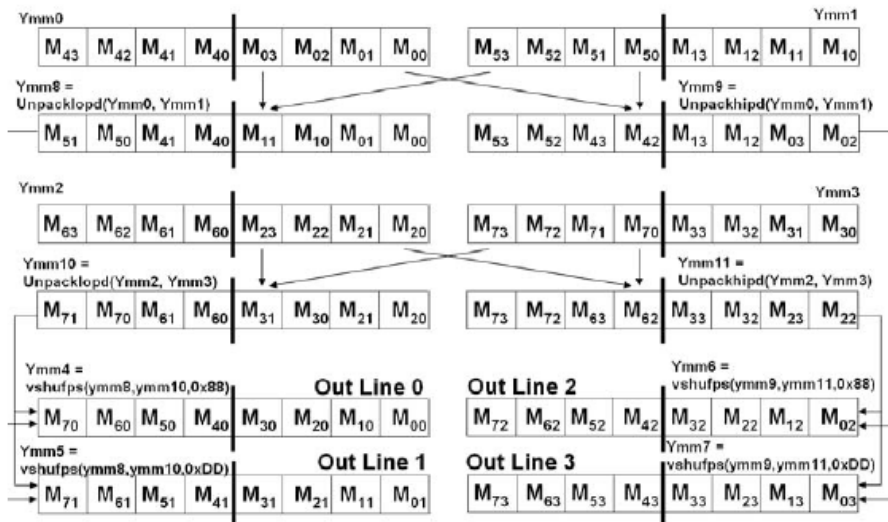
### 12.11.2 シャッフルの数を減らしたアルゴリズムの設計

アルゴリズムで使用するシャッフルの数を減らすことで、ポート 5 の負荷を軽減できる場合があります。下の図では、転置によって、0 ~ 3 行目の要素すべてが下位レーンに移動し、4 ~ 7 行目の要素すべてが上位レーンに移動しています。そのため、アルゴリズムの最初で 256 ビットのロードを使用するには、レーン間の要素を入れ替える VPERM2F128 を使用する必要があります。プロセッサは、ポート 5 でのみ VPERM2F128 命令を実行します。

例 12-19 では、8 つの 256 ビット・ロードと 8 つの VPERM2F128 命令を使用しました。この 256 ビット・ロードと 8 つの VPERM2F128 の代わりに、VINSERTF128 を使用して、同じ 8x8 行列転置を実装できます。メモリーからの VINSERTF128 は、ロードポートとポート 0 またはポート 5 で実行されます。当初の方法では、ロードポートで実行されるロードとポート 5 でのみ実行される VPERM2F128 が必要でした。そのため、VINSERTF128 を使用するようにアルゴリズムを設計し直すことで、ポート 5 の負荷が軽減されパフォーマンスが向上します。



次の図に、vinsertf128 を使用した 8x8 行列転置のステップ 1 を示します。ステップ 2 では、異なる列で同じ操作を行います。



例 12-20 VINSRTPS 使用した 8x8 行列転置

```

mov rcx, inpBuf
mov rdx, outBuf
mov r8, iLineSize
mov r10, NumOfLoops
loop1:
  vmovaps xmm0, [rcx]
  vinsertf128 ymm0, ymm0, [rcx + 128], 1
  vmovaps xmm1, [rcx + 32]
  vinsertf128 ymm1, ymm1, [rcx + 160], 1

  vunpcklpd ymm8, ymm0, ymm1
  vunpckhpd ymm9, ymm0, ymm1
  vmovaps xmm2, [rcx+64]
  vinsertf128 ymm2, ymm2, [rcx + 192], 1
  vmovaps xmm3, [rcx+96]
  vinsertf128 ymm3, ymm3, [rcx + 224], 1

  vunpcklpd ymm10, ymm2, ymm3
  vunpckhpd ymm11, ymm2, ymm3
  vshufps ymm4, ymm8, ymm10, 0x88
  vmovaps [rdx], ymm4
  vshufps ymm5, ymm8, ymm10, 0xDD
  vmovaps [rdx+32], ymm5
  vshufps ymm6, ymm9, ymm11, 0x88
  vmovaps [rdx+64], ymm6
  vshufps ymm7, ymm9, ymm11, 0xDD
  vmovaps [rdx+96], ymm7
  vmovaps xmm0, [rcx+16]
  vinsertf128 ymm0, ymm0, [rcx + 144], 1
  vmovaps xmm1, [rcx + 48]
  vinsertf128 ymm1, ymm1, [rcx + 176], 1

  vunpcklpd ymm8, ymm0, ymm1
  vunpckhpd ymm9, ymm0, ymm1

  vmovaps xmm2, [rcx+80]
  vinsertf128 ymm2, ymm2, [rcx + 208], 1
  vmovaps xmm3, [rcx+112]
  vinsertf128 ymm3, ymm3, [rcx + 240], 1

  vunpcklpd ymm10, ymm2, ymm3
  vunpckhpd ymm11, ymm2, ymm3

  vshufps ymm4, ymm8, ymm10, 0x88
  vmovaps [rdx+128], ymm4
  vshufps ymm5, ymm8, ymm10, 0xDD
  vmovaps [rdx+160], ymm5
  vshufps ymm6, ymm9, ymm11, 0x88
  vmovaps [rdx+192], ymm6
  vshufps ymm7, ymm9, ymm11, 0xDD
  vmovaps [rdx+224], ymm7
  dec r10
  jnz loop1

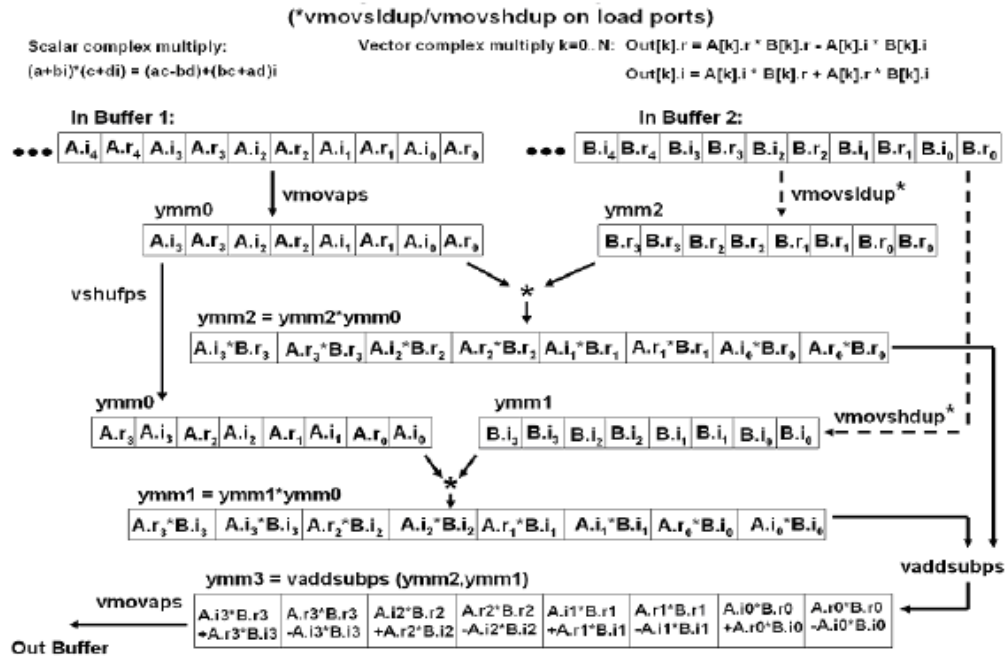
```

例 12-20 では、例 12-19 の VSHUFPS と VBLENDPS の組み合わせよりも、さらにポート 5 の負荷が軽減されます。例 12-19 の VSHUFPS だけを利用した場合と比べ、70% のスピードアップを達成できます。

### 12.11.3 ロードポートで基本的なシャッフルを実行

ソースがメモリーである場合、一部のシャッフルはロードポート (ポート 2、ポート 3) で実行できます。以下に、一部のシャッフル (vmovsldup/vmovshdup) をポート 5 からロードポートに移動することで、いかに大幅なパフォーマンス向上が得られるかを示します。

次の図は、vmovsldup/vmovshdup がロードポートにある複素数乗算アルゴリズムのインテル® AVX の実装を示しています。



例 12-21 には、2 つのバージョンの複素数乗算が含まれます。どちらのバージョンも 2 回アンロールされています。リスト 1 は、レジスター内のすべてのデータをシャッフルしています。リスト 2 は、メモリーからデータをロードしながらシャッフルします。



例 12-21 ポート 5 に対するロード・ポート・シャッフル

1) レジスター内でデータをシャッフル	2) ロードされたデータをシャッフル
<pre> mov rax, inPtr1 mov rbx, inPtr2 mov rdx, outPtr mov r8, len xor rcx, rcx  loop1: vmovaps ymm0, [rax +8*rcx] vmovaps ymm4, [rax +8*rcx +32] vmovaps ymm3, [rbx +8*rcx] vmovsldup ymm2, ymm3 vmulps ymm2, ymm2, ymm0 vshufps ymm0, ymm0, ymm0, 177 vmovshdup ymm1, ymm3 vmulps ymm1, ymm1, ymm0 vmovaps ymm7, [rbx +8*rcx +32] vmovsldup ymm6, ymm7 vmulps ymm6, ymm6, ymm4 vaddsubps ymm2, ymm2, ymm1 vmovshdup ymm5, ymm7 vmovaps [rdx+8*rcx], ymm2 vshufps ymm4, ymm4, ymm4, 177 vmulps ymm5, ymm5, ymm4 vaddsubps ymm6, ymm6, ymm5 vmovaps [rdx+8*rcx+32], ymm6  addrcx, 8 cmprcx, r8 jl loop1 </pre>	<pre> mov rax, inPtr1 mov rbx, inPtr2 mov rdx, outPtr mov r8, len xor rcx, rcx  loop1: vmovaps ymm0, [rax +8*rcx] vmovaps ymm4, [rax +8*rcx +32]  vmovsldup ymm2, [rbx +8*rcx] vmulps ymm2, ymm2, ymm0 vshufps ymm0, ymm0, ymm0, 177 vmovshdup ymm1, [rbx +8*rcx] vmulps ymm1, ymm1, ymm0 vmovsldup ymm6, [rbx +8*rcx +32] vmulps ymm6, ymm6, ymm4 vaddsubps ymm3, ymm2, ymm1 vmovshdup ymm5, [rbx +8*rcx +32]  vmovaps [rdx +8*rcx], ymm3 vshufps ymm4, ymm4, ymm4, 177 vmulps ymm5, ymm5, ymm4 vaddsubps ymm7, ymm6, ymm5 vmovaps [rdx +8*rcx +32], ymm7  addrcx, 8 cmprcx, r8 jl loop1 </pre>

## 12.12 除算と平方根命令

インテル® マイクロアーキテクチャー開発コード名 Skylake より前は、インテル® SSE の除算命令 DIVPS および平方根命令 SQRTPS はレイテンシーが 14 サイクルで、それらはパイプライン構造になっていません。つまり、これらの命令のスルーputは 14 サイクルにつき 1 回になります。128 ビットのデータパスで実行される 256 ビットのインテル® AVX 命令の VDIVPS および VSQRTPS は、レイテンシーが 28 サイクルで、同じパイプライン構造になっていません。そのため、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge では、インテル® SSE の除算命令と平方根命令のパフォーマンスは、インテル® AVX の 256 ビット命令と同程度です。

インテル® マイクロアーキテクチャー開発コード名 Skylake では、256 ビットと 128 ビットの (V)DIVPS/(V)SQRTPS は、256 ビットのデータパスで実行されるため同じレイテンシーです。レイテンシーは改善され、パイプラインで実行できるようになったことでスルーputも大幅に向上しました。付録 C の“命令のレイテンシーとスルーput”をご覧ください。

高いレイテンシーと低いスルーputの DIVPS/SQRTPS を提供するマイクロアーキテクチャーでは、(V)RSQRTPS 命令および(V)RCPPS 命令を使用して単精度の除算計算および平方根計算をスピードアップできます。例えば、128 ビットの RCPPS/RSQRTPS は 5 サイクルのレイテンシーで 1 サイクルのスルーput、また 256 ビットの同命令では 7 サイクルのレイテンシーと 2 サイクルのスルーputです。1 回のニュートンラフソン反復法またはテイラー近似式は、(V)DIVPS 命令および(V)SQRTPS 命令とほぼ同じ精度を実現できます。これらの命令の詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照してください。

除算演算や平方根演算が、これらの演算のレイテンシーの一部を隠蔽する大きなアルゴリズムに含まれる場合、ニュートンラフソンと近似法によって実行速度が低下することがあります。これは、命令が増加したことによって、マイクロオペレーション( $\mu\text{op}$ )が増え、パイプが一杯になるためです。

Skylake<sup>†</sup> マイクロアーキテクチャーでは、簡単な代数計算の最適なパフォーマンスのため近似逆数命令に対する DIVPS/SQRTPS の選択は、いくつかの要因に依存します。表 12-5 に、いくつかの代数式の異なる数値精度交差の実装におけるスループットの比較を示しています。各行の 24 ビット精度実装は IEEE に準拠しており、128 ビットまたは 256 ビット ISA をそれぞれ使用しています。22 ビットと 11 ビットの精度実装の列は、それぞれの命令セットの逆数近似命令を使用しています。

表 12-5 Skylake<sup>†</sup> マイクロアーキテクチャーにおける線形代数の数値代替手段の比較

アルゴリズム	命令タイプ	24 ビット精度	22 ビット精度	11 ビット精度
$Z = X/Y$	SSE	1X	0.9X	1.3X
	256 ビット AVX	1X	1.5X	2.6X
$Z = X^{0.5}$	SSE	1X	0.7X	2X
	256 ビット AVX	1X	1.4X	3.4X
$Z = X^{-0.5}$	SSE	1X	1.7X	4.3X
	256 ビット AVX	1X	3X	7.7X
$Z = (X * Y + Y * Y)^{0.5}$	SSE	1X	0.75X	0.85X
	256 ビット AVX	1X	1.1X	1.6X
$Z = (X+2Y+3)/(Z-2Y-3)$	SSE	1X	0.85X	1X
	256 ビット AVX	1X	0.8X	1X

ターゲット・プロセッサが Skylake<sup>†</sup> マイクロアーキテクチャーをベースとしている場合、表 12-5 のように要約できます。

- Y アルゴリズムが除算ユニットで実行される操作のみを含む場合、Skylake<sup>†</sup> マイクロアーキテクチャー上の 256 ビットのインテル® AVX コードではニュートンラフソン近似が有益です。しかし、単精度除算や平方根操作が長い計算の一部として含まれる場合、低レイテンシーの DIVPS や SQRTPS 命令は全体のパフォーマンス改善に役立ちます。
- Y インテル® SSE や 128 ビットのインテル® AVX 実装では、11 ビットより高い精度を必要としないアルゴリズムや、除算ユニットで実行される複数の処理を含むアルゴリズム向けにのみ、近似の除算と平方根命令を使用することを検討してください。

表 12-6 に、最近の世代のインテル® マイクロアーキテクチャーで必要とされる精度レベルで、単精度命令を使用する演算精度除算や平方根の推奨される計算方法を示します。

表 12-6 単精度除算と平方根の代替

演算	許容される精度	推奨
除算	24 ビット (IEEE)	DIVPS
	22 ビット	Skylake <sup>†</sup> : 表 12-5 を参照。 以前の uarch: RCPPS + 1 ニュートンラフソン反復 + MULPS
	11 ビット	RCPPS + MULPS
逆数平方根	24 ビット (IEEE)	SQRTPS + DIVPS
	22 ビット	RSQRTPS + 1 ニュートンラフソン反復
	11 ビット	RSQRTPS
平方根	24 ビット (IEEE)	SQRTPS
	22 ビット	Skylake <sup>†</sup> : 表 12-5 を参照。 以前の uarch: RSQRTPS + 1 ニュートンラフソン反復 + MULPS
	11 ビット	RSQRTPS + RCPPS

### 12.12.1 単精度除算

次の式を計算:

$$Z[i]=A[i]/B[i]$$

単精度数の大きなベクトルでは、Z[i] は除算演算により、または 1/B[i] に A[i] を掛けることで算出できます。

B[i] を N で表すと、(V)RCPPS 命令を使用して 1/N を計算でき、ほぼ 11 ビットの精度を実現できます。

精度を向上させるには、1 回のニュートンラフソン反復法を使用します。

$$\begin{aligned}
 X_0 &= 1/N && ; \text{初期推測、rcp(N)} \\
 X_0 &= 1/N*(1-E) \\
 E &= 1-N*X_0 && ; E = 2^{(-11)} \\
 X_1 &= X_0*(1+E) = 1/N*(1-E^2) && ; E^2 = 2^{(-22)} \\
 X_1 &= X_0*(1+1-N*X_0) = 2*X_0 - N*X_0^2
 \end{aligned}$$

X<sub>1</sub> は、ほぼ 22 ビット精度の 1/N の近似値です。

例 12-22 24 ビット精度で DIVPS を使用した除算

DIVPS を使用したインテル® SSE コード	VDIVPS を使用
<pre> mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx  loop1: movups xmm0, [rax+rdx*1] movups xmm1, [rbx+rdx*1] divps xmm0, xmm1 movups [rcx+rdx*1], xmm0 add rdx, 0x10 cmp rdx, rsi jl loop1 </pre>	<pre> mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx  loop1: vmovups ymm0, [rax+rdx*1] vmovups ymm1, [rbx+rdx*1] vdivps ymm0, ymm0, ymm1 vmovups [rcx+rdx*1], ymm0 add rdx, 0x20 cmp rdx, rsi jl loop1 </pre>

例 12-23 11 ビット近似で RCPPS を使用した除算

RCPPS を使用したインテル® SSE コード	VRCPPS を使用
<pre> mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx  loop1: movups xmm0, [rax+rdx*1] movups xmm1, [rbx+rdx*1] rcpps xmm1, xmm1 mulps xmm0, xmm1 movups [rcx+rdx*1], xmm0 add rdx, 16 cmp rdx, rsi jl loop1                     </pre>	<pre> mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx  loop1: vmovups ymm0, [rax+rdx] vmovups ymm1, [rbx+rdx] vrcpps ymm1, ymm1 vmulps ymm0, ymm0, ymm1 vmovups [rcx+rdx], ymm0 add rdx, 32 cmp rdx, rsi jl loop1                     </pre>

例 12-24 RCPPS とニュートンラフソン反復を使用した除算

RCPPS + MULPS 22 ビット精度	VRCPPS + VMULPS 22 ビット精度
<pre> mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx  loop1: movups xmm0, [rax+rdx*1] movups xmm1, [rbx+rdx*1] rcpps xmm3, xmm1 movaps xmm2, xmm3 addps xmm3, xmm2 mulps xmm2, xmm2 mulps xmm2, xmm1 subps xmm3, xmm2 mulps xmm0, xmm3 movups xmmword ptr [rcx+rdx*1], xmm0 add rdx, 0x10 cmp rdx, rsi jl loop1                     </pre>	<pre> mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx  loop1: vmovups ymm0, [rax+rdx] vmovups ymm1, [rbx+rdx] vrcpps ymm3, ymm1 vaddps ymm2, ymm3, ymm3 vmulps ymm3, ymm3, ymm3 vmulps ymm3, ymm3, ymm1 vsubps ymm2, ymm2, ymm3 vmulps ymm0, ymm0, ymm2 vmovups [rcx+rdx], ymm0 add rdx, 32 cmp rdx, rsi jl loop1                     </pre>

表 12-7 単精度除算の代替手法の比較

精度	方法	インテル® SSE のパフォーマンス	インテル® AVX のパフォーマンス
24 ビット	(V)DIVPS	ベースライン	1X
22 ビット	(V)RCPPS + ニュートンラフソン	2.7X	4.5X
11 ビット	(V)RCPPS	6X	8X

### 12.12.2 単精度逆数平方根

単精度数の大きなベクトルで  $Z[i]=1/(A[i])^{0.5}$  を計算するには、 $A[i]$  を  $N$  で表すと、(V)RSQRTPS 命令を使用して  $1/N$  を計算できます。

精度を向上させるには、1 回のニュートンラフソン反復法を使用します。

$X_0 = 1/N$ ; 初期推測、RCP(N)

$E = 1 - N * X_0^2$

$X_0 = (1/N)^{0.5} * ((1-E)^{0.5}) = (1/N)^{0.5} * (1-E/2)$ ;  $E/2 = 2^{-11}$

$X_1 = X_0 * (1+E/2) = (1/N)^{0.5} * (1-E^2/4)$ ;  $E^2/4 = 2^{-22}$

$X_1 = X_0 * (1 + 1/2 - 1/2 * N * X_0^2) = 1/2 * X_0 * (3 - N * X_0^2)$

X1 は、ほぼ 22 ビット精度の  $(1/N)^{0.5}$  の近似値です。

例 12-25 24 ビット精度で DIVPS + SQRTPS を使用した逆数平方根

SQRTPS, DIVPS を使用	VSQRTPS, VDIVPS を使用
<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: movups xmm1, [rax+rdx] sqrtps xmm0, xmm1 divps xmm0, xmm1 movups [rbx+rdx], xmm0 add rdx, 16 cmp rdx, rcx j1 loop1                     </pre>	<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: vmovups ymm1, [rax+rdx] vsqrtps ymm0, ymm1 vdivps ymm0, ymm0, ymm1 vmovups [rbx+rdx], ymm0 add rdx, 32 cmp rdx, rcx j1 loop1                     </pre>

例 12-26 11 ビット近似で RCPPS を使用した逆数平方根

RCPPS を使用したインテル® SSE コード	VRCPPS を使用
<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: rsqrtps xmm0, [rax+rdx] movups [rbx+rdx], xmm0 add rdx, 16 cmp rdx, rcx j1 loop1                     </pre>	<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: vrsqrtps ymm0, [rax+rdx] vmovups [rbx+rdx], ymm0 add rdx, 32 cmp rdx, rcx j1 loop1                     </pre>

例 12-27 RCPPS とニュートンラフソン反復を使用した逆数平方根

RCPPS + MULPS 22 ビット精度	VRCPPS + VMULPS 22 ビット精度
<pre> __declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5}; __declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0}; __asm {     mov rax, pIn     mov rbx, pOut     mov rcx, iLen     xor rdx, rdx     movups xmm3, [three]     movups xmm4, [minus_half] loop1:     movups xmm5, [rax+rdx]     rsqrtps xmm0, xmm5     movaps xmm2, xmm0     mulps xmm0, xmm0     mulps xmm0, xmm5     subps xmm0, xmm3     mulps xmm0, xmm2     mulps xmm0, xmm4     movups [rbx+rdx], xmm0     add rdx, 16     cmp rdx, rcx     jl loop1 } </pre>	<pre> __declspec(align(32)) float half[8] = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5}; __declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0}; __asm {     mov rax, pIn     mov rbx, pOut     mov rcx, iLen     xor rdx, rdx     vmovups ymm3, [three]     vmovups ymm4, [half] loop1:     vmovups ymm5, [rax+rdx]     vrsqrtps ymm0, ymm5     vmulps ymm2, ymm0, ymm0     vmulps ymm2, ymm2, ymm5     vsubps ymm2, ymm3, ymm2     vmulps ymm0, ymm0, ymm2     vmulps ymm0, ymm0, ymm4     vmovups [rbx+rdx], ymm0     add rdx, 32     cmp rdx, rcx     jl loop1 } </pre>

表 12-8 単精度逆数平方根命令の比較

精度	方法	インテル® SSE のパフォーマンス	インテル® AVX のパフォーマンス
24 ビット	(V)SQRTPS + (V)DIVPS	ベースライン	1X
22 ビット	(V)RCPPS + ニュートンラフソン	5.2X	9.1X
11 ビット	(V)RCPPS	13.5X	17.5X

### 12.12.3 単精度平方根

単精度数の大きなベクトルで  $Z[i] = (A[i])^{0.5}$  を計算するには、 $A[i]$  を  $N$  で表すと、 $N^{0.5}$  の近似値は  $N$  に  $(1/N)^{0.5}$  を掛けた値となります ( $(1/N)^{0.5}$  の近似値については、直前の節で説明しました)。

$N^{0.5}$  でほぼ 22 ビット精度を実現するには、以下の計算式を使用します。

$$N^{0.5} = X_1 * N = 1/2 * N * X_0 * (3 - N * X_0^2)$$

例 12-28 24 ビット精度で SQRTPS を使用した平方根

SQRTPS を使用	VSQRTPS を使用
<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: movups xmm1, [rax+rdx] sqrtps xmm1, xmm1 movups [rbx+rdx], xmm1 add rdx, 16 cmp rdx, rcx jl loop1 </pre>	<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: vmovups ymm1, [rax+rdx] vsqrtps ymm1, ymm1 vmovups [rbx+rdx], ymm1 add rdx, 32 cmp rdx, rcx jl loop1 </pre>

例 12-29 11 ビット近似で RCPPS を使用した平方根

RCPPS を使用したインテル® SSE コード	VRCPPS を使用
<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx loop1: movups xmm1, [rax+rdx] xorps xmm8, xmm8 cmpneqps xmm8, xmm1 rsqrtps xmm1, xmm1 rcpps xmm1, xmm1 andps xmm1, xmm8 movups [rbx+rdx], xmm1 add rdx, 16 cmp rdx, rcx jl loop1 </pre>	<pre> mov rax, pIn mov rbx, pOut mov rcx, iLen xor rdx, rdx v xorps ymm8, ymm8, ymm8 loop1: vmovups ymm1, [rax+rdx] vcmpneqps ymm9, ymm8, ymm1 vrsqrtps ymm1, ymm1 vrcpps ymm1, ymm1 vandps ymm1, ymm1, ymm9 vmovups [rbx+rdx], ymm1 add rdx, 32 cmp rdx, rcx jl loop1 </pre>

例 12-30 RCPPS と1 つのテイラー・シリーズ展開を使用した平方根

RCPPS + Taylor 22 ビット精度	VRCPPS + Taylor 22 ビット精度
<pre> __declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5}; __declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};  __asm {     mov rax, pIn     mov rbx, pOut     mov rcx, iLen     xor rdx, rdx     movups xmm6, [three]     movups xmm7, [minus_half]  loop1:     movups xmm3, [rax+rdx]     rsqrtps xmm1, xmm3     xorps xmm8, xmm8     cmpneqps xmm8, xmm3     andps xmm1, xmm8     movaps xmm4, xmm1     mulps xmm1, xmm3     movaps xmm5, xmm1     mulps xmm1, xmm4     subps xmm1, xmm6     mulps xmm1, xmm5     mulps xmm1, xmm7     movups [rbx+rdx], xmm1     add rdx, 16     cmp rdx, rcx     jl loop1 } </pre>	<pre> __declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0}; __declspec(align(32)) float minus_half[8] = {-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5};  __asm {     mov rax, pIn     mov rbx, pOut     mov rcx, iLen     xor rdx, rdx     vmovups ymm6, [three]     vmovups ymm7, [minus_half]     vxorps ymm8, ymm8, ymm8  loop1:     vmovups ymm3, [rax+rdx]     vrsqrtps ymm4, ymm3     vcmpneqps ymm9, ymm8, ymm3     vandps ymm4, ymm4, ymm9     vmulps ymm1, ymm4, ymm3     vmulps ymm2, ymm1, ymm4     vsubps ymm2, ymm2, ymm6     vmulps ymm1, ymm1, ymm2     vmulps ymm1, ymm1, ymm7     vmovups [rbx+rdx], ymm1     add rdx, 32     cmp rdx, rcx     jl loop1 } </pre>

表 12-9 単精度平方根命令の比較

精度	方法	インテル® SSE のパフォーマンス	インテル® AVX のパフォーマンス
24 ビット	(V)SQRTPS	ベースライン	1X
22 ビット	(V)RCPPS + Taylor 展開	2.3X	4.3X
11 ビット	(V)RCPPS	4.7X	5.9X

### 12.13 ARRAY SUB SUM の最適化例

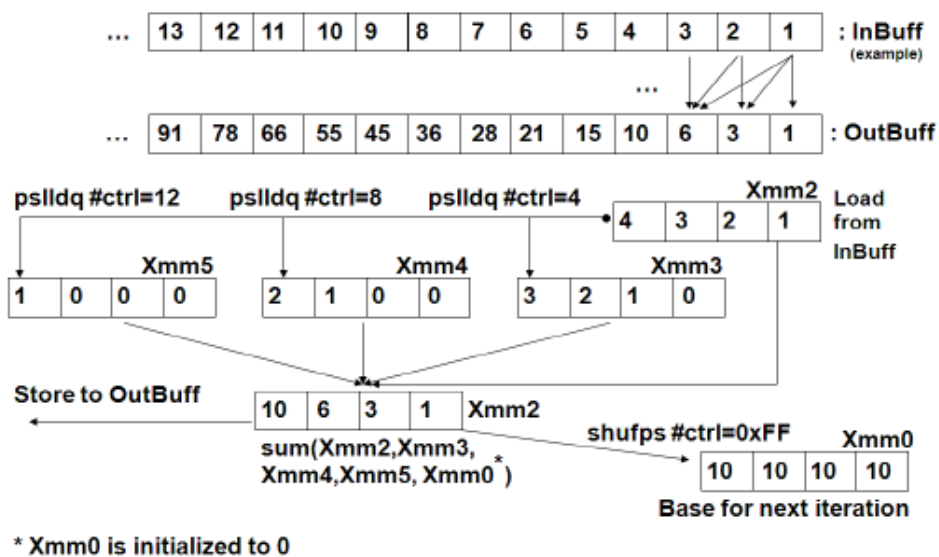
この節では、インテル® SSE 実装の Array Sub Sum アルゴリズムをインテル® AVX 実装に変換する方法を示します。

Array Sub Sum アルゴリズムは以下のとおりです。

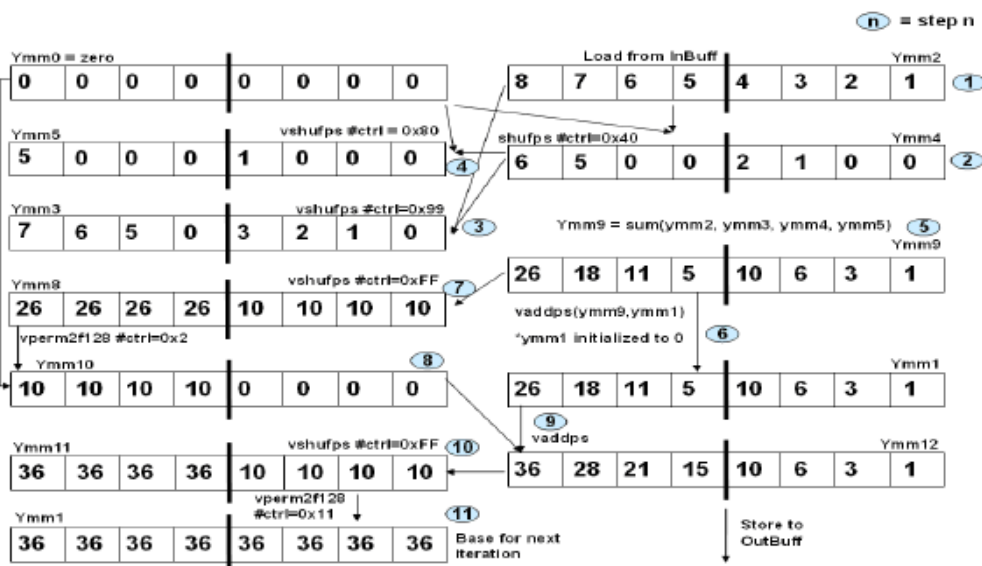
$$Y[i] = \text{Sum of } k \text{ from } 0 \text{ to } i ( X[k] ) = X[0] + X[1] + \dots + X[i]$$



以下の図はインテル® SSE 実装を示します。



以下に、Array Sub Sums アルゴリズムのインテル® AVX 実装を示します。PSLLDQ は、相当する 256 ビットの命令を持たない整数 SIMD 命令です。これは VSHUFPS で置き換えることができます。



例 12-31 Array Sub Sum アルゴリズム

インテル® SSE コード	インテル® AVX コード
<pre> mov rax, InBuff mov rbx, OutBuff mov rdx, len xor rcx, rcx xorps xmm0, xmm0  loop1: movaps xmm2, [rax+4*rcx] movaps xmm3, [rax+4*rcx] movaps xmm4, [rax+4*rcx] movaps xmm5, [rax+4*rcx] pslldq xmm3, 4 pslldq xmm4, 8 pslldq xmm5, 12 addps xmm2, xmm3 addps xmm4, xmm5 addps xmm2, xmm4 addps xmm2, xmm0 movaps xmm0, xmm2 shufps xmm0, xmm2, 0xFF movaps [rbx+4*rcx], xmm2 add rcx, 4 cmp rcx, rdx jle loop1                     </pre>	<pre> mov rax, InBuff mov rbx, OutBuff mov rdx, len xor rcx, rcx v xorps ymm0, ymm0, ymm0 v xorps ymm1, ymm1, ymm1  loop1: vmovaps ymm2, [rax+4*rcx] vshufps ymm4, ymm0, ymm2, 0x40 vshufps ymm3, ymm4, ymm2, 0x99 vshufps ymm5, ymm0, ymm4, 0x80 vaddps ymm6, ymm2, ymm3 vaddps ymm7, ymm4, ymm5 vaddps ymm9, ymm6, ymm7 vaddps ymm1, ymm9, ymm1 vshufps ymm8, ymm9, ymm9, 0xff vperm2f128 ymm10, ymm8, ymm0, 0x2 vaddps ymm12, ymm1, ymm10 vshufps ymm11, ymm12, ymm12, 0xff vperm2f128 ymm1, ymm11, ymm11, 0x11 vmovaps [rbx+4*rcx], ymm12 add rcx, 8 cmp rcx, rdx jle loop1                     </pre>

例 12-31 に Array Sub Sum のインテル® SSE 実装とインテル® AVX 実装を示します。インテル® AVX コードの方が、およそ 40% 高速です。

## 12.14 半精度浮動小数点変換

16 ビット浮動小数点形式の数値範囲と精度のみを必要とする浮動小数点アプリケーションでは、16 ビットでエンコードされた浮動小数点データを保存することでメモリー容量と帯域幅の軽減に大きな利点がもたらされます。これは、グラフィックスやイメージ処理でよく見かけられます。

半精度浮動小数点数のエンコード形式は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 4 章で確認できます。

パックド半精度浮動小数点数とパックド単精度浮動小数点数間の変換命令については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 14 章「Programming with AVX, FMA and AVX2」と、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2B』のリファレンス・ページをご覧ください。

半精度の浮動小数点データ (16 ビット FP データ要素) を計算するには、最初に単精度浮動小数点形式に変換して、必要に応じて単精度の結果を半精度形式に変換して書き戻す必要があります。256 ビットの命令を使用する 8 データ要素の変換は非常に高速で、デノーマル数、無限大、ゼロおよび NaN を適切に処理することができます。

### 12.14.1 パックド単精度から半精度への変換

単精度浮動小数点形式から半精度形式への変換には VCVTPS2PH などのハードウェアのサポートを必要とせず、プログラマーは次のことを行う必要があります。

- Y 各データ要素の範囲を許容するように指数部のバイアスを調整します。
- Y 各データ要素の仮数部をシフトして丸めます。
- Y 各要素の 15 ビット目を符号ビットへコピーします。
- Y 半精度の範囲外の数値に注意してください。
- Y 各データ要素を半分のサイズのレジスターへパックします。

例 12-32 では、単精度から半精度への 2 つの浮動小数点変換の実装を比較しています。左のコードは 128 ビット SIMD 命令セットのみとパックド整数シフト命令を使用しています。右のコードでは 2 回アンロールして VCVTTPS2PH 命令を使用しています。

例 12-32 単精度から半精度への変換

インテル® AVX-128 コード	VCVTTPS2PH コード
<pre> __asm   mov rax, pIn   mov rbx, pOut   mov rcx, bufferSize   add rcx, rax   vmovdqu xmm0, SignMask16   vmovdqu xmm1, ExpBiasFixAndRound   vmovdqu xmm4, SignMaskNot32   vmovdqu xmm5, MaxConvertibleFloat   vmovdqu xmm6, MinFloat loop:   vmovdqu xmm2, [rax]   vmovdqu xmm3, [rax+16]   vpadd xmm7, xmm2, xmm1   vpadd xmm9, xmm3, xmm1   vpand xmm7, xmm7, xmm4   vpand xmm9, xmm9, xmm4   add rax, 32   vminps xmm7, xmm7, xmm5   vminps xmm9, xmm9, xmm5   vpcmpgtd xmm8, xmm7, xmm6   vpcmpgtd xmm10, xmm9, xmm6   vpand xmm7, xmm8, xmm7   vpand xmm9, xmm10, xmm9   vpackssdw xmm2, xmm3, xmm2   vpsrad xmm7, xmm7, 13   vpsrad xmm8, xmm9, 13   vpand xmm2, xmm2, xmm0   vpackssdw xmm3, xmm7, xmm9   vpaddw xmm3, xmm3, xmm2   vmovdqu [rbx], xmm3   add rbx, 16   cmp rax, rcx   jl loop } </pre>	<pre> __asm {   mov rax, pIn   mov rbx, pOut   mov rcx, bufferSize   add rcx, rax loop:   vmovups ymm0, [rax]   vmovups ymm1, [rax+32]   add rax, 64   vcvtps2ph [rbx], ymm0, roundingCtrl   vcvtps2ph [rbx+16], ymm1, roundingCtrl   add rbx, 32   cmp rax, rcx   jl loop } </pre>

VCVTTPS2PH を使用するコードは、128 ビットのインテル® AVX シーケンスよりも約 4 倍高速です。これは 8 つのデータ要素を一度でロードできること (256 ビットのインテル® AVX) から可能なことですが、要素ごとの変換の大部分は 256 ビット拡張を持たないパックド整数命令で行われています。VCVTTPS2PH は高速だけでなく、正常な半精度浮動小数点値にエンコードされない特殊なケースを扱うことができます。

## 12.14.2 パックド半精度から単精度への変換

例 12-33 は、128 ビットのインテル® AVX コードと VCVTPH2PS をともに使用した 2 つの実装を比較しています。

半精度から単精度浮動小数点形式への変換は容易に実装できますが、VCVTPH2PS 命令を使用することで 128 ビットのインテル® AVX コードよりも約 2.5 倍高速に実行できます。

例 12-33 半精度から単精度への変換

128 ビットのインテル® AVX コード	VCVTPS2PH コード
<pre> __asm {     mov rax, pIn     mov rbx, pOut     mov rcx, bufferSize     add rcx, rax     vmovdqu xmm0, SignMask16     vmovdqu xmm1, ExpBiasFix16     vmovdqu xmm2, ExpMaskMarker loop:     vmovdqu xmm3, [rax]     add rax, 16     vpandn xmm4, xmm0, xmm3     vpand xmm5, xmm3, xmm0     vpsrlw xmm4, xmm4, 3     vpaddw xmm6, xmm4, xmm1     vpcmpgtw xmm7, xmm6, xmm2     vpand xmm6, xmm6, xmm7     vpand xmm8, xmm3, xmm7     vpor xmm6, xmm6, xmm5     vpsllw xmm8, xmm8, 13     vpunpcklwd xmm3, xmm8, xmm6     vpunpckhwd xmm4, xmm8, xmm6     vmovdqu [rbx], xmm3     vmovdqu [rbx+16], xmm4     add rbx, 32     cmp rax, rcx     jl loop } </pre>	<pre> __asm {     mov rax, pIn     mov rbx, pOut     mov rcx, bufferSize     add rcx, rax loop:     vcvtph2ps ymm0, [rax]     vcvtph2ps ymm1, [rax+16]     add rax, 32     vmovups [rbx], ymm0     vmovups [rbx+32], ymm1     add rbx, 64     cmp rax, rcx     jl loop } </pre>

## 12.14.3 帯域幅を維持するため半精度 FP の局所性を考慮

例 12-32 と例 12-33 は、ソフトウェアが半精度と単精度データ間の変換を必要とする場合に FP16C 命令を使用するパフォーマンス上の利点を示します。半精度 FP 形式は、単精度 FP 形式よりコンパクトで帯域幅を消費しませんが、数値計算が必要とする数値範囲、精度、そして変換のオーバーヘッド追加を犠牲にしています。ソフトウェアにとって半精度データを使用することが有益であるかどうかは、ワークロードの局所性に大きく依存します。

この節では、水平方向の中央値のフィルタリング・アルゴリズム “Median3” をベースにした例を使用します。Median3 のアルゴリズムは、ベクトル中の 3 つの連続した要素ごとの中央値を計算します。

$$Y[i] = \text{Median3}(X[i], X[i+1], X[i+2])$$

ここで、Y は出力ベクトル、X は入力ベクトルです。

例 12-34 は、Median3 アルゴリズムの 2 つの実装を示しています。1 つは変換なしで単精度形式を使用し、一方は半精度形式と変換を使用しています。左側のリスト 1 は、256 ビット・ロード/ストア命令を使用した単精度形式で動作しそれぞれのロード/ストアは 8 つの 32 ビット数を扱います。リスト 2 は、8 つの半精度形式の 16 ビット数値のロード/ストアに 128 ビット・ロード/ストア命令、そして単精度浮動小数点形式間との変換に VCVTPH2PS/VCVTPS2PH 命令を使用します。

例 12-34 半精度と単精度を使用した Median3 のパフォーマンス比較

1) 単精度コードと変換	2) 半精度コードと変換
<pre> __asm {     xor rbx, rbx     mov rcx, len     mov rdi, inPtr     mov rsi, outPtr     vmovaps ymm0, [rdi] loop:     add rdi, 32     vmovaps ymm6, [rdi]     vperm2f128 ymm1, ymm0, ymm6, 0x21     vshufps ymm3, ymm0, ymm1, 0x4E     vshufps ymm2, ymm0, ymm3, 0x99     vminps ymm5, ymm0, ymm2     vmaxps ymm0, ymm0, ymm2     vminps ymm4, ymm0, ymm3     vmaxps ymm7, ymm4, ymm5     vmovaps ymm0, ymm6     vmovaps [rsi], ymm7     add rsi, 32     add rbx, 8     cmp rbx, rcx     jl loop }                 </pre>	<pre> __asm {     xor rbx, rbx     mov rcx, len     mov rdi, inPtr     mov rsi, outPtr     vcvtp2ps ymm0, [rdi] loop:     add rdi, 16     vcvtp2ps ymm6, [rdi]     vperm2f128 ymm1, ymm0, ymm6, 0x21     vshufps ymm3, ymm0, ymm1, 0x4E     vshufps ymm2, ymm0, ymm3, 0x99     vminps ymm5, ymm0, ymm2     vmaxps ymm0, ymm0, ymm2     vminps ymm5, ymm0, ymm2     vmaxps ymm0, ymm0, ymm2     vminps ymm4, ymm0, ymm3     vmaxps ymm7, ymm4, ymm5     vmovaps ymm0, ymm6     vcvtps2ph [rsi], ymm7, roundingCtrl     add rsi, 16     add rbx, 8     cmp rbx, rcx     jl loop }                 </pre>

メモリー上のワーキングセットの局所性が高い場合、インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge ベースのプロセッサで半精度形式を使用すると、変換のオーバーヘッドがあるにもかかわらず単精度形式よりも約 30% 高速になります。局所性が L3 で維持される場合、半精度形式はまだ 15% 高速です。局所性が L1 で維持されると、L1 データキャッシュのキャッシュ帯域幅により単精度形式を使用した方が高速になります。L1 データキャッシュの帯域幅は上位階層のキャッシュやメモリーよりも高く、変換のオーバーヘッドがパフォーマンス上無視できなくなります。

## 12.15 乗算-加算融合 (FMA) 命令のガイドライン

FMA 命令は、IEEE-754-2008 浮動小数値に準拠した “a \* b + c” のベクトル操作を行います。ここで、“a \* b” の乗算操作は無限精度で行われ、加算の最終結果が必要とする精度に丸められて生成されます。FMA の丸めと特殊ケースの処理の詳細については、『Intel® Architecture Instruction Set Extensions Programming Reference』の 2.3 節をご覧ください。

FMA 命令は多くの FP 計算の精度を改善しスピードアップします。インテル® マイクロアーキテクチャー開発コード名 Haswell は、実行ユニットのポート 0 とポート 1 および 256 ビット・データパスで FMA 命令を実装します。ドット積、行列乗算、および多項式評価は、FMA、256 ビット・データパス、および独立した 2 つの実行ポートを使用することから利点が得られると期待されます。各プロセッサ・コアからの FMA のピーク・スループットは、サイクルごとに 16 個の単精度または 8 個の倍精度の結果です。

FMA 命令を使用するように設計されたアルゴリズムは、(FMA 命令を使用しない) MULPD/PS と ADDPD/PS のシーケンスが生成する結果が FMA を使用する場合と大きく異なることを考慮しなければなりません。収束を判断する数値計算では、丸めの問題より完了時の意図しない結果を避けるため、中間結果の精度の違いが数値形式で縮約することを考慮する必要があります。

**ユーザー/ソース・コーディング規則 33:** FMA 命令を使用せずに実行される乗算/加算命令に置き換えると、精度と丸め特性が FMA 命令とは異なります。アルゴリズムが DGEMM のように実行ポートのスループットに制限される場合、FMA はパフォーマンスを向上させます。

FMA がより高いパフォーマンスをもたらさない状況も考えられます。“ $a * b + c * d$ ” のベクトル操作とデータが同時に利用できる状況を考えてみてください。

3 つの命令シーケンスを考えてみます。

VADDPS ( VMULPS (a,b) , VMULPS (c,b) );

VMULPS は同じサイクルでディスパッチされて並列に実行されますが、VADDPS (3 サイクル) にはレイテンシーがあります。アンロールすることで VADDPS のレイテンシーは相殺されるかもしれません。

2 つの命令シーケンスを使用する場合を考えます。

VFMADD213PS ( c, d, VMULPS (a,b) );

FMA のレイテンシー (5 サイクル) は、それぞれのベクトルの結果生成で現れます。

**ユーザー/ソース・コーディング規則 34:** FP add 命令を FMA 命令で置き換える場合、FP add と FMA 命令のレイテンシーは結果依存です。

## 12.15.1 FMA と浮動小数点 Add/Mul におけるスループットの最適化

Skylake<sup>+</sup> マイクロアーキテクチャーには、FMA、ベクトル FP 乗算、および FP ADD 命令をサポートする 2 つの実行パイプがあります。これら 3 つのカテゴリの命令はすべて 4 サイクルのレイテンシーを持ち、ポート 0 またはポート 1 でディスパッチされてサイクルごとに実行されます。

同一のレイテンシーとパイプ数の配置により、浮動小数点計算が FP 乗算に続く浮動小数点加算操作によって制限されている状況のパフォーマンスをソフトウェアが改善することを可能にします。 $A_n = C_1 + C_2 * A_{n-1}$  のベクトル操作を考えてみます。

例 12-35 FP Mul/FP Add と FMA

1) FP Mul/FP Add シーケンス	2) FMA シーケンス
<pre> mov eax, NumOfIterations mov rbx, pA mov rcx, pC1 mov rdx, pC2 vmovups ymm0, Ymmword ptr [rbx] // A vmovups ymm1, Ymmword ptr [rcx] // C1 vmovups ymm2, Ymmword ptr [rdx] // C2 loop: vmulps ymm4, ymm0, ymm2 // A * C2 vaddps ymm0, ymm1, ymm4 dec eax jnz loop vmovups ymmword ptr[rbx], ymm0 // An をストア                     </pre>	<pre> mov eax, NumOfIterations mov rbx, pA mov rcx, pC1 mov rdx, pC2 vmovups ymm0, Ymmword ptr [rbx] // A vmovups ymm1, Ymmword ptr [rcx] // C1 vmovups ymm2, Ymmword ptr [rdx] // C2 loop: vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2 dec eax jnz loop vmovups ymmword ptr[rbx], ymm0 // An をストア                     </pre>
<p>反復あたりのコスト: fp add レイテンシー + fp add レイテンシー</p>	<p>反復あたりのコスト: fma レイテンシー</p>

リスト 1 のコードシーケンスの全体的なスループットは、特定のマイクロアーキテクチャーの FP MUL と FP ADD 命令のレイテンシーの組み合わせによって制限されます。また、リスト 2 のコードシーケンスの全体的なスループットは、対応するマイクロアーキテクチャーの FMA 命令のスループットによって制限されます。

FP ADD 操作のレイテンシーがパフォーマンスに影響する一般的な状況を次の C のコードに示します。

```
for ( int i = 0; i < arrLength; i ++ ) result += arrToSum[i];
```

例 12-35 にアンロールありとなしの 2 つの実装を示します。

例 12-36 FP Add のレイテンシーを隠匿するアンローリング

1) アンロールなし	2) 8 回アンロール
<pre> mov eax, arrLength mov rbx, arrToSum vmovups ymm0, Ymmword ptr [rbx] sub eax, 8 loop: add rbx, 32 vaddps ymm0, ymm0, ymmword ptr [rbx] sub eax, 8 jnz loop vextractf128 xmm1, ymm0, 1 vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0xe vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0x1 vaddss xmm0, xmm0, xmm1                     </pre>	<pre> mov eax, arrLength mov rbx, arrToSum vmovups ymm0, ymmword ptr [rbx] vmovups ymm1, ymmword ptr 32[rbx] vmovups ymm2, ymmword ptr 64[rbx] vmovups ymm3, ymmword ptr 96[rbx] vmovups ymm4, ymmword ptr 128[rbx] vmovups ymm5, ymmword ptr 160[rbx] vmovups ymm6, ymmword ptr 192[rbx] vmovups ymm7, ymmword ptr 224[rbx] sub eax, 64 loop: add rbx, 256 vaddps ymm0, ymm0, ymmword ptr [rbx] vaddps ymm1, ymm1, ymmword ptr 32[rbx] vaddps ymm2, ymm2, ymmword ptr 64[rbx] vaddps ymm3, ymm3, ymmword ptr 96[rbx] vaddps ymm4, ymm4, ymmword ptr 128[rbx] vaddps ymm5, ymm5, ymmword ptr 160[rbx] vaddps ymm6, ymm6, ymmword ptr 192[rbx] vaddps ymm7, ymm7, ymmword ptr 224[rbx] sub eax, 64 jnz loop vaddps Ymm0, ymm0, ymm1 vaddps Ymm2, ymm2, ymm3 vaddps Ymm4, ymm4, ymm5 vaddps Ymm6, ymm6, ymm7 vaddps Ymm0, ymm0, ymm2 vaddps Ymm4, ymm4, ymm6 vaddps Ymm0, ymm0, ymm4                     </pre>
<pre> vmovss result, ymm0                     </pre>	<pre> vextractf128 xmm1, ymm0, 1 vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0xe vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0x1 vaddss xmm0, xmm0, xmm1 vmovss result, ymm0                     </pre>

(例 12-35 のリスト 1 の) アンロールなしでの 8 つの配列要素ごとの累計のコストは、ワーキングセットが L1 に収まると仮定すると FP ADD 命令のレイテンシーにほぼ比例します。アンロールを効率良く使用するため、アンロールされる操作の数は少なくとも “クリティカルな操作のレイテンシー × パイプの数” にすべきです。アンロールなしバージョンに対する最適化されたアンロールバージョンのパフォーマンス・ゲインは、特定のマイクロアーキテクチャーの “パイプの数 × FP ADD のレイテンシー” で求めることができます。

**ユーザー/ソース・コーディング規則 35:** FMA, FP ADD またはベクトル MUL 操作の連続した依存関係を持つループでは、アンロールの導入を検討します。アンロール数は、依存関係が連鎖するクリティカルな命令のレイテンシーと命令を実行するパイプの数から求めることができます。



## 12.15.2 ベクトルシフトによるスループットの最適化

Skylake<sup>+</sup> マイクロアーキテクチャーでは、ほとんどの一般的なベクトルシフト命令はポート 0 またはポート 1 のいずれかにディスパッチできます。以前の世代では 1 つのポートにのみディスパッチ可能でした。2.2.2 節の表 2-3 と表 2-8 を参照してください。

FP ADD 操作のレイテンシーがパフォーマンスに影響する一般的な状況を次の C コードに示します。

コード中の a, b, c は整数配列です。

```
for ( int i = 0; i < len; i ++ ) c[i] += 4* a[i] + b[i]/2;
```

例 12-35 にアンロールありとなしの 2 つの実装を示します。

例 12-37 FP Mul/FP Add と FMA

FP Mul/FP Add シーケンス	FMA シーケンス
<pre>mov eax, NumOfIterations mov rbx, pA mov rcx, pC1 mov rdx, pC2 vmovups ymm0, Ymmword ptr [rbx] // A vmovups ymm1, Ymmword ptr [rcx] // C1 vmovups ymm2, Ymmword ptr [rdx] // C2 loop: vmulps ymm4, ymm0, ymm2 // A * C2 vaddps ymm0, ymm1, ymm4 dec eax jnz loop vmovups ymmword ptr[rbx], ymm0 // An をストア</pre>	<pre>mov eax, NumOfIterations mov rbx, pA mov rcx, pC1 mov rdx, pC2 vmovups ymm0, Ymmword ptr [rbx] // A vmovups ymm1, Ymmword ptr [rcx] // C1 vmovups ymm2, Ymmword ptr [rdx] // C2 loop: vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2 dec eax jnz loop vmovups ymmword ptr[rbx], ymm0 // An をストア</pre>
反復あたりのコスト: fp add レイテンシー + fp add レイテンシー	反復あたりのコスト: fma レイテンシー

## 12.16 インテル® AVX2 最適化のガイドライン

インテル® AVX2 命令は、128 ビット SIMD 整数命令のほとんどを 256 ビット YMM レジスターで動作するように拡張します。インテル® AVX2 はまた、数値計算を加速する豊富な broadcast/permute/各種シフト命令を備えています。256 ビットのインテル® AVX2 命令は、低レイテンシーの 256 ビット・データパスと高いスループットを実装するインテル® マイクロアーキテクチャー開発コード名 Haswell からサポートされます。

図 12-3 のイントラ符号化 4x4 ブロックのイメージ転置<sup>11</sup> を考えてみます。

128 ビット SIMD 実装では、この転置を次の方法で行うことができます。

- Y 8 ビット・ピクセルを 16 ビットの word 要素に変換し、4 つの行ベクトルとして 2 つの 4x4 イメージブロックをフェッチします。
- Y 行列操作  $1/128 * (B \times R)$  は、PMADDWD、パックドシフト、および blend 命令による SIMD シーケンスを使用して、イメージブロックの行ベクトルと右側の係数行列の列ベクトルを評価できます。
- Y 2 つの 4x4 word 粒度は、中間結果を列ベクトルに再配置できます。

<sup>11</sup> C. Ye, Y. H. Tan, Z. Li and S. Rahardja, "Mode-Dependent Fast Separable KLT for Block-based Intra Coding," JCTVC-B024, Geneva, Switzerland, Jul 2010

- Y 左側の行ベクトルの係数行列と中間ブロックの列ベクトルを (PMADDWD、shift、blend を使用して) 計算して、書き出します。

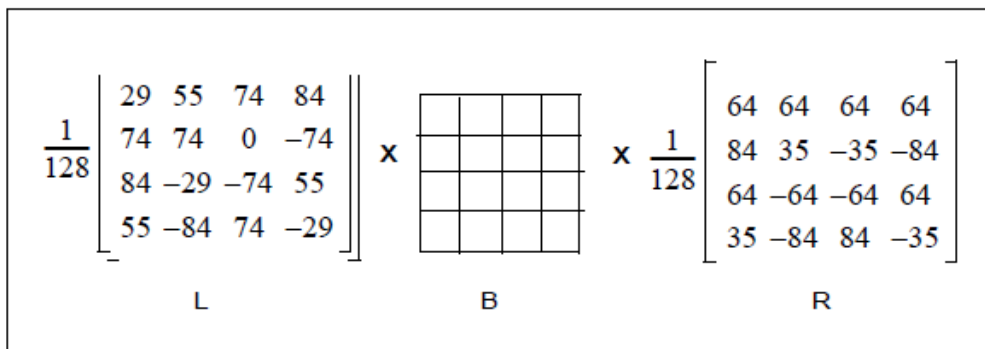


図 12-3 4x4 イメージブロックの転置

いくつかの手法はインテル® AVX2 を使用して簡単に実装できます。インテル® AVX2 のコードシーケンスを例 12-38 と例 12-39 に示します。

例 12-38 インテル® AVX2 を使用した分離可能な KLT ブロック内転置向けのマクロ

```
// b0: ワードピクセルの 4 つの連続した 4x4 イメージブロックから行ベクトルを入力
// rmc0-3: 右側の行列の列ベクトル係数、256 ビットのため 4 回繰り返す
// min32kml: 中間ピクセルを 32767 以下に制限する飽和定数ベクトル
// w0: 不明確な中間行列の行を出力、各ブロック内の要素は不明確
// 例えば、行 0 の下位 128 ビットは降順: y07, y05, y06, y04, y03, y01, y02, y00
#define __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0_256,
rmc1_256, rmc2_256, rmc3_256, min32kml)¥
{__m256i tt0, tt1, tt2, tt3;¥
    tt0 = __mm256_madd_epi16(b0, (rmc0_256));¥
    tt1 = __mm256_madd_epi16(b0, rmc1_256);¥
    tt1 = __mm256_blend_epi16(tt0);¥
    tt1 = __mm256_min_epi32(__mm256_srai_epi32( tt1, 7), min32kml);¥
    tt1 = __mm256_shuffle_epi32(tt1, 0xd8); ¥
    tt2 = __mm256_madd_epi16(b0, rmc2_256);¥
    tt3 = __mm256_madd_epi16(b0, rmc3_256);¥
    tt3 = __mm256_blend_epi16(tt2);¥
    tt3 = __mm256_min_epi32( __mm256_srai_epi32(tt3, 7), min32kml);¥
    tt3 = __mm256_shuffle_epi32(tt3, 0xd8);¥
    w0 = __mm256_blend_epi16(tt1, __mm256_slli_si256( tt3, 2), 0xaa);¥
}
(続く)
```

```
// t0-t3: 正確な中間行列 1/128 * (B x R) の256 ビット入力ベクトル
// lmr_256: 左の係数行 1 つの 256 ビットベクトル (4 回繰り返し)
// min32kml: 最終ピクセルを 32767 以下に制限する飽和定数ベクトル
// w0; 正しい順番で最終結果の行ベクトルを出力
#define __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr_256, min32kml)¥
{__m256itb0, tb1, tb2, tb3;¥
    tb0 = _mm256_madd_epi16( lmr_256, t0);¥
    tb1 = _mm256_madd_epi16( lmr_256, t1);¥
    tb1 = _mm256_blend_epi16(tb0);¥
    tb1 = _mm256_min_epi32( _mm256_srai_epi32( tb1, 7), min32kml);¥
    tb1 = _mm256_shuffle_epi32(tb1, 0xd8);¥
    tb2 = _mm256_madd_epi16( lmr_256, t2);¥
    tb3 = _mm256_madd_epi16( lmr_256, t3);¥
    tb3 = _mm256_blend_epi16(tb2, 0xf0);¥
    tb3 = _mm256_min_epi32( _mm256_srai_epi32( tb3, 7), min32kml);¥
    tb3 = _mm256_shuffle_epi32(tb3, 0xd8); ¥
    tb3 = _mm256_slli_si256( tb3, 2);¥
    tb3 = _mm256_blend_epi16(tb1, tb3, 0xaa);¥
    w0 = _mm256_shuffle_epi8(tb3, _mm256_setr_epi32( 0x5040100, 0x7060302, 0xd0c0908,
    0xf0e0b0a,
    0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a));¥
}
```

例 12-39 では、行列  $1/128 * (B \times R)$  の乗算は、word ピクセルの 4 つの連続した  $4 \times 4$  イメージブロックから最初に 4 ワイドでフェッチすることで評価されます。例 12-38 に示す最初のマクロは、各  $4 \times 4$  ブロックの 2 つの中央要素間の不明瞭なシーケンスにある、それぞれの中間列結果から出力ベクトルを生成します。例 12-39 は、shuffle/unpack プリミティブの代わりに blend プリミティブを使用して、不明瞭な要素を元に戻し中間行ベクトルから列ベクトルへの転置を実装します。

インテル® マイクロアーキテクチャー開発コード名 Haswell では、shuffle/pack/unpack プリミティブはポート 5 のシャッフル実行ユニットにディスパッチされます。重い SIMD シーケンスでは、ポート 5 への負荷がパフォーマンスを決定する重要な要因となる可能性があります。

Haswell<sup>+</sup> 上で実行されるポート 5 への負荷が高い 128 ビット SIMD コードは、256 ビットのインテル® AVX2 へ移植することでパフォーマンスを向上してポート 5 への負荷を軽減できます。

#### 例 12-39 インテル® AVX2 を使用した分離可能な KLT ブロック内転置

```
short __declspec(align(16))cst_rmc0[8] = {64, 84, 64, 35, 64, 84, 64, 35};
short __declspec(align(16))cst_rmc1[8] = {64, 35, -64, -84, 64, 35, -64, -84};
short __declspec(align(16))cst_rmc2[8] = {64, -35, -64, 84, 64, -35, -64, 84};
short __declspec(align(16))cst_rmc3[8] = {64, -84, 64, -35, 64, -84, 64, -35};
short __declspec(align(16))cst_lmr0[8] = {29, 55, 74, 84, 29, 55, 74, 84};
short __declspec(align(16))cst_lmr1[8] = {74, 74, 0, -74, 74, 74, 0, -74};
short __declspec(align(16))cst_lmr2[8] = {84, -29, -74, 44, 84, -29, -74, 55};
short __declspec(align(16))cst_lmr3[8] = {55, -84, 74, -29, 55, -84, 74, -29};
(続く)
```

```

void Klt_256_d(short * Input, short * Output, int iWidth, int iHeight)
{int iX, iY;
__m256i rmc0 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *) &cst_rmc0[0]));
__m256i rmc1 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_rmc1[0]));
__m256i rmc2 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_rmc2[0]));
__m256i rmc3 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_rmc3[0]));
__m256i lmr0 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_lmr0[0]));
__m256i lmr1 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_lmr1[0]));
__m256i lmr2 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_lmr2[0]));
__m256i lmr3 = _mm256_broadcastsil128_si256( _mm_loadu_si128((__m128i *)&cst_lmr3[0]));
__m256i min32kml = _mm256_broadcastsil128_si256( _mm_setr_epi32( 0x7fff7fff, 0x7fff7fff, 0x7fff7fff,
0x7fff7fff));
__m256i b0, b1, b2, b3, t0, t1, t2, t3;
__m256i w0, w1, w2, w3;
short* pImage = Input;
short* pOutImage = Output;
int hgt = iHeight, wid= iWidth;

// 最も内側の括弧から 1/128 * (Mat_L x (1/128 * (Mat_B x Mat_R))) を実装
for( iY = 0; iY < hgt; iY+=4) {
    for( iX = 0; iX < wid; iX+=16) {
        // word ピクセルの 4 つの連続した 4x4 行列の行 0 をロード
        b0 = _mm256_loadu_si256( (__m256i *) (pImage + iY*wid+ iX));
        // 右の行列係数の列ベクトルと行 0 を掛ける
        __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0, rmc1, rmc2, rmc3, min32kml);
        // 不正確な行 0 の下位 128 ビット、上位 -> 下位: y07, y05, y06, y04, y03, y01, y02, y00
        b1 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+1)*wid+ iX));
        __MyM_KIP_PxRMC_ROW_4x4Wx4(b1, w1, rmc0, rmc1, rmc2, rmc3, min32kml);
        // hi->lo y17, y15, y16, y14, y13, y11, y12, y10
        b2 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+2)*wid+ iX));
        __MyM_KIP_PxRMC_ROW_4x4Wx4(b2, w2, rmc0, rmc1, rmc2, rmc3, min32kml);
        b3 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+3)*wid+ iX));
        __MyM_KIP_PxRMC_ROW_4x4Wx4(b3, w3, rmc0, rmc1, rmc2, rmc3, min32kml);
        // 各 4x4 ブロックの不正確な中央 2 つの要素を元に戻し、
        // 不正確なベクトルへ転置: t0 は 4 つの連続した行 0 または 4 つの 4x4 の中間値を持つ
        t0 = _mm256_blend_epil6( w0, _mm256_slli_epi64(w1, 16), 0x22);
        t0 = _mm256_blend_epil6( t0, _mm256_slli_epi64(w2, 32), 0x44);
        t0 = _mm256_blend_epil6( t0, _mm256_slli_epi64(w3, 48), 0x88);
        t1 = _mm256_blend_epil6( _mm256_srli_epi64(w0, 32), _mm256_srli_epi64(w1, 16), 0x22);
        t1 = _mm256_blend_epil6( t1, w2, 0x44);
        t1 = _mm256_blend_epil6( t1, _mm256_slli_epi64(w3, 16), 0x88); // column 1
        t2 = _mm256_blend_epil6( _mm256_srli_epi64(w0, 16), w1, 0x22);
        t2 = _mm256_blend_epil6( t2, _mm256_slli_epi64(w2, 16), 0x44);
        t2 = _mm256_blend_epil6( t2, _mm256_slli_epi64(w3, 32), 0x88); // column 2
        t3 = _mm256_blend_epil6( _mm256_srli_epi64(w0, 48), _mm256_srli_epi64(w1, 32), 0x22);
        t3 = _mm256_blend_epil6( t3, _mm256_srli_epi64(w2, 16), 0x44);
        t3 = _mm256_blend_epil6( t3, w3, 0x88); // column 3
        // 中間ブロックの 4 つの列ベクトルと左の係数ベクトルの行 0 を掛ける
        // 最終的な出力行は通常の並びに配置される
        __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr0, min32kml);
        _mm256_store_si256( (__m256i *) (pOutImage+iY*wid+ iX), w0);
        __MyM_KIP_LMRxP_ROW_4x4Wx4(w1, t0, t1, t2, t3, lmr1, min32kml);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+1)*wid+ iX), w1);
    }
}
(続く)

```

```

__MyM_KIP_LMRxP_ROW_4x4Wx4(w2, t0, t1, t2, t3, lmr2, min32kml);
__mm256_store_si256( (__m256i *) (pOutImage+(iY+2)*wid+ iX), w2) ;
__MyM_KIP_LMRxP_ROW_4x4Wx4(w3, t0, t1, t2, t3, lmr3, min32kml);
__mm256_store_si256( (__m256i *) (pOutImage+(iY+3)*wid+ iX), w3) ;
}
}

```

ここでは 128 ビットの SIMD 実装を示しませんが簡単に作成できます。

この KLT イントラ符号化転置の 128 ビット SIMD コードを、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge で実行すると、2 つのシャッフルユニットによりポート 5 への負荷は軽減され、各 4x4 イメージブロックの転置のための有効スループットはおよそ 50 サイクルとなります。最適化されたスカラー実装からの相対的なスピードアップは、約 2.5 倍です。

Haswell<sup>+</sup> 上で 128 ビット SIMD コードを実行すると、ポート 5 へ発行される  $\mu\text{op}$  はすべての  $\mu\text{op}$  の 50% 未満であり、これまでの世代のマイクロアーキテクチャーではおよそ 3 分の 1 であったため、パフォーマンスが約 25% 低下します。一方、インテル® AVX2 の実装では 4x4 ブロックあたり 35 サイクル未満の有効スループットを提供します。

### 12.16.1 マルチバッファリングとインテル® AVX2

データバッファのストリームを操作する計算集約型のアルゴリズムも多くあります (ハッシュ、暗号化など)。データストリームは、SIMD 命令セットを利用するためパーティション化され、複数の独立したバッファーストリームとして扱われることがあります。

複数のバッファを並列にハッシュする方法の詳細については、<http://eprint.iacr.org/2012/476.pdf> (英語) および <http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995> (英語) をご覧ください。

インテル® AVX2 では、論値と数値操作向けに複数のデータ型で豊富な機能性と完全な 256 ビット SIMD 命令が提供されます。XMM レジスターとこれまでの世代のインテル® SSE 命令セットに投資されたアルゴリズムは、インテル® AVX2 の YMM を使用して高いスループットを提供するため、複数バッファリング・アルゴリズムを拡張できます。最適化された 256 ビットのインテル® AVX2 実装では、128 ビット・バージョンと比較して最大 1.9 倍のスループットが得られます。

12.16 節で述べたイメージブロック転置の例は、4x4 ブロックの複数バッファリングの実装として解釈できます。パフォーマンスのベースラインを 2 つのシャッフルポートのマイクロアーキテクチャー (Sandy Bridge<sup>+</sup>) から、単一シャッフルポートのマイクロアーキテクチャーへ切り替えると、256 ビット幅のインテル® AVX2 は 128 ビット SIMD の実装に対して 1.9 倍のスピードアップをもたらします。

複数バッファリングの詳細については、<http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-iamulti-buffer-paper.pdf> (英語) のホワイトペーパーをご覧ください。

### 12.16.2 剰余除算とインテル® AVX2

RSA 2048 などの公開鍵暗号化において重要なモジュラー冪剰余操作を効率良く実装するため、非常に大きな整数の剰余乗算はしばしば使用されます。剰余乗算のライブラリー実装では、多くの場合 MUL/ADC のチェーンシーケンスが利用されます。通常、MUL 命令は 128 ビットの中間的な整数出力を生成し、64 ビットの間データの粒度で加算キャリーチェーンを使用する必要があります。

インテル® AVX2 において、VPMULUDQ/VPADDQ/VPSRLQ/VPSLLQ/VPBROADCASTQ/VPERMQ 命令は、RSA 1024 と RSA 2048 に対応するキー長に関連する剰余乗算/冪剰余を効率良く実装するベクトル化のアプローチ

チを可能にします。OpenSSL における剰余乗算/冪剰余とインテル® AVX2 の実装の詳細については、[http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3\\_9?LI=true](http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true) (英語) を参照してください。

基本的なヒューリスティックは、冗長表現の 512/1024 ビット指数で大きな整数入力オペランドを再定式化することから始めます。例えば、1024 ビット整数は基数  $2^{29}$  と 36 “桁” を使用して表現できます。ここで各 “桁” は  $2^{29}$  未満です。このような冗長表現の桁は、ベクトルレジスターの dword スロットに配置できます。大きな整数のような冗長表現は、符号なし整数乗算の中間結果のハードウェアの粒度にわたってキャリー加算チェーンの要件を簡略化します。

冗長表現された数値を使用するインテル® AVX2 の VPMULUDQ は、十分な余地を残して 4 つの 64 ビットの中間結果を生成できます (例えば、5 つの最上位ビットが符号ビットを除いて 0)。

その場合、VPADDQ は ADC のような命令と同等の SIMD バージョンを必要とせず、加算キャリーチェーンの要件を実装するには十分です。冗長表現への変換のコスト要因や VPMULUDQ/VPADDQ チェーンの並列出力帯域幅向けの効率良いスピードアップの説明を含む、詳しい情報については前の段落で示したウェブサイトでご覧いただけます。

### 12.16.3 データ移動に関する考察

インテル® マイクロアーキテクチャー開発コード名 Haswell は、2 つの 256 ビット・ロードと 1 つの 256 ビット・ストアの `uop` を各サイクルでディスパッチできます。データ移動操作の多い既存のバイナリーは、以前の世代のマイクロアーキテクチャー向けに最適化されていれば、再コンパイルなしでこの拡張と高帯域幅の L1 や L2 キャッシュから利点を得られます。例えば、256 ビット SAXPY 計算はこれまでの世代のマイクロアーキテクチャーでは、ロード/ストアポートの数で制限されていましたが、インテル® マイクロアーキテクチャー開発コード名 Haswell では直ちに利点が得られます。

状況によっては、命令セットのマイクロアーキテクチャー上の制限との複雑な相互作用があるため、議論の余地があるかもしれません。一般的に使用される 2 つのライブラリー関数 `memcpy()` と `memset()` を新しいマイクロアーキテクチャー上に実装する最適な選択肢について考えてみましょう。

インテル® マイクロアーキテクチャー開発コード名 Haswell 上の `memcpy()` で、大きなコピー長の `memcpy` 操作を実装するため `REP MOVSB` 命令を使用することは、256 ビットのストア・データパスの利点を活用してサイクルごとに 20 バイト以上のスルーputを提供できます。コピー長が数百バイト以下の場合、`REP MOVSB` のアプローチは 12.16.3.1 節に示した 128 ビットの SIMD 手法を使用するよりも低速です。

#### 12.16.3.1 `memcpy()` を実装する SIMD ヒューリスティック

`memcpy()` に 128 ビット SIMD 命令の実装を試みるため、一般的なヒューリスティックを考えることから始めます。対象とする命令セットのレジスター幅に対する 3 つの数値要因 (デスティネーション・アドレス・アライメント、ソース・アドレス・アライメント、コピーするバイト数) を中心に議論します。

`memcpy` のデータ移動は次のフェーズに分割できます。

- Y 最初のアライメントされていない 16 バイトのコピーは、ループ中で使用するデスティネーション・アドレスのポインターが 16 バイトにアライメントされることを可能にします。これにより、後続のストアは同じように 16 バイトにアライメントされたストアを使用することができます。
- Y コピーで残されたバイトは、(a) アンロールされた 16 バイト・コピー操作の倍数と、(b) 16 バイト未満のいくつかのコピー操作を含む残余カウントに分類できます。例えば、ループ反復のオーバーヘッドを軽減するため 8 回アンロールするには、残余カウントは 1 から  $8 \times 16 - 1 = 127$  のケースを個々に考慮しなければなりません。
- Y  $8 \times 16$  にアンロールされたメインループ内部の各 16 バイトのコピー操作では、16 バイト境界にアライメントされていないソースのポインター・アドレスに対処し、16 個のデータを 16 バイトにアライメントされたデス

ディネーション・アドレスにストアする必要があります。繰り返し使用されるソースポインタが 16 バイトにアライメントされていない場合、最も効率良い手法は次の 3 命令シーケンスです。

- 16 バイト・アライメントに調整されたポインタアドレスから 16 バイトのチャンクをフェッチし、このチャンクの一部を以前の 16 バイト・アライメントからフェッチした一部で補って使用します。
- 現在のチャンクと以前のチャンクの一部の結合には PALIGNR を使用します。
- 結合した 16 バイトの新しいデータをアライメントされたデスティネーション・アドレスへストアし、この 3 命令シーケンスを繰り返します。

この 3 命令の手法は、それぞれの 16 バイト・コピー操作のフェッチ:ストアの命令比率が 1:1 となることを可能にします。

上記の手法 (特にメインループが数千バイトのデータをコピーする場合) は、ストア操作に 128 ビットのデータパスを持つインテル® マイクロアーキテクチャー Sandy Bridge<sup>†</sup> と Ivy Bridge<sup>†</sup> ではサイクルごとにおよそ 10 バイトのスループットを達成できます。この手法を拡張して広いデータパスの使用を試みると次の制限を受けます。

- Y 2x128 レーンを持つマイクロアーキテクチャーで 256 ビット VPALIGNR を使用するには、現在の 32 バイトにアライメントされた 256 ビット・フェッチの 2 つのチャンクの結合に、現在の 32 バイトにアライメントされたフェッチから 16 バイト・オフセットされたアドレスからの別のフェッチが必要になります。
  - 各バイト・コピー操作のフェッチ:ストアの比率は 2:1 になります。
  - 32 バイトにアライメントされていないフェッチ (16 バイトにアライメントされている) は、コピー操作の 64 バイトごとにキャッシュライン分割のペナルティーが科せられます。

256 ビット・ストア・データパスを持つマイクロアーキテクチャーの利点を生かす 256 ビット ISA の使用の試みは、4 命令シーケンスとキャッシュライン分割のペナルティーで相殺されました。

### 12.16.3.2 拡張された REP MOVSB を使用した memcpy() の実装

memcpy() を実装するため、拡張された REP MOVSB を使用した代替アプローチを比較するのは興味深いことです。インテル® マイクロアーキテクチャー Haswell<sup>†</sup> と Ivy Bridge<sup>†</sup> では、REP MOVSB は最適化され、ハードウェアによる pop フローを提供します。

インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge では、REP MOVSB の memcpy 実装は、数千バイトをコピーする場合 128 ビット SIMD 実装より少し良いスループットを達成します。しかし、コピー操作が数百バイト未満のサイズを扱う場合、REP MOVSB のアプローチは 12.16.3.1 節の 2 番目のセクションで説明した、明白な残余コピーの手法ほど効率良くありません。これは、残りのコピー長 1 - 127 (ジャンプテーブルや switch/case 文によりメインループの前で実施) の処理に加え、1 回もしくは 2 回の 8x16 バイト反復がハードウェアによる pop フローよりは少ない分岐のオーバーヘッドを被ります。memcpy() の 128 ビット SIMD 実装の詳細を知る方法の 1 つは、glibc などのオープンソース・ライブラリーのソースコードを参照することです。

インテル® マイクロアーキテクチャー開発コード名 Haswell 上で、大きなコピー長の memcpy 操作を実装するため REP MOVSB 命令を使用することは、256 ビットのストア・データパスの利点を活用してサイクルごとに 20 バイト以上のスループットを提供できます。コピー長が数百バイト以下の場合、REP MOVSB のアプローチは 12.16.3.1 節の残余フェーズとしてコピー長を扱うよりもまだ低速です。

### 12.16.3.3 memset() 実装の考察

memset() のインターフェイスには、デスティネーションとして 1 つのアドレスポインタがあります。これは、アライメントされた 256 ビットのストア命令を使用する際にアドレス・アライメントの管理を簡素化します。最初のアライメントされていないストアの後、デスティネーション・ポインタを 32 バイト境界に調整して、12.16.3.1 節で説明される同じ方法で残りを処理します。これには、アンロールされた 32 バイト・アライメントのストアに応じて、最小の分岐によって残りの値を扱う大きなジャンプテーブルを使用する可能性があります。メインループでは、単純に YMM レジスターを 32 バイトにアライメントされたストア操作へサイクルごとに 30 バイト近くを送り出すことができます (長さが数千バイトの場合)。ここでの制限要因は、各 256 ビット VMOVDQA ストアがストアアドレスとストア

データの  $\mu\text{op}$  フローから成り立つということです。各サイクルでストアデータ  $\mu\text{op}$  は、ポート 4 にのみディスパッチ可能です。

`memset()` を実装するため `REP STOSB` を使用すると、`memcpy()` に `REP MOVSB` を使用する場合と同様に、SIMD 実装に対してコードサイズの利点があります。インテル® マイクロアーキテクチャー開発コード名 Haswell 上で `REP STOSB` を使用する `memset()` ルーチンの実装は、256 ビット・データパスと増加した L1 データキャッシュの帯域幅 (サイクルごとに最大 32 バイト) の利点を得られます。

`REP STOSB` と 256 ビットのインテル® AVX2 を使用した `memset()` 実装のパフォーマンスを比較するには、`memset()` の呼び出しパターンを考慮する必要があります。呼び出しパターンによって、使用するパフォーマンス計測の方法が異なります。それぞれの計測方法には副作用があることがあります。

`memset()` のような単純なルーチンでしばしば使用される最も一般的な計測方法は、大きなループカウント内部で `memset()` を呼び出し、その前後で `RDTSC` 命令を使用して計測を行うものです。

`memset()` 呼び出しの間に干渉する命令ストリームが実行されず、異なるカウント値による複数の `memset()` 呼び出しが連続するパターンを計測する場合、この計測方法のバリエーションを適用できます。

上記両方の `memset()` 呼び出しのシナリオでは、分岐予測はループ実行の総サイクルの計測に影響する重要な要因となります。そのため、`RDTSC` のオーバーヘッドを最小化するため、大きなループでインテル® AVX2 が実装された `memset()` を計測すると、そのループカウントによって予測される分岐で歪んだ結果となる可能性があります。

より現実的なソフトウェア・スタックでは、`memset()` 呼び出しのパターンには次のような特性があります。

- Y `memset()` 呼び出しの間で実行される干渉する命令ストリームがあると、`memset()` 呼び出し前の分岐予測器の状態は、`memset()` 実装内の分岐シーケンス向けに事前トレーニングされたものとはなりません。
- Y `memset()` のカウント値は訂正されない傾向があります。

現実的な `memset()` 呼び出しのシナリオを考えると、`memset()` のパフォーマンスを比較する適切な計測手法は、それぞれの `memset()` の呼び出しの前後を 2 つの `RDTSC` で挟み込んで計測する方法です。

呼び出しごとに `RDTSC` で計測する手法では、`RDTSC` のオーバーヘッドが発生し、計測するループ外部で事前補正および事後評価が必要となります。呼び出しごとの計測手法は、ループを包み込んで呼び出しごとに計測するため、キャッシュ・ウォーミングを考慮する必要があるかもしれません。

計測手法に関連するスキュー要因が有効である場合、カウント値が数百バイト以下の `REP STOSB` を使用する `memset()` のパフォーマンスは、一般的にインテル® AVX2 バージョンの `memset()` 呼び出しシナリオよりも高速です。数百ものアンロールされた `memset()` 呼び出しを行う極端なシナリオにおいてのみ、数百バイト以下のカウント値をすべて使用して、`memset()` の間にある干渉する命令ストリームの影響なしで `memset()` のインテル® AVX2 バージョンは分岐予測のトレーニングの利点を生かすことができます。

### 12.16.3.4 使用するコードの前に `Memcpy/Memset` を移動

`memcpy/memset` を呼び出して準備したデータと、それを使用するその後の命令を再配置できることもあります。

```
memcpy ( pBuf, pSrc, Cnt);      // Cnt の情報といくつかのデータをコピー。
.....                          // 後続の命令シーケンスは pBuf をすぐには消費しません。
result = compute( pBuf);      // memcpy の結果をここで使用。
```

カウント (Cnt) が少なくとも 1000 バイト以上であることが明らかである場合、拡張された `REP MOVSB/STOSB` を使用すると非消費コードのコストを相殺できる利点が得られます。次のように `Cnt = 4096` の値で `memset()` を使用することで、ヒューリスティックを理解できます。



- Y memset() の 256 ビット SIMD 実装では、非消費命令シーケンスがリタイアする前に、128 の VMOVDQA と 32 バイトのストア操作を発行/実行/リタイアする必要があります。
- Y ECX = 4096 の拡張 REP STOSB は、ハードウェアによる長い `µop` フローとしてデコードされますが、1 命令としてリタイアします。memset() の結果を使用する前には、完了しなければならない多くのストアデータ操作があります。ストアデータ操作の完了はプログラム順序のリタイアとは分離されるため、非消費コードストリームの実質的な部分がストアバッファのリソースと競合しなければ、発行/実行/リタイアメントを通して処理できません。

拡張された REP MOVSB/STOSB を使用するソフトウェアは、CPUID.(EAX=07H, ECX=0):EBX.[ビット 9] が 1 であることを調査して命令が利用できることを確認する必要があります。

### 12.16.3.5 256 ビット・フェッチと 128 ビット・フェッチ

インテル® マイクロアーキテクチャー Sandy Bridge<sup>+</sup> と Ivy Bridge<sup>+</sup> では、マイクロアーキテクチャーのメモリー・パイプラインの 128 ビット・データパスの制約から見て、16 バイトにアライメントされた 2 つのロードの使用が推奨されます。

インテル® マイクロアーキテクチャー開発コード名 Haswell の 256 ビット・データパス・マイクロアーキテクチャーの利点を活用するには、256 ビット・ロードの暗黙のアライメントを考慮すべきです。メモリーから 256 ビット・データをフェッチする命令は、32 バイトのアライメントに注意する必要があります。32 バイトにアライメントされていないフェッチがキャッシュライン境界にまたがっている場合も、2 つの 16 バイト境界でアライメントされたアドレスからフェッチすることが推奨されます。

### 12.16.3.6 MULX とインテル® AVX2 命令の混在

MULX とインテル® AVX2 命令を組み合わせると、特定の一般的な計算タスクのパフォーマンスをさらに向上できます。例えば、64 ビット整数から ascii 形式への数値変換では、柔軟性のある MULX レジスター割り当て、広い YMM レジスター、そして並列係数/余剰計算のためのバックド・シフト・プリミティブ VPSRLVD から恩恵を受けられます。

例 12-40 に、1 または 2 つの有限範囲の unsigned short 整数を計算してそれぞれ 10 進数にするインテル® AVX2 命令のマクロシーケンスを示します。ここでは、モンゴメリー乗算とともに VPSRLVD 命令を使用しています。

例 12-40 並列に係数/余剰を計算するマクロ

```
static short quoTenThsn_mulplr_d[16] =
{ 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0, 0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static short mten_mulplr_d[16] = { -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1, -10, 1};
// 商 (dword 4) と余り (dword 0) を含む入力 t5 (a __m256i type) を
// 出力 y3 (__m256i) の単一桁の整数 (0-9) へ変換するマクロ、
// 両方の dword 要素 "t5" は 10^4 未満と想定され、残りの dword は 0 である必要がある。
// 出力は 8 つの単一桁の整数で、各 dword の下位バイトに配置され、dword の MS 桁は 0。
#define __ParMod10to4AVX2dw4_0( y3, t5 ) ¥
{ __m256i x0, x2; ¥
  x0 = _mm256_shuffle_epi32( t5, 0); ¥
  x2 = _mm256_mulhi_epu16(x0, _mm256_loadu_si256( (__m256i *) quoTenThsn_mulplr_d)); ¥
  x2 = _mm256_srlv_epi32( x2, _mm256_setr_epi32(0x0, 0x4, 0x7, 0xa, 0x0, 0x4, 0x7, 0xa) ); ¥
  (y3) = _mm256_or_si256(_mm256_slli_si256(x2, 6), _mm256_slli_si256(t5, 2) ); ¥
  (y3) = _mm256_or_si256(x2, y3); ¥
  (y3) = _mm256_madd_epil6(y3, _mm256_loadu_si256( (__m256i *) mten_mulplr_d) ); ¥
}
(続く)
```

```
// dword 整数 (< 10^4) を 4 つの単一桁整数 __m128i へ並列に変換
#define __ParMod10to4AVX2dw( x3, dw32 ) ¥
{ __m128i x0, x2; ¥
  x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128( dw32)); ¥
  x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d));¥
  x2 = _mm_srlv_epi32( x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) ); ¥
  (x3) = _mm_or_si128(_mm_slli_si128(x2, 6), _mm_slli_si128(_mm_cvtsi32_si128( dw32), 2) ); ¥
  (x3) = _mm_or_si128(x2, (x3));¥
  (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) );¥
}
```

例 12-41 は、ヘルパー・ユーティリティと 64 ビット符号付き整数を、MULX を使用して 63 ビット符号なし範囲の整数の商/余剰のペアへ縮小するすべてのステップを示しています。

#### 例 12-41 符号付き 64 ビット整数の変換ユーティリティ

```
#defineQWCG10to80xabcc77118461cefdull
static short quo4digComp_mulplr_d[8] = { 1024, 0, 64, 0, 8, 0, 0, 0};
static int pr_cg_10to4[8] = { 0x68db8db, 0, 0, 0, 0x68db8db, 0, 0, 0};
static int pr_1_m10to4[8] = { -10000, 0, 0, 0, 1, 0, 0, 0};
char * i64toa_avx2i( __int64 xx, char * p)
{int cnt;
  _mm256_zeroupper();
  if( xx < 0) cnt = avx2i_q2a_u63b(-xx, p);
  else cnt = avx2i_q2a_u63b(xx, p);
  p[cnt] = 0;
  return p;
}
// unsigned short (< 10^4) を ascii へ変換
__inline int ubsAvx2_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
  __m128i x0, m0, x2, x3, x4, compv;
  if( x_Lt10k < 10) { *ps = '0' + x_Lt10k; return 1; }
  x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128( x_Lt10k));
  // 除数 10, 100, 1000, 10000 の商を計算
  m0 = _mm_loadu_si128( (__m128i *) quoTenThsn_mulplr_d);
  x2 = _mm_mulhi_epu16(x0, m0);
  // u16/10, u16/100, u16/1000, u16/10000
  x2 = _mm_srlv_epi32( x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) );
  // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
  x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
  x4 = _mm_or_si128(x2, x3);
  各 dword の下位バイトに 4 つの単一桁を生成
  x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) );// ascii エンコード向けにパイア
  スを加算
  x2 = _mm_add_epi32( x4, _mm_set1_epi32( 0x30303030 ) );
  // 4 つの単一桁をパック、最上位桁から開始
  x3 = _mm_shuffle_epi8(x2, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
  if( x_Lt10k > 999 ) *(int *) ps = _mm_cvtsi128_si32( x3); return 4;
  return 3;
}
(続く)
```

```

else {
    tmp = __mm_cvtsi128_si32( x3);
    if (x_Lt10k > 99 ) {
        *((short *) (ps)) = (short ) (tmp >>8);
        ps[2] = (char ) (tmp >>24);
        return 3;
    }
    else if ( x_Lt10k > 9){
        *((short *) ps) = (short ) tmp;
        return 2;
    }
}
}
}

```

例 12-42 は、ベクトル化されたモンゴメリー乗算方式による先進的なリダクション手法を使用して、63 ビットのダイナミック・レンジを ascii 形式に数値変換するステップを示しています。

例 12-42 符号付き 63 ビット整数の変換ユーティリティ

```

unsigned avx2i_q2a_u63b (unsigned __int64 xx, char *ps)
{
    __m128i v0;
    __m256i m0, x1, x2, x3, x4, x5 ;
    unsigned __int64 xxi, xx2, lo64, hi64;
    __int64 w;
    int j, cnt, abv16, tmp, idx, u;
    // 4 桁以下を変換
    if ( xx < 10000 ) {
        j = ubsAvx2_Lt10k_2s_i2 ( (unsigned ) xx, ps); return j;
    } else if (xx < 100000000 ) { // dynamic range of xx is less than 9 digits
        // 5-8 桁の変換
        x1 = __mm256_broadcastd_epi32( __mm_cvtsi32_si128(xx)); // broadcast to every dword
        // 減少した範囲 (< 10^4) でそれぞれ商と余りを計算
        x3 = __mm256_mul_epu32(x1, __mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
        x3 = __mm256_mullo_epi32(__mm256_srli_epi64(x3, 40), __mm256_loadu_si256( (__m256i *)pr_l_m10to4));
        // 商を dw4, 余りを dw0
        m0 = __mm256_add_epi32( __mm256_castsi128_si256( __mm_cvtsi32_si128(xx)), x3);
        __ParMod10to4AVX2dw4_0( x3, m0); // 各 dw の下位バイトの 8 桁
        x3 = __mm256_add_epi32( x3, __mm256_set1_epi32( 0x30303030 ) );
        x4 = __mm256_shuffle_epi8(x3, __mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
        0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
        // 8 つの 1 桁整数をパックして最初の 8 バイトと残りはゼロに設定
        x4 = __mm256_permutevar8x32_epi32( x4, __mm256_setr_epi32(0x4, 0x0, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1) );
        tmp = __mm256_movemask_epi8( __mm256_cmpgt_epi8(x4, __mm256_set1_epi32( 0x30303030 ) ) );
        __BitScanForward((unsigned long *) &idx, tmp);
        cnt = 8 -idx; // 出力へ書き出すゼロ以外の桁数
    } else { // 9-12 桁の変換
        lo64 = __mulx_u64(xx, (unsigned __int64) QWCG10to8, &hi64);
        hi64 >>= 26;
        xxi = __mulx_u64(hi64, (unsigned __int64)100000000, &xx2);
        lo64 = (unsigned __int64)xx - xxi;
    }
    (続く)
}

```

```

if( hi64 < 10000) { // 桁 12-9 を最初に行う
    __ParMod10to4AVX2dw(v0, hi64);
    v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
    // 12 桁未満の下位 8 桁の変換を継続
    x5 = _mm256_setzero_si256( );
    x5 = _mm256_castsi128_si256( _mm_cvtsi32_si128(lo64));
    x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(lo64)); // すべての dword ヘブロードキャスト
    x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
    x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
    m0 = _mm256_add_epi32( x5, x3); // 商を dw4, 余りを dw0 に設定
    __ParMod10to4AVX2dw4_0( x3, m0);
    x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
    x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
    0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
    x5 = _mm256_castsi128_si256( _mm_shuffle_epi8( v0, _mm_setr_epi32(0x80808080, 0x80808080,
    0x0004080c, 0x80808080) ));
    x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x2, 0x4, 0x0, 0x1,
    0x1, 0x1, 0x1, 0x1) );
    tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 ) ) );
    _BitScanForward((unsigned long *) &idx, tmp);
    cnt = 12 -idx;
} else { // 12 桁以上の入力値を処理
    cnt = 0;
    if ( hi64 > 100000000) { // 入力値が 16 桁以上の場合
        xxi = _mulx_u64(hi64, (unsigned __int64) QWCG10to8, &xx2) ;
        abv16 = xx2 >>26;
        hi64 -= _mulx_u64((unsigned __int64) abv16, (unsigned __int64) 100000000, &xx2);
        __ParMod10to4AVX2dw(v0, abv16);
        v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
        v0 = _mm_shuffle_epi8(v0, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
        tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(v0, _mm_set1_epi32( 0x30303030 ) ) );
        _BitScanForward((unsigned long *) &idx, tmp);
        cnt = 4 -idx;
    }
    // 下位 16 個の数値を変換
    x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(hi64)); // broadcast to every dword
    x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
    x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
    m0 = _mm256_add_epi32( _mm256_castsi128_si256( _mm_cvtsi32_si128(hi64)), x3);
    __ParMod10to4AVX2dw4_0( x3, m0);
    x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
    x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
    0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
    x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128(lo64)); // broadcast to every dword
    x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
    x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
    m0 = _mm256_add_epi32( _mm256_castsi128_si256( _mm_cvtsi32_si128(hi64)), x3);
    __ParMod10to4AVX2dw4_0( x3, m0);
    x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
    (続く)

```

```

x5 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080,
0x80808080, 0x80808080, 0x0004080c, 0x80808080) );
x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x4, 0x0, 0x6, 0x2,
0x1, 0x1, 0x1, 0x1) );
cnt += 16;
if (cnt <= 16) {
    tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 ) ) );
    _BitScanForward((unsigned long *) &idx, tmp);
    cnt -= idx;
}
}
w = _mm_cvtsi128_si64( _mm256_castsi256_si128(x4));
switch(cnt) {
    case5:*ps++ = (char) (w >>24); *(unsigned *) ps = (w >>32);
    break;
    case6:*(short *)ps = (short) (w >>16); *(unsigned *) (&ps[2]) = (w >>32);
    break;
    case7:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
    *(unsigned *) (&ps[3]) = (w >>32);
    break;
    case 8: *(long long *)ps = w;
    break;
    case9:*ps++ = (char) (w >>24); *(long long *) (&ps[0]) = _mm_cvtsi128_si64(
    _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case10:*(short *)ps = (short) (w >>16);
    *(long long *) (&ps[2]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
    *(long long *) (&ps[3]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case 12: *(unsigned *)ps = w; *(long long *) (&ps[4]) = _mm_cvtsi128_si64(
    _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
    break;
    case13:*ps++ = (char) (w >>24); *(unsigned *) ps = (w >>32);
    *(long long *) (&ps[4]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
    break;
    case14:*(short *)ps = (short) (w >>16); *(unsigned *) (&ps[2]) = (w >>32);
    *(long long *) (&ps[6]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
    break;
    case15:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
    *(unsigned *) (&ps[3]) = (w >>32);
    *(long long *) (&ps[7]) = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
    break;
    case 16: _mm_storeu_si128( (__m128i *) ps, _mm256_castsi256_si128(x4));
    break;
    case17:u = _mm_cvtsi128_si64(v0); *ps++ = (char) (u >>24);
    _mm_storeu_si128( (__m128i *) &ps[0], _mm256_castsi256_si128(x4));
    break;
    case18:u = _mm_cvtsi128_si64(v0); *(short *)ps = (short) (u >>16);
    _mm_storeu_si128( (__m128i *) &ps[2], _mm256_castsi256_si128(x4));
    break;
}
(続く)

```

```

    case19: u = _mm_cvtsi128_si64(v0); *ps = (char) (u >>8); *(short *) (&ps[1]) = (short) (u >>16);
    _mm_storeu_si128( (__m128i *) &ps[3], _mm256_castsi256_si128(x4));
    break;
    case20: u = _mm_cvtsi128_si64(v0); *(unsigned *)ps = (short) (u);
    _mm_storeu_si128( (__m128i *) &ps[4], _mm256_castsi256_si128(x4));
    break;
}
return cnt;
}

```

3/9/17 出力桁のダイナミック・レンジにわたってインテル® AVX2 バージョンの数値変換は、標準ライブラリー実装の入力ごとの 85/260/560 サイクルと比較すると、およそ入力ごとに 23/57/54 サイクルです。

上記の手法は、バイナリー-整数-10 進 (BID) エンコード IEEE-754-2008 10 進浮動小数点形式など、他のライブラリーの数値変換にも適用できます。BID-128 形式において、係数 10<sup>16</sup> でモンゴメリー乗算を行うため事前に計算された 256 ビット定数を使用して別の範囲のリダクション・ステージを追加することで例 12-42 を適用できます。256 ビットの定数を構築する手法は、『インテル® 64 および IA-32 アーキテクチャー最適化リファレンス・マニュアル』の第 10 章「テキスト処理/字句解析/構文解析における SSE.4.2 と SIMD プログラミング」で詳しく説明されています。

### 12.16.4 Gather 命令に関する考察

VGATHER 命令ファミリーは、ベースアドレスからの相対オフセットを含むベクトル・インデックス・レジスターで指定される複数のデータ要素をフェッチします。Haswell<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサは最初に VGATHER 命令を実装し、単一命令で複数の  $\mu\text{op}$  が実行されます。Broadwell<sup>†</sup> マイクロアーキテクチャーでは、VGATHER 命令ファミリーのスループットがかなり改善されています (表 C-5 を参照)。

データ編成とアクセスパターンによっては、VGATHER 命令を使用することなく、より高速な少ない  $\mu\text{op}$  で等価なコードシーケンスを作成することができます (12.5.1 節を参照)。例 12-43 に、インテル® マイクロアーキテクチャー開発コード名 Haswell 上で VGATHER 命令がパフォーマンスの利点をもたらさない利用状況を示します。

例 12-43 VGATHER を使用しない手法向けのアクセスパターン

アクセスパターン	推奨される命令の選択
連続した要素	通常の SIMD ロード (MOVAPS/MOVUPS, MOVDQA/MOVDQU)
4 要素以下	通常の SIMD ロード + スロットを再配置する水平データ移動
小さなストライド	隣接する要素をすべてロード + VMOVUPD YMM0, [シーケンシャルな要素] VPERMQ YMM1, YMM0, 0x08 // 偶数要素 VPERMQ YMM2, YMM0, 0x0d // 奇数要素
転置	通常の SIMD ロード + shuffle/permute/blend で列へ転置
冗長な要素	一度のロード + shuffle/blend/logical (レジスター内でデータのベクトルを構築) この場合、 $\text{result}[i] = \text{x}[\text{index}[i]] + \text{x}[\text{index}[i+1]]$ 、複数の VGATHER を使用するために以下の手法が望ましい ymm0 <- VGATHER ( x[index[k]] ); // 8 つの要素をフェッチ ymm1 <- VBLEND( VPERM( ymm0), VBROADCAST( x[index[k+8]])); ymm2 <- VPADD( ymm0, ymm1);

VGATHER 命令を使用すると、VGATHER のレイテンシーとスループットを相殺するのを妨げない手法や、フェッチ操作を消費するコードのデスティネーション・レジスターへのフェッチの前に移動することによって、コードサイズを減らし高速に実行できます。例 12-44 には、インテル® マイクロアーキテクチャー開発コード名 Haswell 上で VGATHER を使用することで利点を得られるいくつかのパターンを示します。

VGATHER を使用する一般的なヒント:

- Y VGATHER 命令でより多くの要素を収集すると、VGATHER のレイテンシーとスループットを相殺できるため、等価な VGATHER を使用しないフローよりもパフォーマンス上の利点が得られます。例えば、256 ビット VGATHER のレイテンシーは、等価な 128 ビット VGATHER の半分以下であり、128 ビット VGATHER を 2 つ使用するよりも利点があります。また、データ要素サイズよりも大きなインデックス・サイズを使用すると、レイテンシーが短くなるだけでなくレジスタスロットを半分しか利用しません。したがって、VGATHER の dword インデックス形式は、dword または float 値がフェッチされる場合 qword インデックスよりも適しています。
- Y 消費するコードの手前に VGATHER を配置するのは有効です。
- Y VGATHER は、デスティネーションにある前値と (マスクされていない) 収集された要素をマージします。そのため、デスティネーションの前値をマージする必要がない場合 (例えば、要素がマスクされていない)、デスティネーション・レジスターへの前の値の書き込みと VGATHER 命令の依存関係を排除する利点があります (VXOR 命令でレジスターをゼロにする)。

例 12-44 VGATHER 手法に適したアクセスパターン

アクセスパターン	命令の選択
不明なマスクと 4 つ以上の要素	<p>条件付きで要素を収集するコードは、一般に VGATHER 命令なしではベクトル化できないか、データ依存の分岐予測ミスにより相対的に低パフォーマンスとなります。</p> <p>データ依存の分岐がある C コード:  <code>if (condition[i] &gt; 0) { result[i] = x[index[i]] }</code></p> <p>インテル® AVX2 の等価なシーケンス:  <code>YMM0 &lt;- VPCMPGT (condition, zeros) // ベクトルマスクを計算</code>  <code>YMM2 &lt;- VGATHER (x[YMM1], YMM0) // addr=x[YMM1], mask=YMM0</code></p>
8 つの要素でベクトル化されたインデックス計算	<p>インデックスを生成するためにベクトル化された計算には、VGATHER 命令の機能と相乗効果があります。</p> <p>C コードの断片:  <code>x[index1[i] + index2[i]]</code></p> <p>インテル® AVX2 の等価なシーケンス:  <code>YMM0 &lt;- VPADD (index1, index2) // ベクトル・インデックスを計算</code>  <code>YMM1 &lt;- VGATHER (x[YMM0], mask) // addr=x[YMM0]</code></p>

VGATHER 命令のパフォーマンスを等価な複数の命令で構成されるギャザー操作フローと比較すると、(1) 基本となるアルゴリズムの違い、(2) データ編成の違い、および (3) 等価なフローの有効性などによって変化します。パフォーマンスが重要なアプリケーションでは、両方の選択肢を評価してから採用を決定することを推奨します。

GATHER 命令のスループットは、Broadwell<sup>+</sup> から Skylake<sup>+</sup> マイクロアーキテクチャーでも継続して改善されています。この様子を図 12-4 に示します。

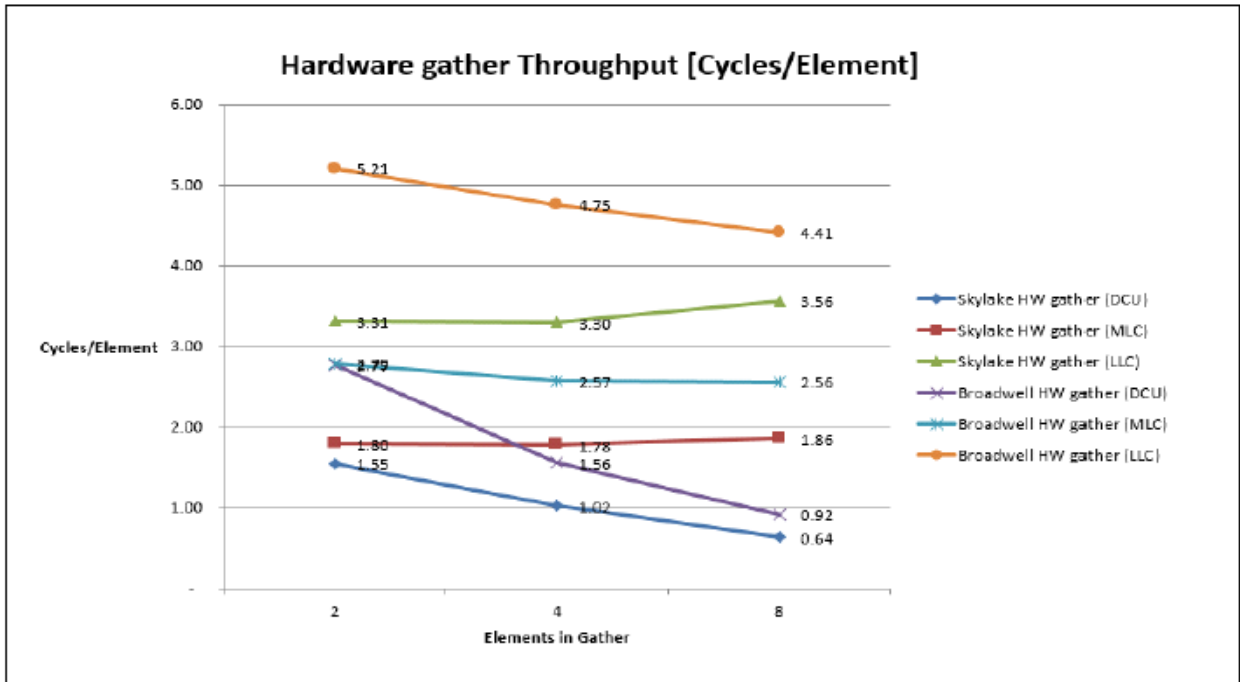


図 12-4 Gather 命令のスループットの比較

例 12-45 は、VGATHER 命令と同等なソフトウェア実装のアセンブリー・シーケンスを示しています。これは、個別要素を挿入した場合の、ハードウェアのギャザー命令とソフトウェア実装のギャザーシーケンスのトレードオフを比較するのに利用できます。

例 12-45 完全なマスクの VPGATHERD と等価なソフトウェアによるインテル® AVX シーケンス

```

mov eax, [rdi] // index0 をロード
vmovd xmm0, [rsi+4*rax] // element0 をロード
mov eax, [rdi+4] // index1 をロード
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x1 // element1 をロード
mov eax, [rdi+8] // index2 をロード
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x2 // element2 をロード
mov eax, [rdi+12] // index3 をロード
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x3 // element3 をロード
mov eax, [rdi+16] // index4 をロード
vmovd xmm1, [rsi+4*rax] // element4 をロード
mov eax, [rdi+20] // index5 をロード
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x1 // element5 をロード
mov eax, [rdi+24] // index6 をロード
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x2 // element6 をロード
mov eax, [rdi+28] // index7 をロード
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x3 // element7 をロード
vinsertil28 ymm0, ymm0, xmm1, 1 // ymm0 に結果を生成
    
```



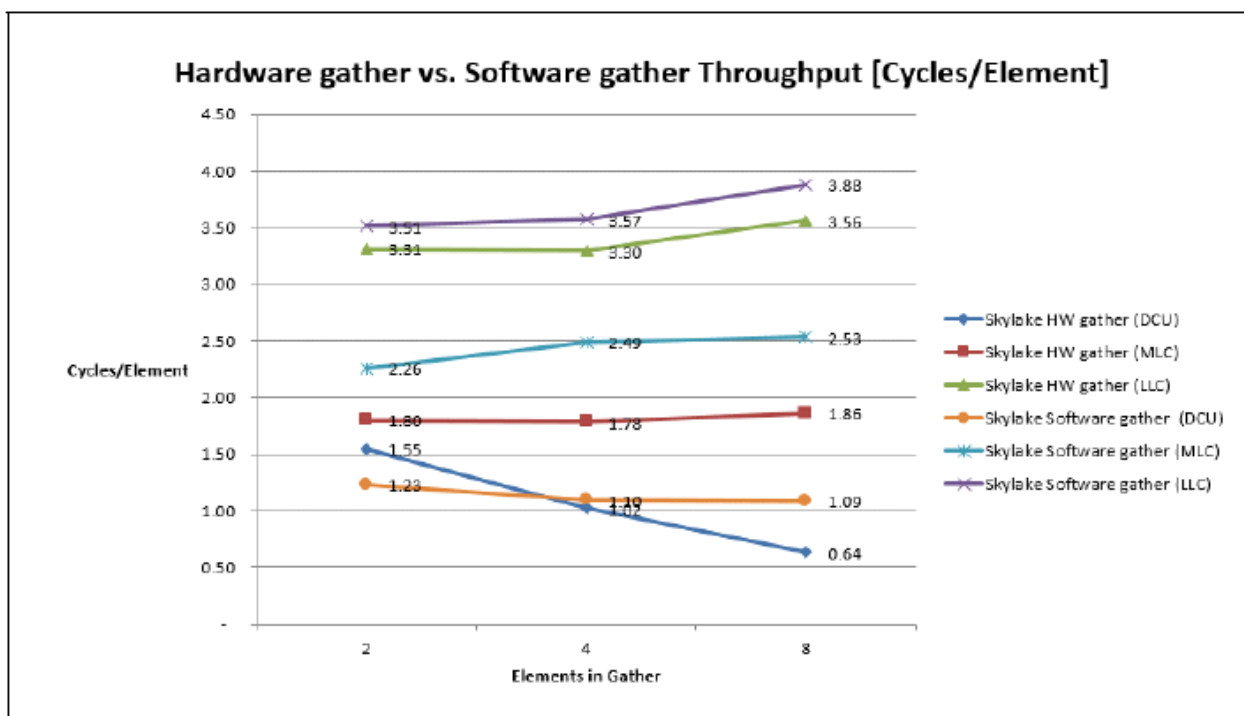


図 12-5 Skylake<sup>+</sup> マイクロアーキテクチャーにおけるハードウェア GATHER とソフトウェア・シーケンスの比較

図 12-5 は、Skylake<sup>+</sup> マイクロアーキテクチャーで、データを供給するキャッシュ局所性関数として、VPGATHERD 命令とソフトウェアによるギャザーシーケンスを使用した要素ごとのスループットを比較したものです。命令あたり 2 つのデータ要素のハードウェア GATHER を使用する場合を除き、Skylake<sup>+</sup> マイクロアーキテクチャーではギャザー命令はソフトウェア・シーケンスよりも優れています。

データがメモリーからローカルに供給される場合、ソフトウェア・シーケンスはハードウェア GATHER 命令よりもうまく動作します。

### 11.16.4.1 スライドロード

この節では、構造体配列 (AOS) から配列構造体 (SOA) への転置を処理する実装とハードウェア GATHER 命令の利用を比較します。コードは、複素数配列内の実数と虚数要素を 2 つの異なる配列へ分離します。

C のコード:

```
for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[i].real;
    Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

例 12-46 AOS から SOA への変換の代替

1: スカラーコード	2: VINSRT+VSHUFPS を使用した インテル® AVX	3: VPGATHERD 使用した インテル® AVX2
<pre> loop: lea eax, ptr [r10+r10*1] movsxd rax, eax inc r10d mov r11d, dword ptr [rsi+rax*8] mov dword ptr [rcx+rax*4], r11d mov r11d, dword ptr [rsi+rax*8+0x4] mov dword ptr [rdx+rax*4], r11d mov r11d, dword ptr [rsi+rax*8+0x8] mov dword ptr [rcx+rax*4+0x4], r11d mov r11d, dword ptr [rsi+rax*8+0xc] mov dword ptr [rdx+rax*4+0x4], r11d cmp r10d, r8d jl loop                     </pre>	<pre> loop: vmovdqu xmm0, xmmword ptr [r10+rcx*8] vmovdqu xmm1, xmmword ptr [r10+rcx*8+0x10] vmovdqu xmm4, xmmword ptr [r10+rcx*8+0x40] vmovdqu xmm5, xmmword ptr [r10+rcx*8+0x50] vinserti128 ymm2, ymm0, xmmword ptr [r10+rcx*8+0x20], 0x1 vinserti128 ymm3, ymm1, xmmword ptr [r10+rcx*8+0x30], 0x1 vinserti128 ymm6, ymm4, xmmword ptr [r10+rcx*8+0x60], 0x1 vinserti128 ymm7, ymm5, xmmword ptr [r10+rcx*8+0x70], 0x1 add rcx, 0x10 vshufps ymm0, ymm2, ymm3, 0x88 vshufps ymm1, ymm2, ymm3, 0xdd vshufps ymm4, ymm6, ymm7, 0x88 vshufps ymm5, ymm6, ymm7, 0xdd vmovups ymmword ptr [r9], ymm0 vmovups ymmword ptr [r8], ymm1 vmovups ymmword ptr [r9+0x20], ymm4 vmovups ymmword ptr [r8+0x20], ymm5 add r9, 0x40 add r8, 0x40 cmp rcx, rsi jl loop                     </pre>	<pre> loop: lea r11, ptr [r10+rcx*8] vpxor ymm5, ymm5, ymm5 add rcx, 0x8 vpxor ymm6, ymm6, ymm6 vmovdqa ymm3, ymm0 vmovdqa ymm4, ymm0 vpgatherdd ymm5, ymmword ptr [r11+ymm2*4], ymm3 vpgatherdd ymm6, ymmword ptr [r11+ymm1*4], ymm4 vmovdqu ymmword ptr [r9], ymm5 vmovdqu ymmword ptr [r8], ymm6 add r9, 0x20 add r8, 0x20 cmp rcx, rsi jl loop                     </pre>

ストライド・アクセス・パターンのインテル® AVX ソフトウェア・シーケンスは、複数の要素をロードしてシャッフルで  
きるさらに最適な手法です。

表 12-10 ストライド・アクセス・パターンと AOS および SOA の比較

マイクロアーキテクチャー	スカラー	VPGATHERD	インテル® AVX の VINSRTF128/VSHUFFLEPS
Broadwell†	1X	1.7X	4.8X
Skylake†	1X	2.7X	4.9X

## 12.16.4.2 隣接したロード

この節では、AOS から SOA への転置を処理する各種実装とハードウェア GATHER 命令の利用を比較します。この場合、AOS データはシーケンシャルにはロードされずインデックス配列が使用されます。

C のコード:

```
for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[Index_buffer[i]].real;
    Imaginary_buffer[i] = Complex_buffer[Index_buffer[i]].imag;
}
```

例 12-47 非ストライドの AOS から SOA

インテル® AVX2 の GATHERPD	インテル® AVX の VINSRTF128 /UNPACK
<pre>loop: vmovdqu ymm1, ymmword ptr [rsi+rdx*4] vpadd ymm3, ymm1, ymm1 vxorpd ymm5, ymm5, ymm5 vmovdqa ymm2, ymm0 vxorpd ymm6, ymm6, ymm6 vmovdqa ymm4, ymm0 vxorpd ymm10, ymm10, ymm10 vmovdqa ymm7, ymm0 vxorpd ymm11, ymm11, ymm11 vmovdqa ymm9, ymm0 vextracti128 xmm8, ymm3, 0x1 vgatherdpd ymm6, ymmword ptr[r8+xmm8*8], ymm4 vgatherdpd ymm5, ymmword ptr[r8+xmm3*8], ymm2 vmovupd ymmword ptr [rax+rdx*8], ymm10 vmovupd ymmword ptr [rax+rdx*8+0x20], ymm11 add rdx, 0x8 cmp rdx, r11 jnb loop</pre>	<pre>loop: movsxd r10, dword ptr [rdx+rsi*4] shl r10, 0x4 movsxd r11, dword ptr [rdx+rsi*4+0x8] shl r11, 0x4 vmovupd xmm0, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0x4] shl r10, 0x4 vinsertf128 ymm2, ymm0, xmmword ptr [r9+r11*1], 0x1 vmovupd xmm1, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0xc] shl r10, 0x4 vinsertf128 ymm3, ymm1, xmmword ptr [r9+r10*1], 0x1 movsxd r10, dword ptr [rdx+rsi*4+0x10] shl r10, 0x4 vunpcklpd ymm4, ymm2, ymm3 vunpckhpd ymm5, ymm2, ymm3 vmovupd ymmword ptr [rcx], ymm4 vmovupd xmm6, xmmword ptr [r9+r10*1] vmovupd ymmword ptr [rax], ymm5 movsxd r10, dword ptr [rdx+rsi*4+0x18] shl r10, 0x4 vinsertf128 ymm8, ymm6, xmmword ptr [r9+r10*1], 0x1 movsxd r10, dword ptr [rdx+rsi*4+0x14] shl r10, 0x4 vmovupd xmm7, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0x1c] add rsi, 0x8 shl r10, 0x4 vinsertf128 ymm9, ymm7, xmmword ptr [r9+r10*1],</pre>

	<pre> 0x1 vunpcklpd ymm10, ymm8, ymm9 vunpckhpd ymm11, ymm8, ymm9 vmovupd ymmword ptr [rcx+0x20], ymm10 add rcx, 0x40 vmovupd ymmword ptr [rax+0x20], ymm11 add rax, 0x40 cmp rsi, r8 jl loop                 </pre>
--	--

AOS から SOA へのストライドなしの通常のアクセスパターンで、VINSERTF128 と複数の要素をインターリーブしてパックするインテル® AVX のコードシーケンスは、より適切であると考えられます。

表 12-11 インデックスによる AOS から SOA への転置の比較

マイクロアーキテクチャー	VPGATHERPD	インテル® AVX の VINSRTF128/VUNPCK*
Broadwell <sup>†</sup>	1X	1.4X
Skylake <sup>†</sup>	1.3X	1.7X

### 12.16.5 インテル® MMX® 命令のスループットの制限によるインテル® AVX2 への変換方法

Skylake<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサでは、インテル® MMX® 命令セットの機能性はこれまでの世代から変更されていません。しかし、多くのインテル® MMX® 命令は、これまでのマイクロアーキテクチャーと比べると、1 つのポートでのみ実行されるため、命令のスループットが半分に抑制されます。スループットが抑制されるインテル® MMX® 命令には次のものがあります。

- ÿ PADD[S/B/W], PADDUS[B/W], PSUB[S/B/W], PSUBUS[B/W]
- ÿ PCMPGT[B/W/D], PCMPEQ[B/W/D]
- ÿ PMAX[UB/SW], PMIN[UB/SW]
- ÿ PAVG[B/W], PABS[B/W/D], PSIGN[B/W/D]

インテル® MMX® 命令のスループット減少への対策として、アセンブリーや組み込み関数のコードをインテル® AVX2 命令を使用するように変換することで劇的なパフォーマンスの向上がもたらされます。例 12-48 に、インテル® AVX2 と等価なインテル® MMX® 命令を使用したアセンブリーのシーケンスを示します。Skylake<sup>†</sup> マイクロアーキテクチャーでは、例 12-48 のインテル® MMX® コードは Broadwell<sup>†</sup> マイクロアーキテクチャーに対しておよそ半分のスピードで実行されます。

これは、PMAXSW/PMINSW のスループットが単一ポートによる実行制限のため、半分に減少したことによります。同じタスクが等価なインテル® AVX2 シーケンスで実装されると、Skylake<sup>†</sup> マイクロアーキテクチャー上のインテル® AVX2 コードのパフォーマンスは、Broadwell<sup>†</sup> マイクロアーキテクチャーで実行されるインテル® MMX® コードの 3.9 倍となります。

例 12-48 スルーブットが減少したインテル® MMX® シーケンスをインテル® AVX2 に変換する

インテル® MMX® コード	インテル® AVX2 コード
<pre> mov rax, pIn mov rbx, pOut mov r8, len xor rcx, rcx mov rcx, 8 movq mm0, [rax] movq mm1, [rax + 8] movq mm2, mm0 movq mm3, mm1 cmp rcx, r8 jge end loop: movq mm4, [rax + 2*rcx] movq mm5, [rax + 2*rcx + 8] pmaxsw mm0, mm4 pmaxsw mm1, mm5 pminsw mm2, mm4 pminsw mm3, mm5 add rcx, 8 cmp rcx, r8 j1 loop end: //リダクション pmaxsw mm0, mm1 pshufw mm1, mm0, 0xE pmaxsw mm0, mm1 pshufw mm1, mm0, 1 pmaxsw mm0, mm1 pminsw mm2, mm3 pshufw mm3, mm2, 0xE pminsw mm2, mm3 pshufw mm3, mm2, 1 pminsw mm2, mm3 movd eax, mm0 mov WORD PTR [rbx], ax movd eax, mm2 mov WORD PTR [rbx + 2], ax emms </pre>	<pre> mov rax, pIn mov rbx, pOut mov r8, len xor rcx, rcx mov rcx, 32 vmovdqu ymm0, [rax] vmovdqu ymm1, [rax + 32] vmovdqu ymm2, ymm0 vmovdqu ymm3, ymm1 cmp rcx, r8 jge end loop: vmovdqu ymm4, ymmword ptr [rax + 2*rcx] vmovdqu ymm5, ymmword ptr [rax + 2*rcx + 32] vpmaxsw ymm0, ymm0, ymm4 vpmaxsw ymm1, ymm1, ymm5 vpminsw ymm2, ymm2, ymm4 vpminsw ymm3, ymm3, ymm5 add rcx, 32 cmp rcx, r8 j1 loop end: //リダクション vpmaxsw ymm0, ymm0, ymm1 vextracti128 xmm1, ymm0, 1 vpmaxsw xmm0, xmm0, xmm1 vpshufd xmm1, xmm0, 0xe vpmaxsw xmm0, xmm0, xmm1 vpshufw xmm1, xmm0, 0xe vpmaxsw xmm0, xmm0, xmm1 vpshufw xmm1, xmm0, 1 vpmaxsw xmm0, xmm0, xmm1 vmovd eax, xmm0 mov WORD PTR [rbx], ax vpminsw ymm2, ymm2, ymm3 vextracti128 xmm1, ymm2, 1 vpminsw xmm2, xmm2, xmm1 vpshufd xmm1, xmm2, 0xe vpminsw xmm2, xmm2, xmm1 vpshufw xmm1, xmm2, 0xe vpminsw xmm2, xmm2, xmm1 vpshufw xmm1, xmm2, 1 vpminsw xmm2, xmm2, xmm1 vmovd eax, xmm2 mov WORD PTR [rbx + 2], ax </pre>

## 第 13 章 インテル® AVX-512 向けの最適化

Skylake Server<sup>†</sup> マイクロアーキテクチャーと Knights Landing<sup>†</sup> マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサでサポートされるインテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512) は、次のような 512 ビット命令セットの拡張を含んでいます。

- Y インテル® AVX-512 基本 (F)
  - 512 ビット・ベクトル幅
  - 32 個の 512 ビット長ベクトルレジスター
  - データのエキスパンドとコンプレス命令
  - 三値論理命令
  - 8 個の 64 ビット長マスクレジスター
  - 2 つのソースのレーン間のパーミュート命令
  - スキャッター命令
  - 組込みブロードキャスト/丸め
  - 超越関数のサポート
- Y インテル® AVX-512 競合検出命令 (CD)
- Y インテル® AVX-512 指数および逆数命令 (ERI)
- Y インテル® AVX-512 プリフェッチ命令 (PFI)
- Y インテル® AVX-512 バイトおよびワード命令 (BW)
- Y インテル® AVX-512 ダブルワードおよびクアッドワード命令 (DQ)
  - 新しい QWORD の計算と変換命令
- Y インテル® AVX-512 ベクトル長の拡張 (VL)

図 13-1 は、2 つのプロセッサ・ファミリーの異なる拡張を示しています。この章のパフォーマンス・レポートは、データ・キャッシュ・ユニット (DCU) に存在するデータの測定を基にしています。Skylake Server<sup>†</sup> サーバースystem では、インテル® ターボ・ブースト・テクノロジーと Intel SpeedStep® テクノロジーは無効化され、コアとアンコアの周波数は 1.80GHz に設定されています。この固定周波数の構成は、他の要因によるコード変更に影響を分かりやすくするために導入されています。

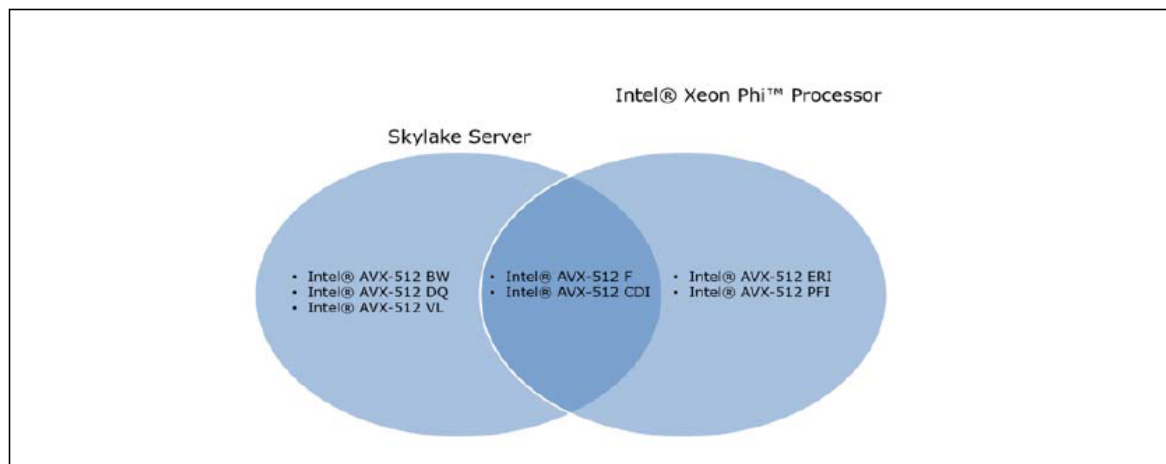


図 13-1 Skylake Server<sup>†</sup> と Knights Landing<sup>†</sup> マイクロアーキテクチャーでサポートされるインテル® AVX-512 拡張

## 13.1 インテル® AVX-512 とインテル® AVX2 コーディングの基礎

ほとんどの場合、インテル® AVX-512 のパフォーマンス向上は 512 ビット幅のレジスターによってもたらされます。この節では、インテル® AVX2 とインテル® AVX-512 コード間の相違点を示し、インテル® AVX2 からインテル® AVX-512 へ簡単に移行する方法を説明します。最初の項では組込み関数、次の項ではアセンブリ・コードの変換について説明します。以下の節では、コード変換を行う際に考慮して扱うべき高度な視点をハイライトします。

次の項の例は、デカルト座標系回転を実装しています。デカルト座標系回転のポイントは、(x, y) のペアによって表されます。次の図は、角度  $\theta$  による (x', y') のデカルト回転 (x, y) を示しています。

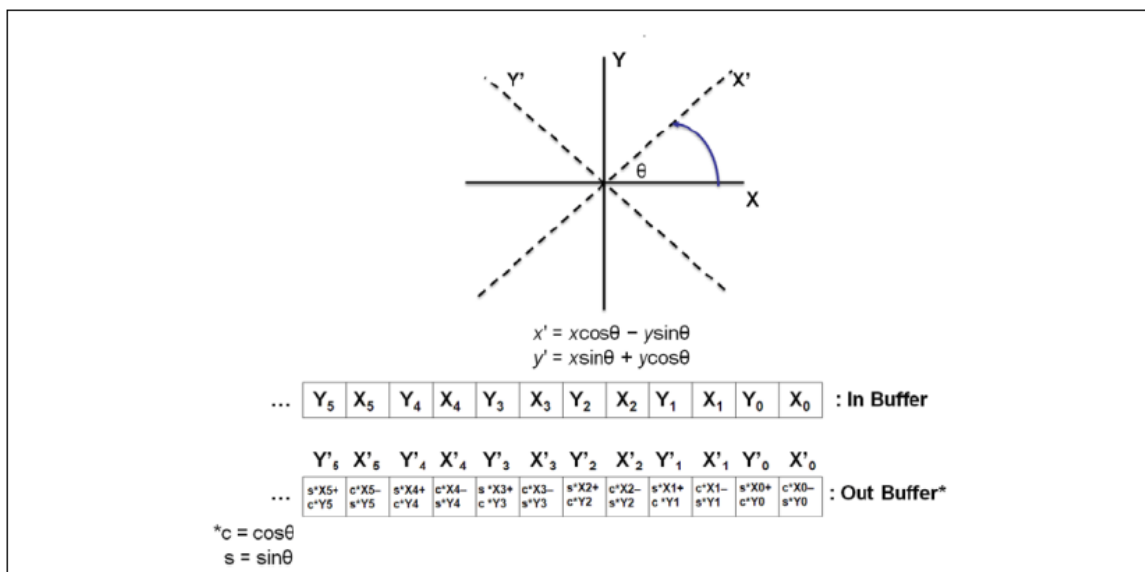


図 13-2 デカルト回転

### 13.1.1 組込み関数によるコーディング

次のインテル® AVX2 とインテル® AVX-512 の比較では、インテル® AVX2 の組込み関数で記述されたコードシーケンスをインテル® AVX-512 へ変換する方法を示しています。この例は、インテル® AVX 命令形式、64 バイト ZMM レジスター、64 バイト・データ境界での静的および動的メモリの割り当て、および ZMM レジスター内の 16 個の浮動小数点を表現する C のデータ型を使用しています。この変換を行うガイドラインを以下に示します。

- Y 静的および動的に割り当てられているバッファを 64 バイトにアライメントします。
- Y 定数向けの補助バッファサイズを倍にします。
- Y "\_\_mm256\_組込み関数名" のプリフィクスを "\_\_mm512\_" に変更します。
- Y 変数のデータ型名を \_\_m256 から \_\_m512 へ変更します。
- Y ループの反復回数を半分にします (倍のストライド長に)。

例 13-1 組み込み関数を使用したデカルト座標変換

インテル® AVX2 組み込み関数のコード	インテル® AVX-512 組み込み関数のコード
<pre> #include &lt;immintrin.h&gt; int main() { int len = 3200; // 32 バイト境界で動的にメモリーを割り当て // float* pInVector = (float *) _mm_malloc(len*sizeof(float),32); float* pOutVector = (float *) _mm_malloc(len*sizeof(float),32);  // データの初期化 for (int i=0; i&lt;len; i++) pInVector[i] = 1;  float cos_teta = 0.8660254037; float sin_teta = 0.5;  // 8 つの float を 32 バイトのアライメントで静的に メモリー割り当て __declspec(align(32)) float cos_sin_teta_vec[8] = {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta};  __declspec(align(32)) float sin_cos_teta_vec[8] = {sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta};  // __m256 データ型は、Ymm レジスターの // 8 つの float 要素を表します __m256 Ymm_cos_sin = _mm256_load_ps(cos_sin_teta_vec);  // インテル® AVX2 の 256 ビットの // パックド単精度ロード __m256 Ymm_sin_cos = _mm256_load_ps(sin_cos_teta_vec); __m256 Ymm0, Ymm1, Ymm2, Ymm3; </pre>	<pre> #include &lt;immintrin.h&gt; int main() { int len = 3200; // 64 バイト境界で動的にメモリーを割り当て // float* pInVector = (float *) _mm_malloc(len*sizeof(float),64); float* pOutVector = (float *) _mm_malloc(len*sizeof(float),64);  // データの初期化 for (int i=0; i&lt;len; i++) pInVector[i] = 1;  float cos_teta = 0.8660254037; float sin_teta = 0.5;  // 16 個の float を 64 バイトのアライメントで静的に メモリー割り当て __declspec(align(64)) float cos_sin_teta_vec[16] = {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta};  __declspec(align(64)) float sin_cos_teta_vec[16] = {sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta};  // __m512 データ型は、Zmm レジスターの // 16 個の float 要素を表します __m512 Zmm_cos_sin = _mm512_load_ps(cos_sin_teta_vec);  // インテル® AVX-512 の 512 ビットの // パックド単精度ロード __m512 Zmm_sin_cos = _mm512_load_ps(sin_cos_teta_vec); __m512 Zmm0, Zmm1, Zmm2, Zmm3; </pre>



<pre>// 2 回分アンロールされたループで 16 個の要素を処理 // for(int i=0; i&lt;len; i+=16) { Ymm0 = _mm256_load_ps(pInVector+i); Ymm1 = _mm256_moveldup_ps(Ymm0); Ymm2 = _mm256_movehdup_ps(Ymm0); Ymm2 = _mm256_mul_ps(Ymm2, Ymm_sin_cos); Ymm3 = _mm256_fmaddsub_ps(Ymm1, Ymm_cos_sin, Ymm2); _mm256_store_ps(pOutVector + i, Ymm3);  Ymm0 = _mm256_load_ps(pInVector+i+8); Ymm1 = _mm256_moveldup_ps(Ymm0); Ymm2 = _mm256_movehdup_ps(Ymm0); Ymm2 = _mm256_mul_ps(Ymm2, Ymm_sin_cos); Ymm3 = _mm256_fmaddsub_ps(Ymm1, Ymm_cos_sin, Ymm2); _mm256_store_ps(pOutVector+i+8, Ymm3); } _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>	<pre>// 2 回分アンロールされたループで 32 個の要素を処理 // for(int i=0; i&lt;len; i+=32) { Zmm0 = _mm512_load_ps(pInVector+i); Zmm1 = _mm512_moveldup_ps(Zmm0); Zmm2 = _mm512_movehdup_ps(Zmm0); Zmm2 = _mm512_mul_ps(Zmm2, Zmm_sin_cos); Zmm3 = _mm512_fmaddsub_ps(Zmm1, Zmm_cos_sin, Zmm2); _mm512_store_ps(pOutVector + i, Zmm3);  Zmm0 = _mm512_load_ps(pInVector+i+16); Zmm1 = _mm512_moveldup_ps(Zmm0); Zmm2 = _mm512_movehdup_ps(Zmm0); Zmm2 = _mm512_mul_ps(Zmm2, Zmm_sin_cos); Zmm3 = _mm512_fmaddsub_ps(Zmm1, Zmm_cos_sin, Zmm2); _mm512_store_ps(pOutVector+i+16, Zmm3); } _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>
<p>ベースライン</p>	<p>スピードアップ: 1.95x</p>

### 13.1.2 アセンブリーによるコーディング

組み込み関数の移植のガイドラインと同様に、アセンブリーで記述されたコードを移植するガイドラインを以下に示します。

- Y 静的および動的に割り当てられているバッファーを 64 バイトにアライメントします。
- Y 必要であれば補足のバッファーサイズを 2 倍にします。
- Y 命令ニーモニックの前に “v” プリフィクスを追加します。
- Y レジスター名を ymm から zmm へ変更します。
- Y ループの反復回数を半分にします (またはストライド長を倍にします)。

例 13-2 アセンブリーを使用したデカルト座標系の回転

インテル® AVX2 アセンブリー・コード	インテル® AVX-512 アセンブリー・コード
<pre>#include &lt;immintrin.h&gt; int main() { int len = 3200; // 32 バイト境界で動的にメモリーを割り当て float* pInVector = (float *) _mm_malloc(len*sizeof(float),32); float* pOutVector = (float *) _mm_malloc(len*sizeof(float),32);  // データの初期化 for (int i=0; i&lt;len; i++) pInVector[i] = 1;  float cos_teta = 0.8660254037; float sin_teta = 0.5;  // 8 つの float を 32 バイトのアライメントで静的に メモリー割り当て __declspec(aligned(32)) float cos_sin_teta_vec[8] = {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta};  __declspec(aligned(32)) float sin_cos_teta_vec[8] = {sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta};  __asm { mov rax,pInVector mov r8,pOutVector // ymm レジスターに 32 バイトをロード vmovups ymm3,ymmword ptr[cos_sin_teta_vec] vmovups ymm4, ymmword ptr[sin_cos_teta_vec]  mov edx, len shl edx, 2 xor ecx, ecx loop1: vmovsldup ymm0, [rax+rcx] vmovshdup ymm1, [rax+rcx] vmulps ymm1, ymm1, ymm4 vfmaddsub213ps ymm0, ymm3, ymm1</pre>	<pre>#include &lt;immintrin.h&gt; int main() { int len = 3200; // 64 バイト境界で動的にメモリーを割り当て float* pInVector = (float *) _mm_malloc(len*sizeof(float),64); float* pOutVector = (float *) _mm_malloc(len*sizeof(float),64);  // データの初期化 for (int i=0; i&lt;len; i++) pInVector[i] = 1;  float cos_teta = 0.8660254037; float sin_teta = 0.5;  // 16 個の float を 64 バイトのアライメントで静的に メモリー割り当て __declspec(aligned(64)) float cos_sin_teta_vec[16] = {cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta};  __declspec(aligned(64)) float sin_cos_teta_vec[16] = {sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta, sin_teta, cos_teta};  __asm { mov rax,pInVector mov r8,pOutVector // zmm レジスターに 64 バイトをロード vmovups zmm3, zmmword ptr[cos_sin_teta_vec] vmovups zmm4, zmmword ptr[sin_cos_teta_vec]  mov edx, len shl edx, 2 xor ecx, ecx loop1: vmovsldup zmm0, [rax+rcx] vmovshdup zmm1, [rax+rcx] vmulps zmm1, zmm1, zmm4 vfmaddsub213ps zmm0, zmm3, zmm1</pre>

<pre>// ymm レジスターから 32 バイトのストア vmovaps [r8+rcx], ymm0 vmovsldup ymm0, [rax+rcx+32] vmovshdup ymm1, [rax+rcx+32] vmulps ymm1, ymm1, ymm4 vfmaddsub213ps ymm0, ymm3, ymm1 // 前のストアからの 32 バイトのオフセット vmovaps [r8+rcx+32], ymm0  // このループで処理された 64 バイト // (コードは 2 回アンロール) add ecx, 64 cmp ecx, edx j1 loop1 }  _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>	<pre>// zmm レジスターから 64 バイトのストア vmovaps [r8+rcx], zmm0 vmovsldup zmm0, [rax+rcx+64] vmovshdup zmm1, [rax+rcx+64] vmulps zmm1, zmm1, zmm4 vfmaddsub213ps zmm0, zmm3, zmm1 // 前のストアからの 64 バイトのオフセット vmovaps [r8+rcx+64], zmm0  // このループで処理された 128 バイト // (コードは 2 回アンロール) add ecx, 128 cmp ecx, edx j1 loop1 }  _mm_free(pInVector); _mm_free(pOutVector); return 0; }</pre>
<p>ベースライン</p>	<p>スピードアップ: 1.95x</p>

## 13.2 マスク処理

拡張 VEX (EVEX) コード構成を使用するインテル® AVX-512 命令は、要素ごと計算操作を条件付きで制御し、結果をデスティネーション・オペランドに更新するためプレディケート・オペランドをエンコードします。プレディケート・オペランドは `opmask` (オペレーションマスク) レジスターとも呼ばれます。`opmask` はそれぞれ 64 ビット長の 8 つのアーキテクチャー・レジスターです。この 8 つのアーキテクチャー・レジスターは、プレディケート・オペランドとして `k1` から `k7` を介してアクセスできます。`k0` は通常のソースまたはデスティネーションとして使用できますが、プレディケート・オペランドとしてはエンコードできません。

プレディケート・オペランドは、メモリー・ソース・オペランドを持つ命令でメモリーフォルトを抑制するために使用できます。

プレディケート・オペランドとして `opmask` レジスターは、ベクトルレジスターの各要素を操作または更新するのを管理する 1 ビットを含んでいます。Skylake<sup>+</sup> マイクロアーキテクチャーのマスク命令は、すべてのデータサイズをサポートします: バイト (`int8`)、ワード (`int16`)、単精度浮動小数点 (`float32`)、整数ダブルワード (`int32`)、倍精度浮動小数点 (`float64`)、整数クアッドワード (`int64`)。ベクトルレジスターはデータサイズに応じて 8、16、32 または 64 個の要素を格納しています。そのため、ベクトル・マスク・レジスターは 64 ビット長となっています。

Skylake<sup>+</sup> マイクロアーキテクチャーでは、128 ビット、256 ビット、および 512 ビット長ベクトルのマスクが可能で、データ型とベクトル長に応じて、各命令は最下位マスクビットを要素の数だけアクセスします。例えば、インテル® AVX-512 命令が 512 ビット・ベクトルの 64 ビット・データ要素を操作する場合、`opmask` レジスターの下位 8 ビット (512/64) のみを使用します。`opmask` レジスターは、要素単位でインテル® AVX-512 命令に影響します。そのため、各データ要素のすべての数値または非数値操作、およびデスティネーション・オペランドへの要素ごとの中間結果の更新は、`opmask` レジスターの対応するビットで決定されます。

インテル® AVX-512 のプレディケート・オペランドとして機能する `opmask` は次のような特性を持っています。

- Y 命令操作は、対応する `opmask` ビットが設定されている要素に対してのみ行われます。これは、例外や違反もマスクがセットされていない要素の操作では引き起こされないことを意味します。その結果、マスクが設定されていない操作の結果として MXCSR 例外フラグが更新されることもありません。

- Y 対応する書込みマスクビットが設定されていない場合、デスティネーション要素には操作の結果は反映されません。その場合、デスティネーション要素の値は、保護されるか (マージマスク)、ゼロに設定されます (ゼロ化マスク)。
- Y メモリー操作を伴う命令のいくつかでは、マスクビット 0 の要素に対するメモリーフォルトが抑制されます。

マスクは、インテル® AVX-512 ベクトルレジスターのデスティネーションに対するマージ動作を提供するため、制御フローのプレディケーションを実装する強力な構造をもたらします。代替手段として、マージの代わりにゼロ化マスクを利用できます。これは、以前の値を保持する代わりにゼロで要素を更新します。古い値を必要としない場合、ゼロ化により暗黙的に依存関係が排除されます。

マスクを使用するほとんどの命令では、両方のマスク形式を利用できます。0 ではない (gather と scatter) EVEX.aaa ビットを持つ命令と、メモリーへの書き込みを行う命令は、マージマスクのみを受け入れます。

また、デスティネーション・オペランドがメモリー・ロケーションである場合、要素ごとにデスティネーション更新規則が適用されます。ベクトルは、プレディケート・オペランドとして使用される opmask レジスターに基づいて要素単位で書き込まれます。

opmask レジスターの値は次のように設定されます。

- Y ベクトル命令 (CMP, FPCLASS など) の結果として生成される。
- Y メモリーからロードされる。
- Y GPR レジスターからロードされる。
- Y マスクからマスクの操作で変更される。

### 13.2.1 マスクの例

マスク付き命令は、各データ要素に関連付けられたマスクビットに応じて、条件付きパックドデータ要素の操作を行います。各データ要素のマスクビットは、マスクレジスター内の対応するビットです。

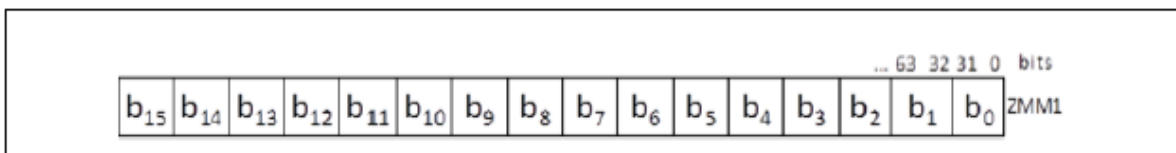
マスク付き命令を実行する場合、マスク値 0 に対応する要素には 0 が返されます。デスティネーション・レジスターの対応する値は、ゼロ化フラグに依存します。

- Y フラグが設定されていれば、メモリー・ロケーションにはゼロが書き込まれます。
- Y 設定されていない場合、メモリー・ロケーションの値は保護されます。

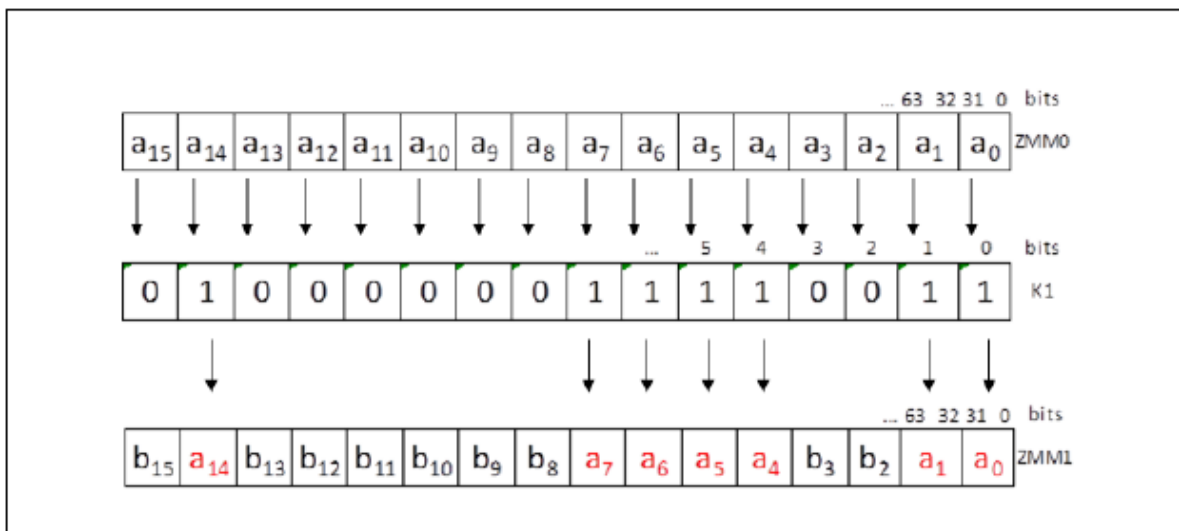
次の図では、マージマスクを使用したレジスターから別のレジスターへのマスク移動の例を示しています。

```
vmovaps zmm1 {k1}, zmm0
```

命令が実行される前のデスティネーション・レジスターの内容は、次のようになっています。

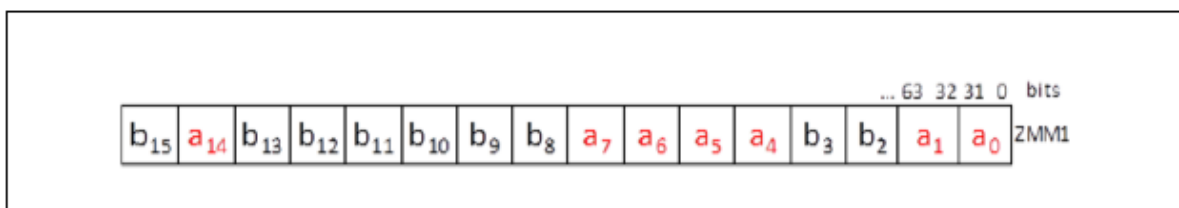


次のように操作されます。



ゼロ化マスクの実行結果には、次の命令を使用します (命令中の {z} に注目)。

```
vmovaps zmm1 {k1}{z}, zmm0
```



マージマスク操作はデスティネーションとの依存関係がありますが、ゼロ化マスクにはそのような依存関係がないことに注意してください。

次の例では、インテル® AVX2 に対してインテル® AVX-512 でどのようにマスク操作が行われるかを示しています。

C のコードに示します。

```
const int N = miBufferWidth;
const double* restrict a = A;
const double* restrict b = B;
double* restrict c = Cref;

for (int i = 0; i < N; i++){
    double res = b[i];
    if(a[i] > 1.0){
        res = res * a[i];
    }
    c[i] = res;
}
```

例 13-3 組込み関数とマスク

インテル® AVX2 組込み関数のコード	インテル® AVX-512 組込み関数のコード
<pre> for (int i = 0; i &lt; N; i+=32){     __m256d aa, bb, mask;     #pragma unroll(8)     for (int j = 0; j &lt; 8; j++){         aa = _mm256_loadu_pd(a+i+j*4);         bb = _mm256_loadu_pd(b+i+j*4);         mask =         _mm256_cmp_pd(_mm256_set1_pd(1.0), aa,         1);         aa = _mm256_and_pd(aa, mask); // 偽の値を         ゼロにする         false values         aa = _mm256_mul_pd(aa, bb);         bb = _mm256_blendv_pd(bb, aa, mask);         _mm256_storeu_pd(c+4*j, bb);     }     c += 32; } </pre>	<pre> for (int i = 0; i &lt; N; i+=32){     __m512d aa, bb;     __mmask8 mask;     #pragma unroll(4)     for (int j = 0; j &lt; 4; j++){         aa = _mm512_loadu_pd(a+i+j*8);         bb = _mm512_loadu_pd(b+i+j*8);         mask =         _mm512_cmp_pd_mask(_mm512_set1_pd(1.0),         aa, 1);         bb = _mm512_mask_mul_pd(bb, mask, aa,         bb);         _mm512_storeu_pd(c+8*j, bb);     }     c += 32; } </pre>
ベースライン	スピードアップ: 2.9x

例 13-4 アセンブリーとマスク

インテル® AVX2 アセンブリー・コード	インテル® AVX-512 アセンブリー・コード
<pre> loop: vmovupd ymm1, ymmword ptr [rax+rcx*8] inc r9d vmovupd ymm6, ymmword ptr [rax+rcx*8+0x20] vmovupd ymm2, ymmword ptr [r11+rcx*8] vmovupd ymm7, ymmword ptr [r11+rcx*8+0x20] vmovupd ymm11, ymmword ptr [rax+rcx*8+0x40] vmovupd ymm12, ymmword ptr [r11+rcx*8+0x40] vcmpdpd ymm4, ymm0, ymm1, 0x1 vcmpdpd ymm9, ymm0, ymm6, 0x1 vcmpdpd ymm14, ymm0, ymm11, 0x1 vandpd ymm16, ymm1, ymm4 vandpd ymm17, ymm6, ymm9 vmulpd ymm3, ymm16, ymm2 vmulpd ymm8, ymm17, ymm7 vmovupd ymm1, ymmword ptr [rax+rcx*8+0x60] vmovupd ymm6, ymmword ptr [rax+rcx*8+0x80] vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymm2, ymmword ptr [r11+rcx*8+0x60] vmovupd ymm7, ymmword ptr [r11+rcx*8+0x80] vmovupd ymmword ptr [rsi+r10*1], ymm5 vmovupd ymmword ptr [rsi+r10*1+0x20], ymm10 vcmpdpd ymm4, ymm0, ymm1, 0x1 vcmpdpd ymm9, ymm0, ymm6, 0x1 vandpd ymm18, ymm11, ymm14 vandpd ymm19, ymm1, ymm4 vandpd ymm20, ymm6, ymm9 </pre>	<pre> loop: vmovups zmm0, zmmword ptr [rax+rcx*8] inc r9d vmovups zmm2, zmmword ptr [rax+rcx*8+0x40] vmovups zmm4, zmmword ptr [rax+rcx*8+0x80] vmovups zmm6, zmmword ptr [rax+rcx*8+0xc0] vmovups zmm1, zmmword ptr [r11+rcx*8] vmovups zmm3, zmmword ptr [r11+rcx*8+0x40] vmovups zmm5, zmmword ptr [r11+rcx*8+0x80] vmovups zmm7, zmmword ptr [r11+rcx*8+0xc0] vcmpdpd k1, zmm8, zmm0, 0x1 vcmpdpd k2, zmm8, zmm2, 0x1 vcmpdpd k3, zmm8, zmm4, 0x1 vcmpdpd k4, zmm8, zmm6, 0x1 vmulpd zmm1{k1}, zmm0, zmm1 vmulpd zmm3{k2}, zmm2, zmm3 vmulpd zmm5{k3}, zmm4, zmm5 vmulpd zmm7{k4}, zmm6, zmm7 vmovups zmmword ptr [rsi+r10*1], zmm1 vmovups zmmword ptr [rsi+r10*1+0x40], zmm3 vmovups zmmword ptr [rsi+r10*1+0x80], zmm5 vmovups zmmword ptr [rsi+r10*1+0xc0], zmm7 add rcx, 0x20 add rsi, 0x100 cmp r9d, r8d jb loop </pre>

<pre> vmulpd ymm13, ymm18, ymm12 vmulpd ymm3, ymm19, ymm2 vmulpd ymm8, ymm20, ymm7 vmovupd ymm11, ymmword ptr [rax+rcx*8+0xa0] vmovupd ymm1, ymmword ptr [rax+rcx*8+0xc0] vmovupd ymm6, ymmword ptr [rax+rcx*8+0xe0] vblendvpd ymm15, ymm12, ymm13, ymm14 vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymm12, ymmword ptr [r11+rcx*8+0xa0] vmovupd ymm2, ymmword ptr [r11+rcx*8+0xc0] vmovupd ymm7, ymmword ptr [r11+rcx*8+0xe0] vmovupd ymmword ptr [rsi+r10*1+0x40], ymm15 vmovupd ymmword ptr [rsi+r10*1+0x60], ymm5 vmovupd ymmword ptr [rsi+r10*1+0x80], ymm10 vcmpdpd ymm14, ymm0, ymm11, 0x1 vcmpdpd ymm4, ymm0, ymm1, 0x1 vcmpdpd ymm9, ymm0, ymm6, 0x1 vandpd ymm21, ymm11, ymm14 add rcx, 0x20 vandpd ymm22, ymm1, ymm4 vandpd ymm23, ymm6, ymm9 vmulpd ymm13, ymm21, ymm12 vmulpd ymm3, ymm22, ymm2 vmulpd ymm8, ymm23, ymm7 vblendvpd ymm15, ymm12, ymm13, ymm14 vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymmword ptr [rsi+r10*1+0xa0], ymm15 vmovupd ymmword ptr [rsi+r10*1+0xc0], ymm5 vmovupd ymmword ptr [rsi+r10*1+0xe0], ymm10 add rsi, 0x100 cmp r9d, r8d jb loop </pre>	
<p>ベースライン</p>	<p>スピードアップ: 2.9x</p>

### 13.2.2 マスクのコスト

マスクを使用すると、マスクなしのコードに比べパフォーマンスが低下します。これは、次のいずれかが原因で発生します。

- Y それぞれのロードで追加のブレンド操作が行われるため。
- Y マージマスクによりデスティネーションで依存関係が存在するため、この依存関係はゼロ化マスクでは発生しません。
- Y より制限されたマスクのフォワード規則 (詳細については、フォワードとメモリーマスクを参照してください)。

次の例では、マージマスクを使用する際にどのようにデスティネーション・レジスターと依存関係が生じるか示しています。

表 13-1 マスク処理の例

マスクなし	マージマスク	ゼロマスク
<pre> mov rbx, iter loop:   vmulps zmm0, zmm9, zmm8   vmulps zmm1, zmm9, zmm8   dec rbx   jnle loop                     </pre>	<pre> mov rbx, iter loop:   vmulps zmm0{k1}, zmm9, zmm8   vmulps zmm1{k1}, zmm9, zmm8   dec rbx   jnle loop                     </pre>	<pre> mov rbx, iter loop:   vmulps zmm0{k1}{z}, zmm9, zmm8   vmulps zmm1{k1}{z}, zmm9, zmm8   dec rbx   jnle loop                     </pre>
ベースライン	スローダウン: 4x	スローダウン: ベースラインと同じ

マスクなしでは、プロセッサは 2 つの FMA ポートでサイクルごとに 2 つの乗算を実行できます。

マスクを伴うと、反復 N の乗算が反復 N - 1 の乗算の出力と依存性があるため、プロセッサは 4 サイクルごとに 2 つの除算を実行します。

ゼロ化マスクではデスティネーション・レジスターと依存関係を持たないため、2 つの FMA ポートでサイクルごとに 2 つの乗算を実行できます。

**推奨事項:** マスクにはコストが伴うため必要な場合にのみ使用します。可能であれば、マージマスクではなくゼロ化マスクを使用します。

### 13.2.3 マスクとブレンド

この節では、条件付きコードにおいてブレンドとマスクを使用する利点と欠点について説明します。

次のコードについて考えてみます。

```

for ( i=0; i<SIZE; i++ )
{
  if ( a[i] > 0 )
  {
    b[i] *= 2;
  }
  else
  {
    b[i] /= 2;
  }
}
                    
```



次の例では、2 つのコンパイル可能なコードを示しています。

- Y 代替 1 は、マスクされたコードと簡単なデータの数値処理を使用します。
- Y 代替 2 では、コードを連続して処理される 2 つの独立したマスクなしのフローに分割し、次にメモリーヘストアする前にマスクされた移動 (ブレンド) を行います。

例 13-5 マスクとブレンドの例 1

代替 1	代替 2
<pre> mov rax, pImage mov rbx, pImage1 mov rcx, pOutImage mov rdx, len vpxord zmm0, zmm0, zmm0 mainloop: vmovdqa32 zmm2, [rax+rdx*4-0x40] vmovdqa32 zmm1, [rbx+rdx*4-0x40] vpcmpgtd k1, zmm0, zmm1 knotw k2, k1 (1) vpslld zmm2 {k1}, zmm2, 1 (2) vpsrld zmm2 {k2}, zmm2, 1 (3) vmovdqa32 [rcx+rdx*4-0x40], zmm2 sub rdx, 16 jne mainloop                     </pre>	<pre> mov rax, pImage mov rbx, pImage1 mov rcx, pOutImage mov rdx, len vpxord zmm0, zmm0, zmm0 mainloop: vmovdqa32 zmm2, [rax+rdx*4-0x40] vmovdqa32 zmm1, [rbx+rdx*4-0x40] vpcmpgtd k1, zmm0, zmm1 vmovdqa32 zmm3, zmm2 vpslld zmm2, zmm2, 1 vpsrld zmm3, zmm3, 1 (1) vmovdqa32 zmm3 {k1}, zmm2 (2) vmovdqa32 [rcx+rdx*4-0x40], zmm3 sub rdx, 16 jne mainloop                     </pre>
<p>ベースライン・サイクル 1x                      ベースライン命令 1x</p>	<p>スピードアップ: 1.23x                      命令: 1.11x</p>

代替 1 には、命令 (1) と (2)、および (2) と (3) の間に依存関係があります。これは、命令 (2) は実行を開始する前に命令 (1) のブレンドの結果を待機する必要があり、また命令 (3) は命令 (2) を待機することを意味します。

代替 2 では、それぞれの条件コードの分岐はすべてのデータに対し並列に実行され、マスクはメモリーにデータを書き戻す前に 1 つのレジスターへのブレンドに使用されるため、依存関係は 1 つしかありません。

ブレンドは高速ですが、マスクなしデータに対して発生する可能性がある例外をマスクしません。

代替 2 は 11% 多くの命令を実行しますが、全体の実行で 23% のスピードアップをもたらします。代替 2 ではレジスターを 1 つ多く使用します (zmm3)。この追加レジスターの使用は、レジスター・プレッシャー (レジスターをメモリーに退避し、その後ロードする) が高まると追加のレイテンシーを生じさせます。

次のコードはマスクとブレンドのもう一つの例です。

```

for (int =0;i<len;i++){
    if (a[i] > b[i]){
        a[i] += b[i]
    }
}
                    
```

例 13-6 マスクとブレンドの例 2

代替 1	代替 2
<pre> mov rax,a mov rbx,b mov rdx,size2 vpxord zmm0,zmm0,zmm0 loop1: vmovdqa32 zmm1,[rax +rdx*4 -0x40] vmovdqa32 zmm2,[rbx +rdx*4 -0x40] (1) vpcmpgtd k1,zmm1,zmm2 (2) vmovdqa32 zmm3{k1}{z},zmm2 (3) vpadd zmm1,zmm1,zmm3 vmovdqa32 [rax +rdx*4 -0x40],zmm1 sub rdx,16 jne loop1                     </pre>	<pre> mov rax,a mov rbx,b mov rdx,size2 vpxord zmm0,zmm0,zmm0 loop1: vmovdqa32 zmm1,[rax +rdx*4 -0x40] vmovdqa32 zmm2,[rbx +rdx*4 -0x40] (1)vpcmpgtd k1,zmm1,zmm2 (2)vpadd zmm1{k1},zmm1,zmm2 vmovdqa32 [rax +rdx*4 -0x40],zmm1 sub rdx,16 jne loop1                     </pre>
<p>ベースライン・サイクル 1x ベースライン命令 1x</p>	<p>スピードアップ: 1.05x 命令: 0.87x</p>

代替 1 には、命令 (1) と (2)、および (2) と (3) の間に依存関係があります。

代替 2 では、2 つの命令 (1) と (2) にのみ依存関係のチェーンがあります。

### 13.2.4 入れ子になった条件 / マスク集合

インテル® AVX-512 には、入れ子と (または) 複数の条件操作の実装を容易にするため、マスクレジスターですべてのビット単位の論理演算子の実行を可能にする、一連のマスク操作命令が含まれます。次の例では、論理和 (&&) は *kandw* 命令を使用して実行されます。

```

for(int iX = 0; iX < iBufferWidth; iX++)
{
    if ((*pInImage)>0 && ((*pInImage)&3)==3)
    {
        *pRefImage = (*pInImage)+5;
    }
    else
    {
        *pRefImage = (*pInImage);
    }
    pRefImage++;
    pInImage++;
}
                    
```

例 13-7 複数条件式の実行

スカラー	インテル® AVX2	インテル® AVX-512
<pre>mainloop: mov r8d, dword ptr [rsi+rax*4] mov r9d, r8d cmp r8d, 0 jle label1 and r9d, 0x3 cmp r9d, 3 jne label1 add r8d, 5 label1: mov dword ptr [rdi+rax*4], r8d add rax, 1 cmp rax, rbx jne mainloop</pre>	<pre>mov rsi, pImage mov rdi, pOutImage mov rbx, [len] xor rax, rax vpbroadcastd ymm1, [five] vpbroadcastd ymm7, [three] vpxord zmm3, zmm3, zmm3 mainloop: vmovdqa ymm0, [rsi+rax*4] vmovaps ymm6, ymm0 vpcmpgtd ymm5, ymm0, ymm3 vpandd ymm6, ymm6, ymm7 vpcmpeqd ymm6, ymm6, ymm7 vpandd ymm5, ymm5, ymm6 vpaddd ymm4, ymm0, ymm1 vblendvps ymm4, ymm0, ymm4, ymm5 vmovdqa32 [rdi+rax*4], ymm4 add rax, 8 cmp rax, rbx jne mainloop</pre>	<pre>mov rsi, pImage mov rdi, pOutImage mov rbx, [len] xor rax, rax vpbroadcastd zmm1, [five] vpbroadcastd zmm5, [three] vpxord zmm3, zmm3, zmm3 mainloop: vmovdqa32 zmm0, [rsi+rax*4] vpcmpgtd k1, zmm0, zmm3 vpandd zmm6, zmm5, zmm0 vpcmpeqd k2, zmm6, zmm5 kandw k1, k2, k1 vpaddd zmm0 {k1}, zmm0, zmm1 vmovdqa32 [rdi+rax*4], zmm0 add rax, 16 cmp rax, rbx jne mainloop</pre>
ベースライン 1x	スピードアップ: 5x	スピードアップ: 11x

### 13.2.5 メモリーマスクのマイクロアーキテクチャーを改善

表 13-2 に、Broadwell<sup>†</sup> マイクロアーキテクチャー以降で改善されたマスク操作の一覧を示します。

表 13-2 Skylake Server<sup>†</sup> と Broadwell<sup>†</sup> マイクロアーキテクチャーのキャッシュの比較

項目	Broadwell <sup>†</sup> マイクロアーキテクチャー	Skylake Server <sup>†</sup> マイクロアーキテクチャー
1	VMASKMOV ストアのアドレスは、マスクが判明した後のみ解決されると考えられるべきです。マスク付きストアに続くロードは、マスク値が判明するまでブロックされます (メモリー・ディスアンピゲーションによって解決されない限り)。	この問題は解決されています。
2	マスクがすべて 1 または 0 でないロードはマスク付きストアに依存し、ストアデータがキャッシュに書き込まれるまで待機します。マスクがすべて 1 のデータは、マスク付きストアから依存関係のあるロードへフォワードされます。マスクがすべて 0 のロードはマスク付きストアと依存関係はありません。	マスクがすべて 1 または 0 でないロードはマスク付きストアに依存し、ストアデータがキャッシュに書き込まれるまで待機します。マスクがすべて 1 のデータは、マスク付きストアから依存関係のあるロードへフォワードされます。マスクがすべて 0 のロードはマスク付きストアと依存関係はありません。
3	マスク付きロード (vmaskmov 命令を使用した) が不正なメモリーアドレス範囲を含んでいる場合、プロセッサはどの不正な範囲にマスク値が「1」に設定されているかどうかを確認するため、複数サイクルのアシストコードを実行する可能性があります。このアシストは、マスクが「ゼロ」であってもプログラマーにとってロードを実行すべきでないことが明白な場合に行われる可能性があります。	インテル® AVX-512 のマスク処理では、マスク値がすべてゼロであればメモリーフォルトは無視されアシストは発行されません。

### 13.2.6 ピーリングとリマインダーのマスク

アライメントされたデータのキャッシュラインへのアクセスは、アライメントされていないデータへのアクセスより高いパフォーマンスをもたらします。多くの場合、アドレスはコンパイル時には不明であるが、判明していてもアライメントされていません。このような場合、最初の要素からアライメントされているアドレスまでをマスク付きモードで処理し、その後ループボディーをマスクなしで処理するピーリング・アルゴリズムを導入することを推奨します。必要に応じてループの終端をリマインダー・ループでマスク付きで処理します。この方式はコードサイズを増加させますが、データ処理全体のパフォーマンスを改善します。

例 13-8 は、2 つのコードバージョン（両者ともアライメントされていない出力データ配列を使用）の実装と実行スピードの違いを示しています。

例 13-8 ピーリングとリマインダーのマスク

ピーリングなし、マスクなしのボディー、マスクされたリマインダー	ピーリングあり、マスクなしのボディー、マスクされたリマインダー
<pre> mov rbx, pOutImage // 出力 mov rax, pImage // 入力 mov rcx, len mov edx, addValue vpbroadcastd zmm0, edx mov edx, alfa vpbroadcastd zmm3, edx mov rdx, rcx sar rdx, 4 // 反復あたり 16 要素、RDX - ループ総数  jz remainder // すべてを反復しない xor r8, r8 vmovups zmm10, [indices] mainloop: vmovups zmm1, [rax + r8] vfmadd213ps zmm1, zmm3, zmm0 vmovups [rbx + r8], zmm1 add r8, 0x40 sub rdx, 1 jne mainloop  remainder: // リマインダーのマスクを作成 and rcx, 0xF // リマインダーでの要素数 jz end // リマインダーの要素なし vpbroadcastd zmm2, ecx vpcmpd k2, zmm10, zmm2, 1 // 低位を比較  vmovups zmm1 {k2}{z}, [rax + r8] vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0 vmovups [rbx + r8] {k2}, zmm1 end: </pre>	<pre> mov rax, pImage // 入力 mov rbx, pOutImage // 出力 mov rcx, len mov edx, addValue vpbroadcastd zmm0, edx mov edx, alfa vpbroadcastd zmm3, edx xor r8, r8 xor r9, r9 vmovups zmm10, [indices] vmovups zmm11, [itersize] vpbroadcastd zmm12, ecx  peeling: mov rdx, rbx and rdx, 0x3F jz endofpeeling // ピールの必要なし neg rdx add rdx, 64 // 64 - X // rdx にアライメントされた位置までのバイト数が格納されている mov r9, rdx sar r9, 2 // r9 にはピーリングされた要素数が含まれる vpbroadcastd zmm12, r9d vpcmpd k2, zmm10, zmm12, 1 // ピーリングのマスクを生成するため低位を比較  vmovups zmm1 {k2}{z}, [rax] vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0 vmovups [rbx] {k2}, zmm1 // アライメントされていないストア  endofpeeling: sub rcx, r9 mov r8, rcx sar r8, 4 // ループの総数 jz remainder // すべてを反復しない </pre>

	<pre> mainloop:     vmovups zmm1, [rax + rdx]     vfmadd213ps zmm1, zmm3, zmm0     vmovaps [rbx + rdx], zmm1 // アライメントされた     ストア     add rdx, 0x40     sub r8, 1     jne mainloop  remainder:     // リマインダーのマスクを作成     and rcx, 0xF // リマインダーでの要素数     jz end // リマインダーの要素なし     vpbroadcastd zmm2, ecx     vpcmpd k2, zmm10, zmm2, 1 // 低位を比較     vmovups zmm1 {k2}{z}, [rax + rdx]     vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0     vmovaps [rbx + rdx] {k2}, zmm1 //アライメントさ     れている end:                 </pre>
ベースライン 1x	スピードアップ: 1.04x

### 13.3 フォワーディングとマスク付き操作

マスクなしのストア命令を使用しその後にロード命令が続く場合、データのフォワーディングはロードのタイプ、サイズそしてストアアドレスからのアドレス・オフセットに依存しますが、ストアアドレスそのものには関係しません (ストアアドレスはキャッシュラインにアライメントおよびフィットする必要がなく、フォワーディングはアライメントされずライン分割されたストアで発生します)。

次の図は、データのフォワーディングが起こる可能性があるすべてのケースを示しています。

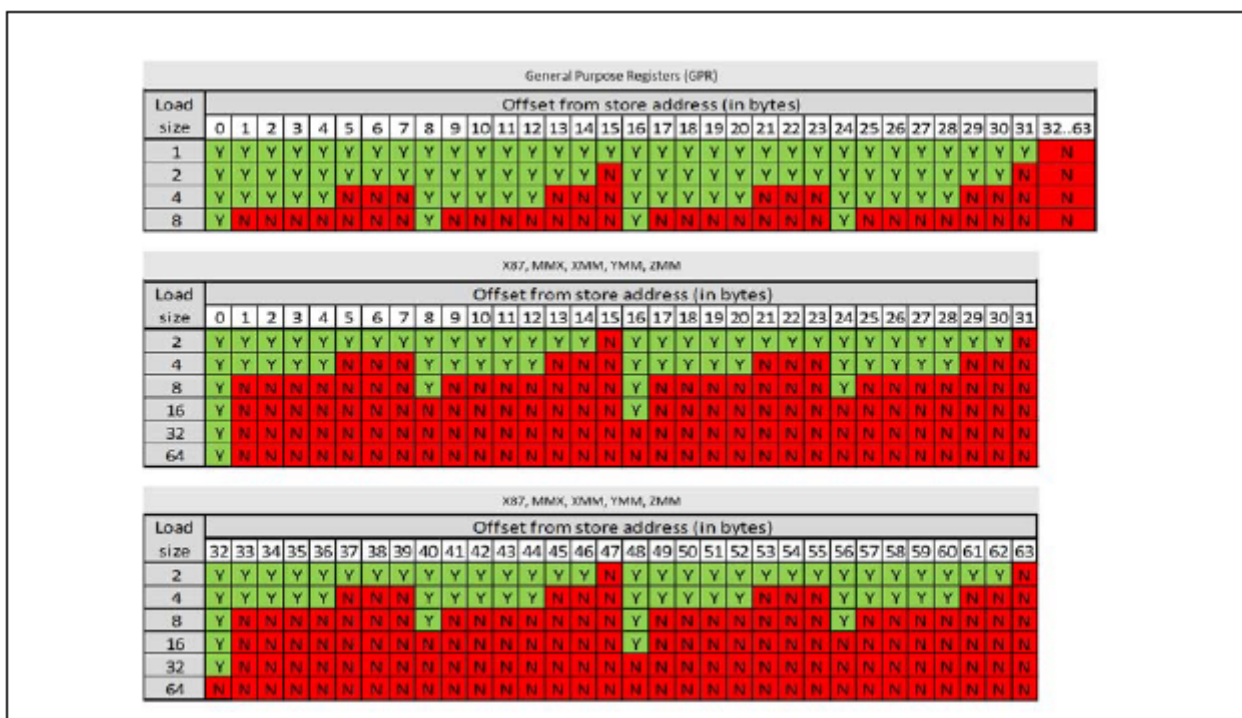


図 13-3 データがフォワードされる場合

データ・フォワーディングを使用する際に考慮すべき 2 つの重要なことがあります。

1. GPR へのデータフォワードは、ストア命令の下位 256 ビットのみが可能です。直前に書き込まれたデータを GPR にロードする際は注意が必要です。
2. フォワーディングは特定のマスクでのみ行われるため、マスクを使用してはいけません。

## 13.4 フォワーディングとメモリーマスク

マスク付きロードやストアを使用する場合、次のことを考慮してください。

- Y マスク値がすべてゼロまたはすべて非ゼロではない場合、マスク付きストアに続くロード操作が同じアドレスを持っていると、データがキャッシュに書き込まれるまでブロックされます。
- Y GPR のフォワーディングとは異なり、ロードとストアアドレスが完全に同じではない場合、ベクトルロードのマスクありなしにかかわらずフォワードは行われません。
  - st\_mask = 10101010、ld\_mask = 01010101、フォワードなし、ブロックされる。
  - st\_mask = 00001111、ld\_mask = 00000011、フォワードなし、ブロックされる。
- Y マスク値がすべて 1 である場合、データはロード操作にフォワードされるためブロックされません。
  - st\_mask = 11111111、ld\_mask = 影響なし、フォワードあり、ブロックされない。
- Y また、マスク値がすべてゼロである場合、フォワードされませんがブロックされることもありません。
  - st\_mask = 00000000、ld\_mask = 影響なし、フォワードなし、ブロックされない。

マスク付きのストアは慎重に利用されるべきです。例えば、リマインダーのサイズがコンパイル時に 1 であることが判明していれば、その後に同じキャッシュラインからロードする操作があれば (または、アドレス + ベクトル長でオーバーラップする)、マスク付きのリマインダー・ブロックよりもスカラーのリマインダーを使用した方が良いでしょう。

## 13.5 データコンプレス

データコンプレス操作は、入力バッファーからマスクレジスター 1 ビットで指定されたインデックスの要素を読み込みます。読み込まれた要素はデスティネーション・バッファーへ書き込まれます。要素数がデスティネーション・レジスターのサイズ未満であれば、余りの空間はゼロで埋められます。

次の図はデータコンプレス操作を示します。

```
if (k[i] == 1)
{
    dest[a] = src[i];
    a++;
}
```

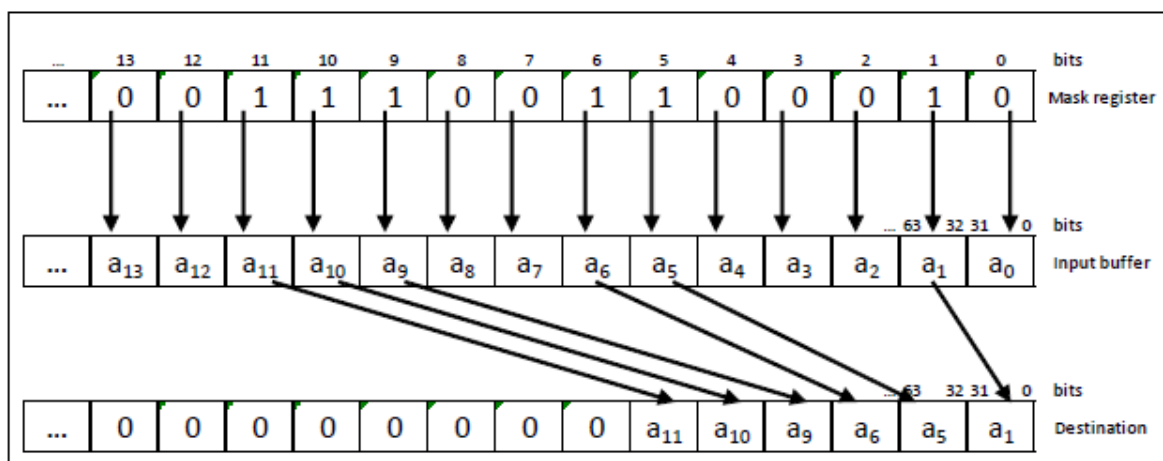


図 13-4 データコンプレス操作

### 13.5.1 データコンプレスの例

次のコードは、配列 a から正の値を持つデータを配列 b に移動 (収集) しています。

```
for (int i=0; i<SIZE; i++)
{
    if ( a[i] > 0 )
        b[j++] = a[i];
}
```

次の 4 つの実装は dword 要素の配列からのコンプレス操作を行っています。

- Y 代替 1 では、スカラーのデータアクセスを行い、各要素を個別にチェックしています。0 より大きな値はデスティネーション配列に書き込まれます。
- Y 代替 2 は、事前割り当てによりシャッフルキーで初期化されたテーブルと、インテル® AVX の shuffle 命令を使用しています。比較命令はシャッフルテーブルのエントリーポイント数を提供し、次にキーがロードされ、キーに従って元の配列がシャッフルされます。反復ごとに 4 つの要素が処理されます。
- Y 代替 3 は、代替 2 と同じアルゴリズムを使用しますが、インテル® AVX2 の 256 ビットレジスターと、バイトシャッフルに変わって dword 命令で並べ替えを使用します。反復ごとに 8 つの要素が処理されます。
- Y 代替 4 は、コンプレスキーとしてマスクレジスターと *vpcompress* 命令を使用するインテル® AVX-512 のアルゴリズムです。反復ごとに 16 の要素が処理されます。

例 13-9 インテル® AVX-512 のデータコンプレスと他の代替実装を比較

代替 1 スカラー	代替 2 インテル® AVX
<pre> mov rsi, source mov rdi, dest mainloop: mov r11d, dword ptr [rsi+r8*4] test r11d, r11d jle m1 mov dword ptr [rdi+r10*4], r11d inc r10 m1: inc r8 cmp r8, r9 jne mainloop                     </pre>	<pre> mov rsi, source mov rdi, dest mov r14, shuffle_LUT mov r15, write_mask mainloop: vmovdqa xmm1, [rsi+r8*4] vpcmpgtd xmm2, xmm1, xmm0 mov r10, 4 vmovmskps r13, xmm2 shl r13, 4 vmovdqu xmm3, [r14+r13] vpshufb xmm2, xmm1, xmm3 popcnt r13, r13 sub r10, r13 vmovdqu xmm3, [r15+r10*4] vmaskmovps [rdi+r11*4], xmm3, xmm2 add r11, r13 add r8, 4 cmp r8, r9 jne mainloop                     </pre>
<p>ベースライン 1x</p>	<p>スピードアップ: 2.87x</p>
代替 3 インテル® AVX	代替 4 インテル® AVX-512
<pre> mov rsi, source mov rdi, dest mov r14, shuffle_LUT mov r15, write_mask mainloop: vmovdqa ymm1, [rsi+r8*4] vpcmpgtd ymm2, ymm1, ymm0 mov r10, 8 vmovmskps r13, ymm2 shl r13, 5 vmovdqu ymm3, [r14+r13] vpermd ymm2, ymm3, ymm1 popcnt r13, r13 sub r10, r13 vmovdqu ymm3, [r15+r10*4] vmaskmovps [rdi+r11*4], ymm3, ymm2 add r11, r13 add r8, 8 cmp r8, r9 jne mainloop                     </pre>	<pre> mov rsi, source mov rdi, dest mainloop: vmovdqa32 zmm1, [rsi+r8*4] vpcmpgtd k1, zmm1, zmm0 vpcompressd zmm2 {k1}, zmm1 vmovdqu32 [rdi+r10*4], zmm2 kmovd r11d, k1 popcnt r12, r11 add r8, 16 add r10, r12 cmp r8, r9 jne mainloop                     </pre>
<p>スピードアップ: 5.27x</p>	<p>スピードアップ: 11.9x</p>



## 13.6 データ・エクスパンド

データ・エクスパンド操作は、ソース配列 (レジスター) から要素を読み込み、マスクレジスターのビットで有効化された位置の要素をデスティネーション・レジスターに書き込みます。有効化されたビット数がデスティネーション・レジスターのサイズ未満である場合、余分な値は無視されます。

```
if (k[i] == 1)
{
    dest[i] = src[a];
    a++;
}
```

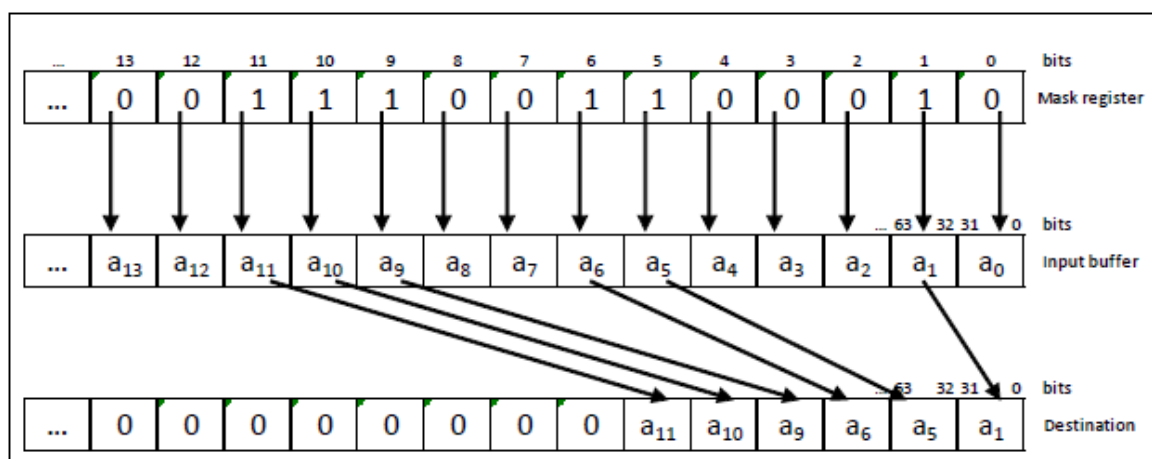


図 13-5 データ・エクスパンド操作

### 13.6.1 データ・エクスパンドの例

次のコードはエクスパンド操作を使用した例を示しています。配列 `a` のすべての正数に対し、コードは一連の番号を設定します。

```
for (int i=0; i<SIZE; i++){
    if (a[i] > 0)
        dest[i] = a[count++];
    else
        dest[i] = 0;
}
```

次の 3 つの実装は、16 個の `dword` 要素の配列からのエクスパンド操作を行っています。

- Y 代替 1 では、スカラーのデータアクセスを行い、各要素を個別にチェックしています。値が 0 より大きければデスティネーション配列の対応する要素は、インデックス・カウンターで示されるソースの値で上書きされ、カウンターがインクリメントされます。
- Y 代替 2 は、事前割り当てによりシャッフルキーで初期化されたテーブルと、インテル® AVX の `shuffle` 命令を使用しています。比較命令はシャッフルテーブルのエントリーポイント数を提供し、次にキーがロードされ、キーに従って元の配列がシャッフルされます。反復ごとに 4 つの要素が処理されます。
- Y 代替 3 は、エクスパンド・キーとしてマスクレジスターと `vpexpandd` 命令を使用するインテル® AVX-512 のアルゴリズムです。反復ごとに 16 の要素が処理されます。

例 13-10 インテル® AVX-512 のデータ・エクスパンドと他の代替実装を比較

代替 1: スカラー	代替 2: インテル® AVX2 コード	代替 3: インテル® AVX-512 コード
<pre>mainloop:   mov r11d, dword ptr     [rsi+r8*4]   test r11d, r11d   jle m1   mov r11d, dword ptr     [rsi+r10*4]   mov dword ptr [rdi+r8*4],     r11d   inc r10 m1:   inc r8   cmp r8, r9   jne mainloop</pre>	<pre>vpxord ymm0, ymm0, ymm0 mov r14, shuf2 xor r15,r15  mainloop: vmovdqa ymm1,[rsi+r8*4] vpbroadcastd ymm4,r15d vpcmpgtd ymm2,ymm1,ymm0 vmovdqu ymm1,[rsi+r10*4] vmovmskps r13,ymm2 shl r13, 5 vmovdqa ymm3,[r14+r13] vpermd ymm4,ymm3,ymm1 popcnt r13,r13 add r10, r13 vmaskmovps [rdi+r8*4],ymm2,   ymm4 add r8, 8 cmp r8, r9 jne mainloop</pre>	<pre>vpxord zmm0, zmm0, zmm0 mainloop: vmovdqa32 zmm1,[rsi+r8*4] vpcmpgtd k1, zmm1, zmm0 vmovdqu32 zmm1,[rsi+r10*4] vpexpandd zmm2 {k1}{z}, zmm1 vmovdqu32 [rdi+r8*4],zmm2 add r8, 16 kmovd r11d, k1 popcnt r12, r11 add r10, r12 cmp r8, r9 jne mainloop</pre>
ベースライン 1x	スピードアップ: 4.23x	スピードアップ: 8.58x

### 13.7 三値論理

三値論理 `vpternlog` 命令は、指定される 3 つのオペランド間のどのような論理関数も実行します。命令は 3 つのオペランドと即値 (この論理式の真理値表) を必要とします。最初のオペランドはデスティネーションでもあるため、実行後元の値は上書きされます。

#### 13.7.1 三値論理の例 1

次の例は 3 つの値のビット単位の論理関数を示しています。この例の関数は次の真理値表で定義されます。

X	1	1	1	1	0	0	0	0	Immediate value that is used in 0x92
Y	1	1	0	0	1	1	0	0	
Z	1	0	1	0	1	0	1	0	
f(X,Y,Z)	1	0	0	1	0	0	1	0	

図 13-6 三値論理の例 1 - 真理値表

この真理値表のカルノー図を使用して、次のように関数を定義できます。

$$f(X,Y,Z) = \bar{x}\bar{y}z \vee xyz \vee x\bar{y}\bar{z}$$

または、短い表記で、より少ないブール演算を使用します。

$$f(X,Y,Z) = \bar{y}(z \oplus x) \vee xyz$$

上記の関数の C コードは次のようになります。

```
for (int i=0; i<SIZE; i++)
{
  Dst[i] = (( Src2[i]) & (Src1[i] ^ Src3[i])) | (Src1[i] & Src2[i] & Src3[i]);
}
```

X、Y および Z のそれぞれの組み合わせによる関数の値は、命令で使用される即値を与えます。

以下に X、Y および Z のすべての値に、この論理関数を適用する 3 つの実装を示します。

- Y 代替 1 は、インテル® AVX2 で利用可能なビット単位の論理関数を使用した、256 ビット・ベクトル計算です。
- Y 代替 2 は、インテル® AVX-512 で利用可能なビット単位の論理関数を使用した 512 ビット・ベクトル計算です (*vpternlog* 命令なし)。
- Y 代替 2 は、*vpternlog* 命令を使用したインテル® AVX-512 による 512 ビット・ベクトル計算です。すべての例で、2 回のアンロールが行われています。

### 例 13-11 三値論理と代替手法の比較

<p><b>代替 1: インテル® AVX2</b></p> <pre> mov rax, src1 mov rbx, src2 mov rcx, src3 mov r11, dst mainloop:   vmovdqu ymm1, ymmword ptr [rax+r10*4]   vmovdqu ymm3, ymmword ptr [rdx+r10*4]   vmovdqu ymm2, ymmword ptr [rcx+r10*4]   vmovdqu ymm10, ymmword ptr [rcx+r10*4+0x20]   vpand ymm0, ymm1, ymm3   vpxor ymm4, ymm1, ymm2   vpand ymm5, ymm0, ymm2   vpandn ymm6, ymm3, ymm4   vpor ymm7, ymm5, ymm6   vmovdqu ymmword ptr [r11+r10*4], ymm7   vmovdqu ymm9, ymmword ptr [rax+r10*4+0x20]   vmovdqu ymm11, ymmword ptr [rdx+r10*4+0x20]   vpxor ymm12, ymm9, ymm10   vpand ymm8, ymm9, ymm11   vpandn ymm14, ymm11, ymm12   vpand ymm13, ymm8, ymm10   vpor ymm15, ymm13, ymm14   vmovdqu ymmword ptr [r11+r10*4+0x20], ymm15   add r10, 0x10   cmp r10, r8   jb mainloop </pre>
<p><b>ベースライン 1x</b></p>

例 13-11 三値論理と代替手法の比較 (続き)

代替 2: インテル® AVX-512 論理命令	代替 3: インテル® AVX-512 の vpternlog 命令を使用
<pre> mov rdi, src1 mov rsi, src2 mov rdx, src3 mov r10, dst mainloop: vmovups zmm2,k0,zmmword ptr [rdi+r10*4] vmovups zmm4,k0,zmmword ptr [rdi+r10*4+0x40] vmovups zmm6,k0,zmmword ptr [rsi+r10*4] vmovups zmm8,k0,zmmword ptr [rsi+r10*4+0x40] vmovups zmm3,k0,zmmword ptr [rdx+r10*4] vmovups zmm5,k0,zmmword ptr [rdx+r10*4+0x40] vpandd zmm0,k0, zmm2, zmm6 vpandd zmm1,k0, zmm4, zmm8 vpxord zmm7,k0, zmm2, zmm3 vpxord zmm9,k0, zmm4, zmm5 vpandd zmm10,k0, zmm0, zmm3 vpandd zmm12,k0, zmm1, zmm5 vpandnd zmm11,k0, zmm6, zmm7 vpandnd zmm13,k0, zmm8, zmm9 vpord zmm14,k0, zmm10, zmm11 vpord zmm15,k0, zmm12, zmm13 vmovups zmmword ptr [r10+r10*4],k0,zmm14 vmovups zmmword ptr [r10+r10*4+0x40],k0,zmm15 add r10, 0x20 cmp r10, r9 jb mainloop                     </pre>	<pre> mov rdi, src1 mov rsi, src2 mov rdx, src3 mov r10, dst mainloop: vmovaps zmm1,[r8+rax*4] vmovaps zmm0,[r9+rax*4] vpternlogd zmm0,zmm1,[r10], 0x92 vmovaps [r10],zmm0 vmovaps zmm1,[r8+rax*4+0x40] vmovaps zmm0,[r9+rax*4+0x40] vpternlogd zmm0,zmm1,[r10+0x40],0x92 vmovaps [r10+0x40],zmm0 add rax,32 add r10,0x80 cmp eax,esi jne mainloop                     </pre>
<p>スピードアップ: 1.94x</p>	<p>スピードアップ: 2.36x (インテル® AVX-512 論理命令に対し 1.22x)</p>

### 13.7.2 三値論理の例 2

次の例は、Fortran で良く使用される符号変換操作です。浮動小数点数を持つ 2 つの配列を使用するコードを考えてみましょう。

```

for (int i=0; i<SIZE; i++)
{
    b[i] = a[i] > 0 ? b[i] : -b[i];
}
                    
```

これは、次のコードと等価です。

```

for (int i=0; i<SIZE; i++)
{
    b[i] = ( a[i] & 0x80000000 ) ^ b[i];
}
                    
```

言い換えると、

$$x = (y \wedge z) \oplus x$$

は、次の真理値表を定義します。

X	1	1	1	1	0	0	0	0	<div style="border: 1px solid black; padding: 2px;">                     Immediate value that is used in vpternlog instruction.                 </div>
Y	1	1	0	0	1	1	0	0	
Z	1	0	1	0	1	0	1	0	
f(X,Y,Z)	0	1	1	1	1	0	0	0	

図 13-7 三値論理の例 2 - 真理値表

つまり 2 つの論理命令 (*vpand* と *vpxor*) は、*vpternlog* 命令に置き換えることができます。

```
vpternlog x,y,z,0x78
```

### 13.8 新しいシャッフル命令

インテル® AVX-512 では 3 つの新しい shuffle 操作が追加されています。

- Y *vpermw*: いずれの word も並べ替え可能な新しい単一ソース命令
- Y *permt2[w/d/q/ps/pd]*: 2 つのソースのいずれも並べ替え可能な新しい命令 (ソースレジスターを上書き)
- Y *permi2[w/d/q/ps/pd]*: 2 つのソースのいずれも並べ替え可能な新しい命令 (制御レジスターを上書き)

次の図に *vpermi2ps* 命令の仕組みを示します。この例では *zmm0* はシャッフルの制御に使用されますが、出力レジスター (制御レジスターは上書きされます) でもあることに注意してください。

```
vpermi2ps zmm0, zmm1, zmm2
```

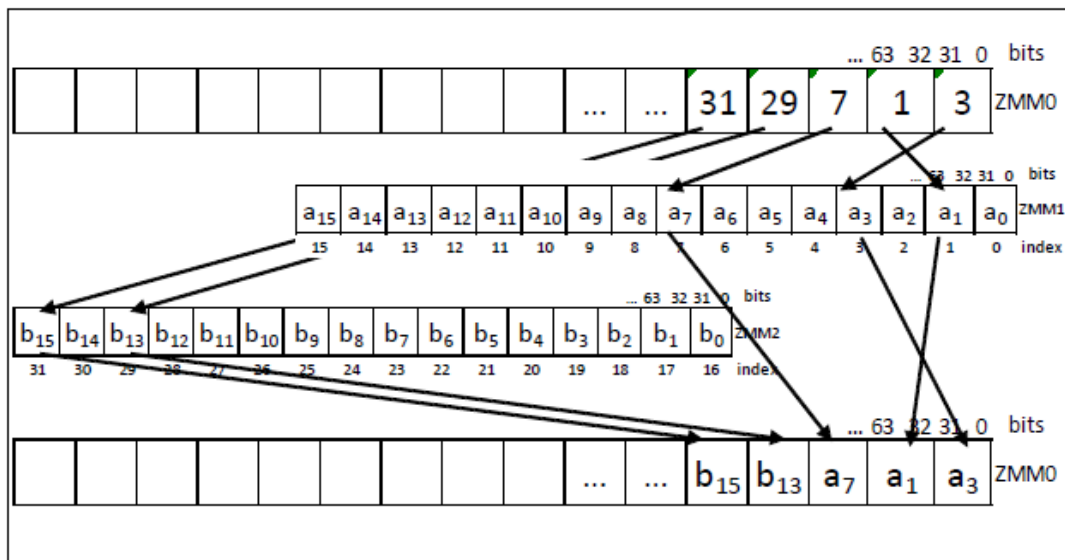


図 13-8 *vpermi2ps* 命令の操作

インデックス・レジスターの値は、命令とソースレジスターで同じ解像度でなければならないことに注意してください (word を扱うときは word、dword の時は dword など)。

### 13.8.1 2 つのソースのパーミュートの例

この例では、行列の転置操作における 2 つのソースを持つパーミュート命令の使い方を示します。転置する行列は word 要素の 8x8 正方形行列です。

$$\begin{bmatrix} a_{00} & \dots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \dots & a_{77} \end{bmatrix}^T \longrightarrow \begin{bmatrix} a_{00} & \dots & a_{71} \\ \vdots & \ddots & \vdots \\ a_{17} & \dots & a_{77} \end{bmatrix}$$

C コードで表すと次のようになります (各行列は  $8 \times 8 \times 2 = 128$  バイトの連続したブロックであると仮定しています)。

```
for(int iY = 0; iY < 8; iY++)
{
    for(int iX = 0; iX < 8; iX++)
    {
        trasposedMatrix[iY*8+iX] = originalMatrix[iX*8+iY];
    }
}
```

この行列転置の 3 つの実装を用意しました。

- Y 代替 1 は、スカラーコードで、ソース行列のそれぞれの要素をアクセスして、デスティネーション行列の対応する位置にそれを配置します。このコードは、1 つの行列ごとに 64 (8x8) 回反復します。
- Y 代替 2 は、インテル® AVX2 コードで、インテル® AVX2 のパーミュートとシャッフル (アンパック) 命令を使用します。8x8 行列ごとに必要な反復回数は 1 回のみです。
- Y 代替 3 は、インテル® AVX-512 コードで、2 ソースのパーミュート命令を使用します。このコードは最初にパーミュート・マスクをロードして、次に行列データをロードしていることに注目してください。パーミュート操作に使用されるマスクは、次の配列に保存されています。

```
short permMaskBuffer [8*8] = { 0, 8, 16, 24, 32, 40, 48, 56,
                               1, 9, 17, 25, 33, 41, 49, 57,
                               2, 10, 18, 26, 34, 42, 50, 58,
                               3, 11, 19, 27, 35, 43, 51, 59,
                               4, 12, 20, 28, 36, 44, 52, 60,
                               5, 13, 21, 29, 37, 45, 53, 61,
                               6, 14, 22, 30, 38, 46, 54, 62,
                               7, 15, 23, 31, 39, 47, 55, 63 };
```

各代替コードは、50 個の行列 (それぞれ 2 バイト要素の 8x8) を転置します。

例 13-12 三値論理と代替手法の比較

代替 1: スカラーコード	代替 2: インテル® AVX2 コード	代替 3: インテル® AVX-512 コード
<pre> mov rsi, pImage mov rdi, pOutImage xor rdx, rdx matrix_loop: xor rax, rax outerloop: xor rbx, rbx innerloop: mov rcx, rax shl rcx, 3 add rcx, rbx mov r8w, word ptr [rsi+rcx*2] mov rcx, rbx shl rcx, 3 add rcx, rax mov word ptr[rdi+rcx*2], r8w add rbx, 1 cmp rbx, 8 jne innerloop add rax, 1 cmp rax, 8 jne outerloop add rdx, 1 add rsi, 64*2 add rdi, 64*2 cmp rdx, 50 jne matrix_loop                     </pre>	<pre> mov rsi, pImage mov rdi, pOutImage xor rdx, rdx vmovdqa xmm0, [rsi] vmovdqa xmm1, [rsi+0x10] vmovdqa xmm2, [rsi+0x20] vmovdqa xmm3, [rsi+0x30] vinserti128 ymm0, ymm0,                 [rsi+0x40], 0x1 vinserti128 ymm1, ymm1,                 [rsi+0x50], 0x1 vinserti128 ymm2, ymm2,                 [rsi+0x60], 0x1 vinserti128 ymm3, ymm3,                 [rsi+0x70], 0x1 vpunpcklwd ymm4, ymm0, ymm1 vpunpckhwd ymm5, ymm0, ymm1 vpunpcklwd ymm6, ymm2, ymm3 vpunpckhwd ymm7, ymm2, ymm3 vpunpckldq ymm0, ymm4, ymm6 vpunpckhdq ymm1, ymm4, ymm6 vpunpckldq ymm2, ymm5, ymm7 vpunpckhdq ymm3, ymm5, ymm7 vpermq ymm0, ymm0, 0xD8 vpermq ymm1, ymm1, 0xD8 vpermq ymm2, ymm2, 0xD8 vpermq ymm3, ymm3, 0xD8 vmovdqa [rdi], ymm0 vmovdqa [rdi+0x20], ymm1 vmovdqa [rdi+0x40], ymm2 vmovdqa [rdi+0x60], ymm3 add rdx, 1 add rsi, 64*2 add rdi, 64*2 cmp rdx, 50 jne matrix_loop                     </pre>	<pre> mov rax, permMaskBuffer vmovdqa32 zmm10, [rax] vmovdqa32 zmm11, [rax+0x40] mov rsi, pImage mov rdi, pOutImage xor rdx, rdx vmovdqa32 zmm2, [rsi] vmovdqa32 zmm3, [rsi+0x40] vmovdqa32 zmm0, zmm10 vmovdqa32 zmm1, zmm11 vpermi2w zmm0, zmm2, zmm3 vpermi2w zmm1, zmm2, zmm3 vmovdqa32 [rdi], zmm0 vmovdqa32 [rdi+0x40], zmm1 add rdx, 1 add rsi, 64*2 add rdi, 64*2 cmp rdx, 50 jne matrix_loop                     </pre>
ベースライン 1x	スピードアップ: 13.7x	スピードアップ: 37.3x (インテル® AVX2 コードと比べて 2.7x)

## 13.9 ブロードキャスト処理

### 13.9.1 組み込みブロードキャスト

インテル® AVX-512 では、ブロードキャストなしの命令シンタックス内でも暗黙的にブロードキャスト操作を適用できる、組み込みブロードキャスト操作が実装されています。メモリーからのソースをブロードキャストできます。つまり、有効なソースオペランドのすべての要素に渡って、追加のソースレジスターを使用することなく、32 ビット・データ要素では最大 16 回 (64 ビット・データでは最大 8 回) 繰り返されます。これは、ベクトル命令のすべての操作で同じスカラーオペランドを繰り返して使用する際に便利です。

組込みブロードキャストは、要素サイズが 32 または 64 ビットの命令で利用できます。バイトとワード要素のブロードキャストは、組込みブロードキャストではサポートされません。そのためバイトとワードのブロードキャストには、通常のブロードキャストを使用してください。

組込みブロードキャストは使用するレジスター数が少ないため、レジスター・プレッシャーが高いコードでは特に有効です。

さらに、組込みブロードキャストを使用すると、ロード `μop` は操作を行う `μop` と同じ命令にあるため、マイクロフュージョンの恩恵を受けることができます。

例えば、次のコードの置き換えについて考えてみます。

```
vbroadcastss zmm3, [rax]
vmulps zmm1, zmm2, zmm3
```

置換後:

```
vmulps zmm1, zmm2, [rax] {1to16}
```

{1to16} プリミティブでは、次のことが行われます。

1. メモリーから float32 (単精度) 要素を 1 つロードします。
2. 16 個の 32 ビット浮動小数点要素を形成するため、16 回繰り返します。

インテル® AVX-512 命令のストア・セマンティクスと純粋なロード命令は、ブロードキャスト・プリミティブをサポートしていません。

### 13.9.2 ロードポートで実行されるブロードキャスト

Skylake Server<sup>†</sup> マイクロアーキテクチャーでは、32 ビット以上のメモリーオペランドを持つブロードキャスト命令は、ロードポートで実行されます。シャッフルのようにポート 5 では実行されません。次の代替 2 の例では、ロードポートでブロードキャストを実行すると、どのようにポート 5 のワークロードが減少して、パフォーマンスが向上するか示しています。代替 3 は、組込みブロードキャストが、ロードポートとマイクロフュージョンでブロードキャストを実行する利点を示します。

例 13-13 ロードポートでブロードキャストを代替実行

代替 1: 32 ビットのロード操作とレジスター・ブロードキャスト	代替 2: 32 ビット・メモリー・オペランドとブロードキャスト	代替 3: 32 ビット組込みブロードキャスト
<pre>loop: vmovd xmm0, [rax] vpbroadcastd zmm0, xmm0 vpadd zmm2, zmm1, zmm0 vpermd zmm2, zmm3, zmm2 inc rax sub rdx, 0x1 jnz loop</pre>	<pre>loop vpbroadcastd zmm0, [rax] vpadd zmm2, zmm1, zmm0 vpermd zmm2, zmm3, zmm2 inc rax sub rdx, 0x1 jnz loop</pre>	<pre>loop: vpadd zmm2, zmm1, [rax]{1to16} vpermd zmm2, zmm3, zmm2 inc rax sub rdx, 0x1 jnz loop</pre>
ベースライン 1x	スピードアップ: 1.57x	スピードアップ: 1.9x

次の例は、Skylake Server<sup>†</sup> マイクロアーキテクチャーでは、16 ビット・ブロードキャストがポート 5 で実行されるため、メモリー・オペランド・ブロードキャストからは利点を得られないことを示しています。



例 13-14 ポート 5 での 16 ビット・ブロードキャストの実行

代替 1: 16 ビット・ロードとレジスター・ブロードキャスト	代替 2: 16 ビット・メモリー・オペランドとブロードキャスト
<pre>loop: vmovd xmm0, [rax] vpbroadcastw zmm0, xmm0 vpaddw zmm2, zmm1, zmm0 vpermw zmm2, zmm3, zmm2 inc rax sub rdx, 0x1 jnz loop</pre>	<pre>loop: vpbroadcastw zmm0, [rax] vpaddw zmm2, zmm1, zmm0 vpermw zmm2, zmm3, zmm2 inc rax sub rdx, 0x1 jnz loop</pre>
ベースライン 1x	スピードアップ: ベースラインと同じ

組込みブロードキャストは、16 ビット・メモリー・オペランドではサポートされないことに注意してください。

### 13.10 組込み丸め操作

デフォルトでは、丸めモードは MXCSR レジスターの 13:14 ビットで設定されます。

インテル® AVX-512 では、(命令ごとの) 静的丸めモード (RM) や丸めモードの上書きと呼ばれる新しい命令属性が導入されています。この属性は、MXCSR の RM ビットの値にかかわらず、特定の演算丸めモードを適用することを可能にします。丸めモードと組み合わせると、インテル® AVX-512 はまた SAE (“suppress-all-exceptions” - すべての例外を抑制) 属性を持っています。これにより、MXCSR のすべての浮動小数点例外フラグを無効化できます。丸めモードが有効であると SAE は常に適用されます。

静的丸めモードと SAE の制御は、レジスター-レジスターベクトル命令の EVEX.b ビットを 1 に設定することで、命令のエンコードで有効にできます。この場合、ベクトル長は最大ベクトル長 (インテル® AVX-512 では 512 ビット) であると想定されます。次の表にインテル® AVX-512 で設定可能な静的丸めモードの一覧を示します。いくつかの命令では、静的に指定される即値ビットで、すでに丸めモードが設定されていることに注意してください。この場合、即値ビットは MXCSR.RM のビットに優先されるように、組込み丸めモードよりも優先されます。

#### 13.10.1 静的丸めモード

静的丸めモードの機能と説明を以下に示します。

表 13-3 静的丸めモードの機能

機能	説明
{rne}	最も近い偶数に丸める + SAE
{rd}	切り下げ (負の無限大へ) + SAE
{ru}	切り上げ (正の無限大へ) + SAE
{rz}	ゼロへ丸め (切り捨て) + SAE

次に利用例を示します。

例 13-15 組込みと非組込み丸め

組込み丸めを使用	組込み丸めを使用しない
<pre>vaddps zmm7 {k6}, zmm2, zmm4, {ru}</pre>	<pre>; rax &amp; rbx は、MXCSR 値のロードとセーブ（復元用）に使用されるメモリー内の一次的な dword 値を指します  vstmxcsr [rax] ; mxcsr 値をメモリーへロード mov ebx, [rax] ; レジスターへ移動 and ebx, 0xFFFF9FFF ; ゼロ RM ビット または ebx, 0x5F80 ; {ru} を RM ビットに設定してすべての例外を抑制 mov [rcx], ebx ; 新しい値をメモリーへ移動 vldmxcsr [rcx] ; MXCSR へ保存  vaddps zmm7 {k6}, zmm2, zmm4 ; 自身を操作  vldmxcsr [rax] ; 以前の MXCSR 値をリストア</pre>

このコードは、ベクトル zmm2 と zmm4 の単精度浮動小数数を正の無限大への切り上げで加算しており、結果は k6 の条件付き書き込みマスクによってベクトル zmm7 に生成されます。MXCSR.RM ビットはこの命令では無視され、結果に影響しないことに注意してください。

以下は静的丸めモードが許されない命令の例です。

```
; 丸めモードは命令即値で設定済み
vrndscaleps zmm7 {k6}, zmm2 {rd}, 0x00

; メモリーオペランド付き命令
vmulps zmm7 {k6}, zmm2, [rax] {rd}

; 最大ベクトル長 (512 ビット) とは異なるベクトル長を持つ命令
vaddps ymm7 {k6}, ymm2, ymm4 {rd}

; 浮動小数点命令なし
vpadd zmm7 {k6}, zmm2, zmm4 {rd}
```

### 13.11 スキャッター命令

この命令は連続しないデータのストア（散布）を実行します。命令は、与えられたベースアドレス、符号付きのオフセット、およびデータ項目により、データレジスターの各要素を、ベースアドレスとオフセットから計算されるメモリー・ロケーションに書き込みます。命令は、ベースアドレスとインデックス・ベクトルで示されるメモリー・ロケーションに、ダブルワード・ベクトルでは最大 16 要素（qword インデックスでは 8 要素）、またはクワッドワード・ベクトルでは最大 8 要素をストアします。各要素は対応するマスクビットが 1 の場合にのみストアされます。図 13-9 は次の操作を示しています。

```
vscatterdpd [rax + zmm0]{k1}, zmm1
```

この例では、rax にはベースアドレス、zmm0 にはオフセット、そして zmm1 には書き込むデータが含まれます。

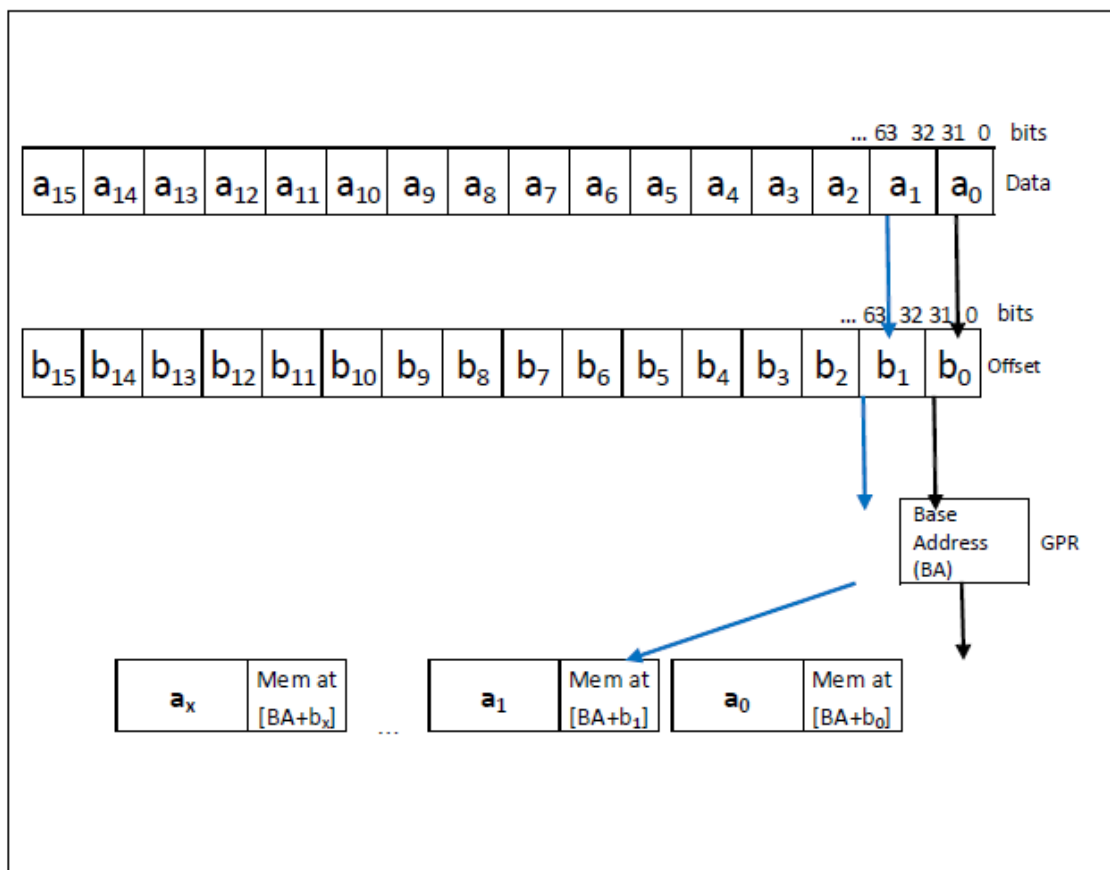


図 13-9 vscatterdpd 命令の操作

### 13.11.1 データ・スキッターの例

与えられた 0 から N の範囲のユニークなインデックスの配列で、long long 整数 (64 ビット) から浮動小数点数 (32 ビット) へ値を変換しながら、対応するインデックスに従って N 個の配列をソートします。

```
for ( int i=0; i < N; i++ )
{
dst[ ind [i] ] = (float)src[i];
}
```

上記のコードの 3 つの実装例を示します。

- Y 代替 1 はスカラーコードです。
- Y 代替 2 はスキッターのソフトウェア・シーケンスです。
- Y 代替 3 は、ハードウェア・スキッターです。

#### 注意

ハードウェア・スキッター操作では、ベクトルの要素数と同じ数のストア操作が発行されます。連続した要素のストアにはスキッターを使用せず、vmov 命令を使用してストアを行います。

例 13-16 組込みと非組込み丸め

<p><b>スカラー</b></p> <pre> mov rax, pImage //入力 mov rcx, pOutImage //出力 mov rbx, pIndex //インデックス mov rdx, len //レングス xor r9, r9  mainloop: mov r9d, [rbx+rdx-0x4] vcvtss2ss xmm0, xmm0, qword ptr [rax+rdx*2-0x8] vmovss [rcx+r9*4], xmm0 sub rdx, 4 jnz mainloop                     </pre>	
<p><b>ベースライン 1x</b></p>	
<p><b>ソフトウェア・シーケンス</b></p> <pre> mov rax, pImage //入力 mov rcx, pOutImage //出力 mov rbx, pIndex //インデックス mov rdx, len //レングス mov r9, shufMaskP vmovaps ymm2, [r9] mainloop: vmovaps zmm1, [rax + rdx*2 - 0x80] //データロード vcvtuqq2ps ymm0, zmm1 //float ^変換 movsxd r9, [rbx + rdx - 0x40] //8 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x3c] //7 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x38] //6 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x34] //5 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x30] //4 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x2c] //3 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0                     </pre>	<p><b>ハードウェア・スカッター</b></p> <pre> mov rax, pImage //入力 mov rcx, pOutImage //出力 mov rbx, pIndex //インデックス mov rdx, len //レングス mainloop: vmovdqa32 zmm0, [rbx+rdx-0x40] vmovdqa32 zmm1, [rax+rdx*2-0x80] vcvtuqq2ps ymm1, zmm1 vmovdqa32 zmm2, [rax+rdx*2-0x40] vcvtuqq2ps ymm2, zmm2 vshuff32x4 zmm1, zmm1, zmm2, 0x44 kxnorw k1,k1,k1 vscatterdps [rcx+4*zmm0] {k1}, zmm1 sub rdx, 0x40 jnz mainloop                     </pre>

<pre> movsxd r9, [rbx + rdx - 0x28] //2 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x24] //最初のインデックスをロード vmovss [rcx + 4*r9], xmm0 vmovaps zmm1, [rax + rdx*2 - 0x40] //データロード vcvtuqq2ps ymm0, zmm1 //float へ変換 movsxd r9, [rbx + rdx - 0x20] //8 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x1c] //7 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x18] //6 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x14] //5 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x10] //4 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0xc] //3 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x8] //2 番目のインデックスをロード vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x4] //最初のインデックスをロード vmovss [rcx + 4*r9], xmm0 sub rdx, 0x40 jnz mainloop </pre>	
<p><b>スピードアップ:</b> 1.48x</p>	<p><b>スピードアップ:</b> 1.53x</p>

## 13.12 静的丸めモード、すべての例外を抑制 (SAE)

インテル® AVX-512 の浮動小数点命令には、すべての例外を抑制する機能 (SAE) が導入されています。この機能は、スプリアスフラグの設定が望ましくない場合に有用です。現在のベクトル数学関数の実装では、通常スプリアスフラグの設定を許容するため、例外が有効化されたアプリケーションの実行で問題が生じることがあります。標準化に準拠するコードはスプリアスフラグの設定を許可しません。

静的丸めモードには、規格化された用途 (IEEE、OpenCL) に加え、デフォルトの丸めモード (動的に設定可能) で動作する数学ライブラリーのアプリケーションがあります。

## 13.13 QWORD 命令のサポート

インテル® AVX-512 では、インテル® AVX とインテル® AVX2 で導入された多くの命令の QWORD 拡張をサポートします。QWORD は次の命令でサポートされます。

### 13.13.1 算術命令での QUADWORD サポート

インテル® AVX-512 では、`vpmassq`、`vpmassuq`、`vpmasssq`、`vpmassuq`、および `vpmullq` 命令に新たな `quadword` のサポートが追加されました。次の例は、2 つの 64 ビット数の加算と乗算の最大値を配列 `C` に格納しています。

```
const int N = miBufferWidth;
const __int64* restrict a = A;
const __int64* restrict b = B;
__int64* restrict c = Cref;

for (int i = 0; i < N; i++){
    int64 sum = a[i] + b[i];
    int64 mul = a[i] * b[i];
    c[i] = mul > sum ? mul : sum;
}
```

次のコードは、新たな機能がどのように命令数をインテル® AVX2 の 118 からインテル® AVX-512 の 30 に減らし、3.1 倍のスピードアップをもたらすかを示します。

例 13-17 インテル® AVX2 とインテル® AVX-512 での QWORD の例

インテル® AVX2 組込み関数	インテル® AVX-512 組込み関数
<pre>for (int i = 0; i &lt; N; i+= 32){     __m256i aa, bb, aah, bbh, mul, sum;     #pragma unroll(8)     for (int j = 0; j &lt; 8; j++){         aa = _mm256_loadu_si256((const             __m256i*)(a+i+4*j));         bb = _mm256_loadu_si256((const             __m256i*)(b+i+4*j));         sum = _mm256_add_epi64(aa, bb);         mul = _mm256_mul_epu32(aa, bb);         aah = _mm256_srli_epi64(aa, 32);         bbh = _mm256_srli_epi64(bb, 32);         aah = _mm256_mul_epu32(aah, bb);         bbh = _mm256_mul_epu32(bbh, aa);         aah = _mm256_add_epi32(aah, bbh);         aah = _mm256_slli_epi64(aah, 32);         mul = _mm256_add_epi64(mul, aah);         aah = _mm256_cmpgt_epi64(mul, sum);         aa=(__m256i)_mm256_blendv_pd(             (__m256d)sum,             (__m256d)mul, (__m256d)aah);         _mm256_storeu_si256((__m256i*)(c+4*j),aa);     }     c += 32; }</pre>	<pre>for (int i = 0; i &lt; N; i+= 32){     __m512i aa, bb, mul, sum;     #pragma unroll(4)     for (int j = 0; j &lt; 4; j++){         aa = _mm512_loadu_si512((const             __m512i*)(a+i+8*j));         bb = _mm512_loadu_si512((const             __m512i*)(b+i+8*j));         sum = _mm512_add_epi64(aa, bb);         mul = _mm512_mullo_epi64(aa, bb);         aa = _mm512_max_epi64(sum, mul);         _mm512_storeu_si512((__m512i*)(c+8*j),a             a);     }     c += 32; }</pre>
<p>ベースライン 1x</p>	<p>スピードアップ: 3.1x</p>
<p>インテル® AVX2 アセンブリ</p>	<p>インテル® AVX-512 アセンブリ</p>
<pre>loop: vmovdqu32 ymm28,ymmword ptr[rax+rcx*8+0x20] inc r9d vmovdqu32 ymm26,ymmword ptr[r11+rcx*8+0x20] vmovdqu32 ymm17,ymmword ptr[r11+rcx*8] vmovdqu32 ymm19,ymmword ptr[rax+rcx*8] vmovdqu ymm13,ymmword ptr[rax+rcx*8+0x40] vmovdqu ymm11,ymmword ptr[r11+rcx*8+0x40] vpsrlq ymm25,ymm28,0x20 vpsrlq ymm27,ymm26,0x20 vpsrlq ymm16,ymm19,0x20 vpsrlq ymm18,ymm17,0x20 vpaddq ymm6,ymm28,ymm26 vpsrlq ymm10,ymm13,0x20 vpsrlq ymm12,ymm11,0x20 vpaddq ymm0,ymm19,ymm17 vpmuludq ymm29,ymm25,ymm26 vpmuludq ymm30,ymm27,ymm28 vpadd ymm31,ymm29,ymm30 vmovdqu32 ymm29,ymmword ptr[r11+rcx*8+0x80] vpsllq ymm5,ymm31,0x20 vmovdqu32 ymm31,ymmword ptr[rax+rcx*8+0x80] vpsrlq ymm30,ymm29,0x20 vpmuludq ymm20,ymm16,ymm17</pre>	<pre>loop: vmovups zmm0,ymmword ptr[rax+rcx*8] inc r9d vmovups zmm5,ymmword ptr[rax+rcx*8+0x40] vmovups zmm10,ymmword ptr[rax+rcx*8+0x80] vmovups zmm15,ymmword ptr[rax+rcx*8+0xc0] vmovups zmm1,ymmword ptr[r11+rcx*8] vmovups zmm6,ymmword ptr[r11+rcx*8+0x40] vmovups zmm11,ymmword ptr[r11+rcx*8+0x80] vmovups zmm16,ymmword ptr[r11+rcx*8+0xc0] vpaddq zmm2,zmm0,zmm1 vpmullq zmm3,zmm0,zmm1 vpaddq zmm7,zmm5,zmm6 vpmullq zmm8,zmm5,zmm6 vpaddq zmm12,zmm10,zmm11 vpmullq zmm13,zmm10,zmm11 vpaddq zmm17,zmm15,zmm16 vpmullq zmm18,zmm15,zmm16 vpmaxsq zmm4,zmm2,zmm3 vpmaxsq zmm9,zmm7,zmm8 vpmaxsq zmm14,zmm12,zmm13 vpmaxsq zmm19,zmm17,zmm18 vmovups zmmword ptr[rsi+r10*1],zmm4 vmovups zmmword ptr[rsi+r10*1+0x40],zmm9</pre>

<pre> vpmuludq ymm21, ymm18, ymm19 vpmuludq ymm4, ymm28, ymm26 vpaddq ymm22, ymm20, ymm21 vpaddq ymm7, ymm4, ymm5 vpsrlq ymm28, ymm31, 0x20 vmovdqu32 ymm20, ymmword ptr [r11+rcx*8+0x60] vpsllq ymm24, ymm22, 0x20 vmovdqu32 ymm22, ymmword ptr [rax+rcx*8+0x60] vpsrlq ymm21, ymm20, 0x20 vpaddq ymm4, ymm22, ymm20 vpcmpgtq ymm8, ymm7, ymm6 vblendvpd ymm9, ymm6, ymm7, ymm8 vmovups ymmword ptr [rsi+r10*1+0x20], ymm9 vpmuludq ymm14, ymm10, ymm11 vpmuludq ymm15, ymm12, ymm13 vpmuludq ymm8, ymm28, ymm29 vpmuludq ymm9, ymm30, ymm31 vpmuludq ymm23, ymm19, ymm17 vpaddq ymm16, ymm14, ymm15 vpsrlq ymm19, ymm22, 0x20 vpaddq ymm10, ymm8, ymm9 vpaddq ymm1, ymm23, ymm24 vpsllq ymm18, ymm16, 0x20 vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0xc0] vpsllq ymm12, ymm10, 0x20 vpmuludq ymm23, ymm19, ymm20 vpmuludq ymm24, ymm21, ymm22 vpaddq ymm25, ymm23, ymm24 vmovdqu32 ymm19, ymmword ptr [rax+rcx*8+0xa0] vpsllq ymm27, ymm25, 0x20 vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm16, ymm19, 0x20 vpcmpgtq ymm2, ymm1, ymm0 vblendvpd ymm3, ymm0, ymm1, ymm2 vpaddq ymm0, ymm13, ymm11 vmovups ymmword ptr [rsi+r10*1], ymm3 vpmuludq ymm17, ymm13, ymm11 vpmuludq ymm11, ymm31, ymm29 vpaddq ymm1, ymm17, ymm18 vpaddq ymm13, ymm31, ymm29 vpaddq ymm14, ymm11, ymm12 vmovdqu32 ymm17, ymmword ptr [r11+rcx*8+0xa0] vmovdqu ymm12, ymmword ptr [r11+rcx*8+0xe0] vpsrlq ymm18, ymm17, 0x20 vpcmpgtq ymm2, ymm1, ymm0 vpmuludq ymm26, ymm22, ymm20 vpcmpgtq ymm15, ymm14, ymm13 vblendvpd ymm3, ymm0, ymm1, ymm2 vblendvpd ymm0, ymm13, ymm14, ymm15 vmovdqu ymm14, ymmword ptr [rax+rcx*8+0xe0] vmovups ymmword ptr [rsi+r10*1+0x40], ymm3 vmovups ymmword ptr [rsi+r10*1+0x80], ymm0 vpaddq ymm5, ymm26, ymm27 vpsrlq ymm11, ymm14, 0x20 </pre>	<pre> vmovups zmmword ptr [rsi+r10*1+0x80], zmm14 vmovups zmmword ptr [rsi+r10*1+0xc0], zmm19 add rcx, 0x20 add rsi, 0x100 cmp r9d, r8d jnb loop </pre>
--	---



<pre> vpsrlq ymm13, ymm12, 0x20 vpaddq ymm1, ymm19, ymm17 vpaddq ymm0, ymm14, ymm12 vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0xc0] vpmuludq ymm20, ymm16, ymm17 add rcx, 0x20 vpmuludq ymm21, ymm18, ymm19 vpadd ymm22, ymm20, ymm21 vpsrlq ymm27, ymm26, 0x20 vpsllq ymm24, ymm22, 0x20 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpmuludq ymm15, ymm11, ymm12 vpmuludq ymm16, ymm13, ymm14 vpmuludq ymm23, ymm19, ymm17 vpadd ymm31, ymm29, ymm30 vpadd ymm17, ymm15, ymm16 vpaddq ymm2, ymm23, ymm24 vpsllq ymm19, ymm17, 0x20 vpcmpgtq ymm6, ymm5, ymm4 vblendvpd ymm7, ymm4, ymm5, ymm6 vpsllq ymm6, ymm31, 0x20 vmovups ymmword ptr [rsi+r10*1+0x60], ymm7 vpaddq ymm7, ymm28, ymm26 vpcmpgtq ymm3, ymm2, ymm1 vpmuludq ymm5, ymm28, ymm26 vpmuludq ymm18, ymm14, ymm12 vblendvpd ymm4, ymm1, ymm2, ymm3 vpaddq ymm8, ymm5, ymm6 vpaddq ymm1, ymm18, ymm19 vmovups ymmword ptr [rsi+r10*1+0xa0], ymm4 vpcmpgtq ymm9, ymm8, ymm7 vpcmpgtq ymm2, ymm1, ymm0 vblendvpd ymm10, ymm7, ymm8, ymm9 vblendvpd ymm3, ymm0, ymm1, ymm2 vmovups ymmword ptr [rsi+r10*1+0xc0], ymm10 vmovups ymmword ptr [rsi+r10*1+0xe0], ymm3 add rsi, 0x100 cmp r9d, r8d jb loop                 </pre>	<p>ベースライン 1x</p>
<p>スピードアップ: 3.1x</p>	

### 13.13.2 変換命令での QUADWORD サポート

次の表は、変換命令のクワッドワード拡張を示しています。

表 13-4 ベクトル Quadword 拡張

From / To	ベクトル SP	ベクトル DP	ベクトル int64	ベクトル uint64
ベクトル SP	-		vcvtps2qq	vcvtps2uqq
ベクトル DP		-	vcvtpd2qq	vcvtpd2qq
ベクトル int64	vcvtqq2ps	vcvtqq2pd	-	
ベクトル uint64	vcvtqq2ps	vcvtuqq2pd		-

表 13-5 スカラー Quadword 拡張

From / To	スカラー SP	スカラー DP	スカラー int64	スカラー uint64
スカラー SP	-		vcvtss2si	vcvtss2usi
スカラー DP		-	vcvtss2sd	vcvtss2usd
スカラー int64	vcvtss2sd	vcvtss2sd	-	
スカラー uint64	vcvtusi2sd	vcvtusi2sd		-

### 13.13.3 切り捨て変換命令での QUADWORD サポート

次の表は、切り捨てモードの変換命令のクワッドワード拡張を示しています。

表 13-6 ベクトル Quadword 拡張

From / To	ベクトル int64	ベクトル uint64
ベクトル SP	vcvttps2qq	vcvttps2uqq
ベクトル DP	vcvttpd2qq	vcvttpd2uqq

表 13-7 スカラー Quadword 拡張

From / To	スカラー int64	スカラー uint64
スカラー SP	vcvtss2sd	vcvtss2usd
スカラー DP	vcvtss2sd	vcvtss2usd

### 13.14 ベクトル長の直交性

ベクトル長拡張 (VL) をサポートするプロセッサでは、すべてのインテル® AVX-512 命令が 128 ビット、256 ビット、そして 512 ビットのベクトルを操作できます。組込み丸めを伴う命令を除き、これら 3 つのベクトル長がインテル® AVX-512 命令でサポートされます。命令のエンコードにおいて、ベクトル長と組込み丸め制御は同じ 2 つのビットを使用してコード化されているため、組込み丸めが使用される場合ベクトル長は自動的に 512 ビット (インテル® AVX-512 における最大ベクトル長) と仮定されます。

“組込み丸め” については 13.10 節をご覧ください。

### 13.15 超越計算サポート向けのインテル® AVX-512 命令

この節では、インテル® AVX-512 で導入された新しい超越計算向けの命令について説明します。

#### 13.15.1 VRCP14、VRSQRT14 - 1/x, x/y, sqrt(x) 向けのソフトウェア・シーケンス

シンタックス:

VRCP14PD/PS dest, src  
VRSQRT14PD/PS dest, src

##### 13.15.1.1 アプリケーションの例

逆数、除算、平方根、逆平方根向けのソフトウェア・シーケンス。

1/x, x/y, sqrt(x) 向けのソフトウェア・シーケンスは、スルーブットに利点があります (レイテンシーはそれほどでもなく、精度がかなり低いわけでもありません)。通常これらは、ニュートンラフソン近似や多項式近似によって実装されます。

VRCP14 と VRSQRT14 の利点の 1 つは、これまでの RCPPS や RSQRTPS に比べ精度が改善されていることです。これは、特に倍精度では計算を短縮するのに役立ちます (倍精度では 50 - 52 ビットの近似のため 3 つのニュートンラフソン反復が必要です)。

これらの命令のもう 1 つの利点は、倍精度バージョンがサポートされることです (RCP/RSQRT 命令ではサポートされていませんでした)。これらの機能により倍精度のパフォーマンスが向上します。Skylake Server<sup>†</sup> マイクロアーキテクチャー上では、倍精度逆数と平方根ソフトウェア・シーケンスは、512 ビット・ベクトル・モードの倍精度超越引数リダクション (log, cbrt など) における VDIV や VSQRT よりもかなり高いスループットをもたらします。

log() や cbrt() (立方根) などの関数では、高価な逆数テーブル索引に代わって丸められた VRCP14PD の結果を使用できます。同様の手法を RCPPS にも使用できますが、倍精度ではそれほど効果はありません。

log() 引数リダクションの例は、13.15.3 節「VRNDSCALE - ベクトル丸めスケール」をご覧ください。

## 13.15.2 VGETMANT、VGETEXP - ベクトル仮数とベクトル指数の取得

シンタックス:

```
VGETMANTPD/PS dest_mant, src, imm
VGETEXPPD/PS dest_exp, src
```

### 13.15.2.1 アプリケーションの例

対数関数。

```
log2(x) = VGETEXP(x) + log2(VGETMANT(x, 8))
log(x) = VGETEXP(x) * log(2.0) + log(VGETMANT(x, 8))
```

上記に見られるように、VGETMANT(x,8) がすべての有効な関数入力で [1,2) であり、不正な入力 (x < 0) が NaN であることが保証されると、log(VGETMANT(x,8)) の計算を軽減できます。

これは仮数の対数を計算するさまざまなアルゴリズムに適用できます。特定のアルゴリズムの選択は、必要とする精度、最適化の目標 (レイテンシーやスループットの最適化)、またはマイクロアーキテクチャーに依存します。いくつかのアルゴリズムでは、仮数の正規化にほかの手法を使用する可能性があります。[0.5, 1) または [0.75, 1.5); しかし、計算の基礎となる基本的な同一性は上に示されています。

$X^{\alpha}$  (alpha 定数) と除算の詳細については、13.15.5 節「VSCALEF - ベクトルスケール」をご覧ください。

## 13.15.3 VRNDSCALE - ベクトル丸めスケール

シンタックス:

```
VRNDSCALEPD/PS dest, src, imm
```

### 13.15.3.1 アプリケーションの例

テーブル索引は超越関数の実装でよく利用されます。テーブル・インデックスは、多くの場合入力の先頭数ビットに基づいています。テーブル・インデックスに対応する浮動小数点入力値を生成するため、引数リダクション処理の一部として VRNDSCALE 命令を使用できます。以下は、 $1 \leq x < 2$  における log(x) の引数リダクションの実装例です。

```
y = RCP14(x);           // y is in (0.5, 1]
y0=VRNDSCALE(y, k*16); // y0 には k 仮数ビットがあります
                        // (先頭 1 ビットを含みます)
R = x?y0 - 1;          // |R| < 2^-14+2-k.
```

そして、 $\log(x) = -\log(y0) + \log(1+R)$  となります。

$\log(1+R)$  は多項式で計算でき、 $\log(y0)$  は  $2k - 1 + 1$  要素または  $2k - 1$  要素のテーブル索引から求められます。追加のチェックは犠牲になります。

### 13.15.4 VREDUCE - ベクトルレデュース

シンタックス:

VREDUCEPD/PS dest, src, imm

#### 13.15.4.1 アプリケーションの例

VREDUCE の最も有益な点は、`exp2` や `pow` (`exp2` も含まれます) などの一般的な超越操作においてレイテンシーを軽減できることです。また、`atan()` などほかの超越関数でも使用できます。

13.15.5 節「VSCALEF - ベクトルスケール」もご覧ください。

### 13.15.5 VSCALEF - ベクトルスケール

シンタックス:

VSCALEFPD/PS dest, src1, src2

#### 13.15.5.1 アプリケーションの例

```
exp2 (2x)
exp2(x) = VSCALEF( 2VREDUCE(x, RD_mode), x)
```

$R(x) = VREDUCE(x, RD\_mode) = x - \text{floor}(x)$  は、 $[0, 1)$  の範囲にあります。 $2R(x)$  は、多項式近似や多項式近似とテーブル索引によって求められます。VSCALEF はオーバーフローとアンダーフローを適切に処理します。また、`exp()` の特殊ケース (入力は無限大など) も処理できるように定義されているため、ベクトル実装において特殊な処理パスを必要としません。VSCALEF を使用しないと、入力が非常に大きな場合に異なる処理パスが必要になります。

もはや明示的に指数を操作する必要はなく、VSCALEF はスループットを改善します。

```
Exp(x)
Exp(x) = VSCALEF( 2R(x), x*(1/log(2.0)),
```

ここで、

```
R(x) = x - log(2.0)*floor(x*(1/log(2.0)));
```

$\log(2.0)$  近似 (ネイティブの浮動小数点形式より長い) を使用することで、 $R(x)$  を正確に計算できます。

`exp2()` のように、VSCALEF を使用する利点は、スループットを改善し二次的な分岐を排除できることです。

$x^{\text{alpha}}$  (alpha 定数)

例えば、 $\text{alpha}=1/3$  (立方根 `cbrt`) の場合、この計算の基本的なリダクションは次のようになります。

```
xalpha = VSCALEF( (VGETMANT(x, imm))alpha?2VREDUCE (VGETEXP(x)*alpha, RD_mode),
VGETEXP(x)*alpha)
```

即値 (imm) の選択は alpha 定数の値を基にしています

除算:

$a/b = \text{VSCALEF}(\text{VGETMANT}(a, 0)/\text{VGETMANT}(b, 0), \text{VGETEXP}(a) - \text{VGETEXP}(b))$

このリダクションは分岐なしの除算実装を可能にし、オーバーフロー、アンダーフロー、および特殊入力 (ゼロ、無限大、デノーマル) を処理できます。

すべての非 NaN 入力では、 $|\text{VGETMANT}(x, 0)|$  は [1,2) の範囲にあります。

$\text{VGETMANT}(a, 0)/\text{VGETMANT}(b, 0)$  は、必要な精度で計算できます。

インテル® AVX-512 で利用可能な「すべての例外を抑制 (SAE)」する機能は、スプリアスフラグが設定されないことを確実にします。フラグは、計算の結果として正しく設定できます。ただし、ゼロ除算は追加の処理が必要となるため除外されます。

高い精度や IEEE の順守が必要である場合、ハードウェア命令が一般により高いパフォーマンス (特にレイテンシーに関して) をもたらします。

### 13.15.6 VFPCLASS - ベクトル浮動小数点クラス

シンタックス:

VFPCLASSPD/PS dest\_mask, src, imm

#### 13.15.6.1 アプリケーションの例

VFPCLASS は、特殊なケースを検出して直接処理を行うパスに導いたり、代替としてメインパスでマスク操作に使用されます。以下に、逆数シーケンスと平方根シーケンスの 2 つの例を示します。

$1/x$  計算の引数リダクションは、 $e=1-x*\text{RCP14}(x)$  です。 $x$  が  $\pm 0$  または  $\pm \text{Inf}$  である場合、この式は NaN と評価され、RCP14 は特殊ケースに正しい結果を返します。VFPCLASS は、 $x$  が  $\pm 0$  または  $\pm \text{Inf}$  では  $\text{mask}=1$  を設定し、それ以外の  $x$  には  $\text{mask}=0$  を設定することを可能にします。そして、このマスクを使用して RCP14 の出力 (特殊ケースの結果) や、RCP14 から始まる相反精密計算の結果 (一般的な入力) を選択できます。

同様に、RSQRT14 を基にする平方根計算では、 $x=\pm 0$  または  $x=+\text{Inf}$  のマスクを作成するため VFPCLASS を使用できます。

Pow(x,y) 関数:

$\text{pow}(x,y)=2y*\log_2(x)$  のメインパスは、 $x\neq 0$ 、 $x=\text{Inf}/\text{NaN}$ 、または  $y=\text{Inf}/\text{NaN}$  を処理しません。最初の VFPCLASS 操作は、 $x\neq 0$  や  $x=\text{Inf}/\text{NaN}$  のケースに  $\text{special\_x\_mask}=1$  を設定するために使用できます。2 番目の VFPCLASS 操作は、 $x\neq 0$  や  $x=\text{Inf}/\text{NaN}$  のケースに  $\text{special\_y\_mask}=1$  を設定するために使用します。どちらかのマスクが設定されると、2 番目のパスへ分岐します。

## 13.15.7 VPERM, VPERMI2, VPERMT2 - 小規模テーブル索引の実装

### 13.15.7.1 アプリケーションの例

数学ライブラリー関数は、頻繁にテーブル索引を使用して実装されます。ベクトルモードでは、大きなテーブル索引はベクトルギャザーが使用されます。小さなテーブル索引は、VPERM\* 命令を使用することで劇的に高速化できます。

VPERM\* 命令によるテーブル索引を使用することで劇的な高速化を達成できた超越関数には、単精度と倍精度の `exp()`、`log()`、`pow()` などがあります。

## 13.16 競合検出

インテル® AVX-512 競合検出命令は、インテル® AVX-512 基本命令とともに、ベクトルの依存関係の可能性のあるループを効率良くベクトル化することを可能にします。VPCONFLICT は、単一ベクトルレジスター内で要素の水平比較を行います。VPCONFLICT は、ベクトルレジスターの各要素と直前の要素をすべて比較して、その比較結果を出力します。水平比較はその他の用途にも利用できます。

ほかの競合検出命令は、比較結果の効率良い操作を考慮しています。VPLZCNT 命令により、ベクトル要素と一致する値を結合する際に使用されるレジスター内パーミュート操作の制御を生成できます。

### 13.16.1 競合検出とベクトル化

インテル® AVX-512CD 命令は、配列ポインター (`*ptr[i] += val[i]`) や `A[B[i]] += val[i]` などの間接アドレス配列など) を使用した読み込みと書き込みを伴うループを効率良くベクトル化することを可能にします。

次のようなヒストグラムの計算を考えてみます。

```
for(int i = 0; i < num_inputs; i++)
{
    histogram[input[i] & (num_bins - 1)]++;
}
```

`input[0] = input[1] = 3` である場合、SIMD 命令を使用して `histogram[input[0]]` と `histogram[input[1]]` をレジスターへ (ギャザーを使用して) 読み込み、インクリメントを行って書き戻す (スキャッターを使用して) と、不正な結果がもたらされます。この操作を行うと `histogram[3]` の値は 1 になります。本来は 2 でなければなりません。

この問題はインデックスが複製されることで生じます。反復 0 の `histogram` への書き込みと反復 1 の `histogram` からの読み込みに依存関係がありますが、読み込みは以前の書き込みの値を取得する必要があります。

この問題を特定するため、VPCONFLICT 命令を使用してインデックス (またはポインター値) の複製を検出します。この命令はレジスター内のすべての以前の要素とベクトルレジスターの各要素を比較します。

例:

```
vpconflictd zmm0, zmm1
```

以下の図は、VPCONFLICTD 命令の実行例を示しています。入力 ZMM1 は 16 個の整数要素を含みます (青色のボックス)。図の上部にある ZMM1 は、左に示されるように転移しています。白いボックスは、ハードウェアが ZMM1 の異なる要素を比較した結果を示しています (0 = 等しくない、1 = 等しい)。命令からのそれぞれの比較出力は単一のビットです。実行されなかった比較 (灰色のボックス) は、単一ビット '0' を生成します。最終的に結果は ZMM0 に生成されます (図の下にある黄色のボックス)。各要素は上記のビットに相当する 10 進数で表されています。

VPCONFLICT は異なる方法でループをベクトル化するのに役立ちます。

最も簡単なものは、特定の SIMD レジスターのインデックスの複製をチェックすることです。複製されていないければ、すべての要素を安全に SIMD 命令で計算できます。競合が検出された場合、そのグループの要素をスカラーループで実行します。

インデックスの複製がそれほど多くない場合、ループのスカラーバージョンへの分岐はうまく動作します。しかし、ベクトル化されたループ反復が複製できるほど十分に大きければ、可能な限り並列性を高めるためできるだけ多くの SIMD 命令を使用する方がいいでしょう。

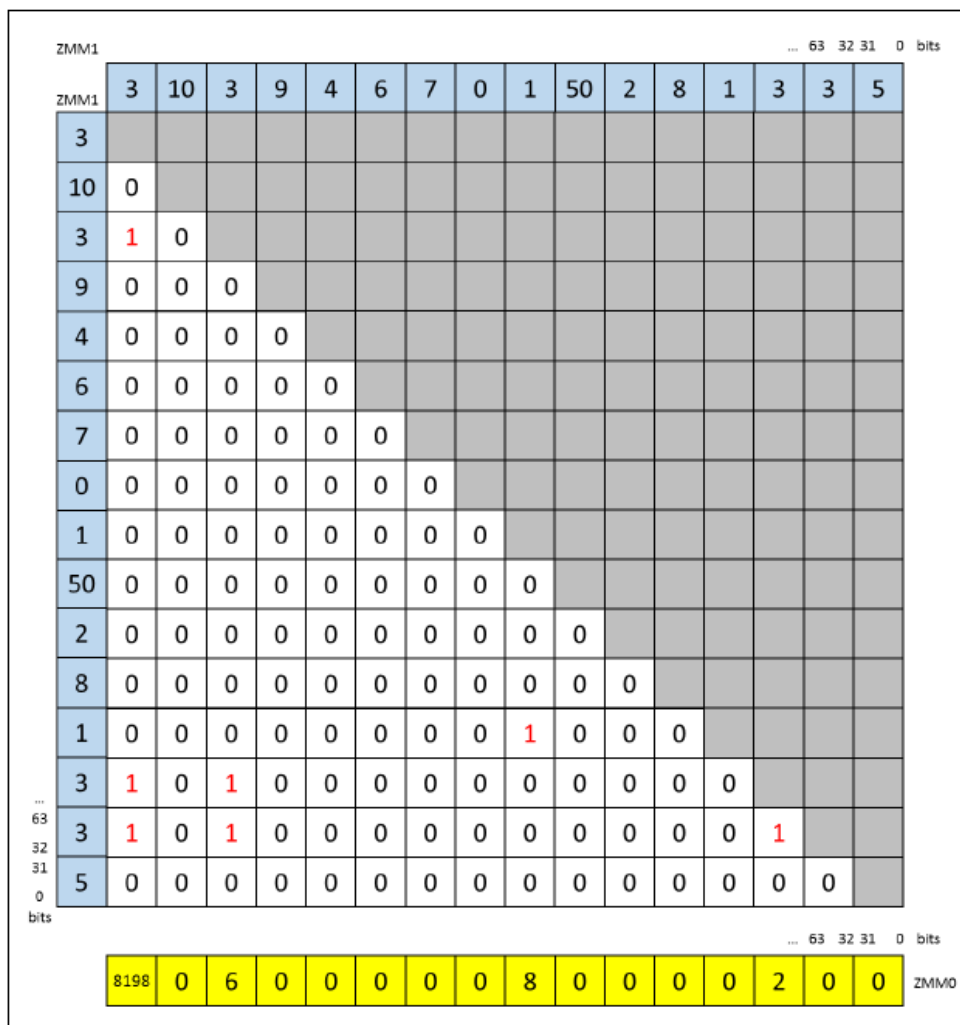


図 13-10 VPCONFLICTD 命令の実行

ヒストグラムの例のようにメモリー・ロケーションを更新するループでは、データがレジスター内に存在する間に個別のインデックスを使用して更新をマージし、それぞれのメモリー・ロケーションに一度だけ書き込みを行うことで、ストア・ロード・フォワードを最小限にできます。さらに、マージは並列に実行できます。

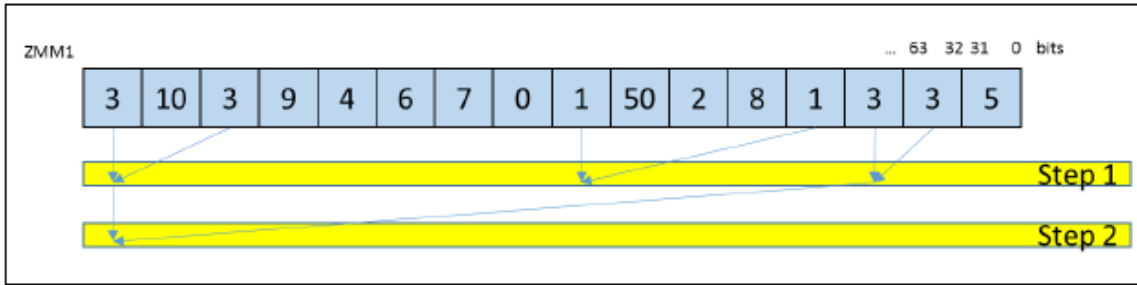


図 13-11 VPCONFLICTD マージ処理

図 13-11 は、インデックスのセットの例をマージする様子を示しています。図ではインデックスのみが示されていますが、実際には値をマージします。ほとんどのインデックスはユニークであり、マージする必要はありません。ステップ 1 では 3 組のインデックスを結合します: 2 つの '3' のペアと 1 つの '1' のペア。ステップ 2 は、ステップ 1 の '3' 向けの中間結果を結合します。これにより、それぞれのインデックスは 1 つの値のみを指します。ステップ 2 のみで、3 つのインデックス値を持つ 4 つの要素がマージされていることに注目してください。これは、ツリーのリダクションを行ったことで、結果のペアまたは各ステップの中間結果をマージしたことによるものです。

上記に示すマージ処理 (結合やリダクション) は、一連のパーミュート操作によって行われます。

最初のパーミュート制御は、VPLZCNT + VPSUB シーケンスで生成されます。VPLZCNT は、それぞれのベクトル要素に先行するゼロ数を供給します (例えば、最上位ビットに隣接する連続したゼロなど)。各ベクトル要素のビット数から VPLZCNT の結果を引くことで、VPCONFLICT 命令の結果の再上位 1 ビットの位置を示し、または -1 であれば要素間に競合しないことを示します。上記の例によるシーケンスの結果、次のパーミュート制御が得られます。

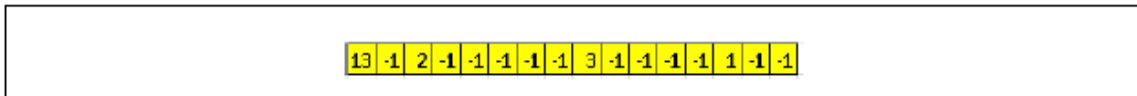


図 13-12 VPCONFLICTD パーミュート制御

一致するインデックスをマージするパーミュート・ループと、次のパーミュート・インデックスのセットを生成する処理は、パーミュート制御の値がすべて '-1' と等しくなるまで繰り返されます。

以下のアセンブリ・コードは、スカラーバージョンのヒストグラム・ループとベクトルバージョンのツリー・リダクションを示しています。ループはわずかな計算しか含んでいないためスピードアップは控え目です。SIMD の利点は論理積和操作とインクリメントをベクトル化することで得られています。ループがさらにベクトル化可能な計算を含んでいれば、SIMD のスピードアップはより高くなります。



例 13-18 スキャッター実装の代替

スカラーコード (2 回アンロール)	インテル® AVX-512 コード
<pre> mov r9d, bins_minus_1 mov ebx, num_inputs mov r10, pInput mov r15, pHistogram xor rax, rax histogram_loop:   lea ecx, [rax + rax]   incl eax   movslq rcx, ecx   movl esi, [r10+rcx*4]   andl esi, r9d   movl r8d, [r10+rcx*4+4]   movslq rsi, esi   andl r8d, r9d   movslq r8, r8d   incl [r15+rsi*4]   incl [r15+r8*4]   cmpl eax, ebx   jb histogram_loop </pre>	<pre> vmovaps zmm4, all_1 // {1, 1, ..., 1} vmovaps zmm5, all_negative_1 vmovaps zmm6, all_31 vmovaps zmm7, all_bins_minus_1 mov ebx, num_inputs mov r10, pInput mov r15, pHistogram histogram_loop:   vpandd zmm3, [r10+rcx*4], zmm7   vpconflictd zmm0, zmm3   kxnorw k1, k1, k1   vmovaps zmm2, zmm4   vpxord zmm1, zmm1, zmm1   vpgatherdd zmm1{k1}, [r15+zmm3*4]   vptestmd k1, zmm0, zmm0   kortestw k1, k1   je update    vplzcntd zmm0, zmm0   vpsubd zmm0, zmm6, zmm0    conflict_loop:     vpermd zmm8{k1}{z}, zmm2, zmm0     vpermd zmm0{k1}, zmm0, zmm0     vpaddd zmm2{k1}, zmm2, zmm8     vpcmpd k1, 4, zmm5, zmm0     kortestw k1, k1     jne conflict_loop    update:     vpaddd zmm0, zmm2, zmm1     kxnorw k1, k1, k1     addq rcx, 16     vpscatterdd [r15+zmm3*4]{k1}, zmm0     cmpl ebx, ecx     jb histogram_loop </pre>
スカラーのベースライン 1x	スピードアップ: 1.11x (ランダム入力); 1.34x (同じ入力値)

競合するループの結果には (すべてのマージが完了した結果のベクトル、上記のシーケンスでは ZMM2)、部分積のすべてが含まれることに注意してください。つまり、各要素の結果には、同じインデックス値を持つ以前の要素がすべてマージされているものが含まれます。例 13-18 を実行した結果、ZMM2 には次の値が格納されます。

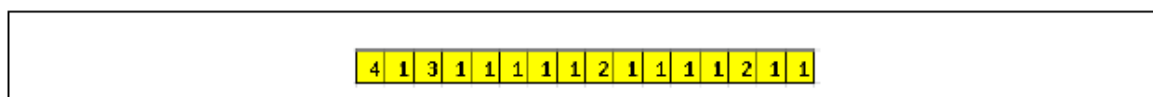


図 13-13 VPCONFICTD ZMM2 の結果

### 13.16.2 VPCONFLICT による疎ドット積

疎なベクトルが 1 組の配列として格納されることを想定します。1 つには非ゼロの値が含まれ、もう一方はこれらの値の元の場所のベクトルを含んでいます。インデックスは増分でソートされていることに注意してください。

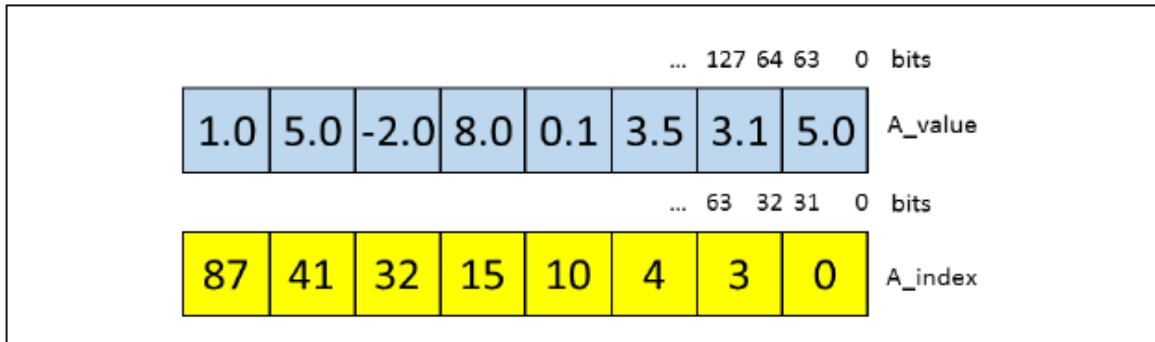


図 13-14 疎ベクトルの例

2 つの疎ベクトルのドット積を効率良く実行するには、インデックスが一致する要素を検出する必要があります。そして、それらに対し乗算と累加を行います。これを行うスカラー手法では、2 つのインデックス配列の先頭からそれらのインデックスを比較して、一致すれば乗算と累加を行い、次にインデックスを進めます。一致しなければ少ない方のインデックスを進めます。

```
A_offset = 0; B_offset = 0; sum = 0;
while ((A_offset < A_length) && (B_offset < B_length))
{
    if (A_index[A_offset] == B_index[B_offset]) // 一致
    {
        sum += A_value[A_offset] * B_value[B_offset];
        A_offset++;
        B_offset++;
    }
    else if (A_index[A_offset] < B_index[B_offset])
    {
        A_offset++;
    }
    else
    {
        B_offset++;
    }
}
```

インテル® AVX-512CD 命令は、このループを効率良くベクトル化します。各ベクトルの 1 つのインデックスを比較する代わりに、8 つを同時に比較します。最初に、各ベクトル向けの 8 つのインデックスを単一のベクトルレジスターに結合します。そして、VPCONFLICT 命令でインデックスを比較します。その出力をベクトル A の一致する要素のマスクを生成するため使用し、また B の値を対応する場所へ移動するパーミュート制御を作成します。そして、ベクトル FMA 命令を実行します。

次のアセンブリ・コードでは、スカラーとベクトルバージョンの単一比較と FMA を行っています。簡潔にするため、オフセットの更新とループは省略されています。

例 13-19 インテル® AVX-512CD を使用してスカラーとベクトルを更新

スカラーコード	インテル® AVX-512 コード
<pre> mov rdx, A_index mov rcx, A_offset mov rax, A_value mov r12, B_index mov r13, B_offset mov rbx, B_value movl r10d, [rdx+rcx*4] movl r11d, [r12+r13*4] cmpl r10d, r11d jne skip_fma // 一致したら fma を実行 movsd xmm5, [rbx+r14*8] mulsd xmm5, [rax+rcx*8] addsd xmm4, xmm5 skip_fma: </pre>	<pre> mov rdx, A_index mov rcx, A_offset mov rax, A_value mov r12, B_index mov r13, B_offset mov rbx, B_value mov r14, all_31s // {31, 31, ...} の配列 vmovaps zmm2, [r14] mov r15, upconvert_control // {0, 7, 0, 6, 0, 5,0, 4, 0, 3, 0, 2, 0, 1, 0, 0} の配列 vmovaps zmm1, [r15] vpternlogd zmm0, 255, zmm0, zmm0, zmm0 movl esi, 21845 kmovw k1, esi // 基数ビットを設定  // A のインデックスを 8 つリード vmovdqa ymm5, [rdx+rcx*4] // B のインデックスを 8 つリードして、 // zmm6 の上位へ配置 vinserti64x4 zmm6, 1, [r12+r13*4], zmm5 vpconflictz zmm7, zmm6 // A と B の比較を抽出 vextracti64x4 ymm8, 1, zmm7 // 比較結果を permute 制御へ変換 vplzcntd zmm9, zmm8 vptestmd k2, zmm8, zmm0 vpsubd zmm10, zmm2, zmm9 // データが 64 ビットであるため、パーミュート制御を // 32 ビットから 64 ビットへアップコンバート vpermd zmm11{k1}, zmm10, zmm1 // A の値を対応する B の値に移動し、 // FMA を実行 vpermpd zmm12{k2}{z}, [rax+rcx*8], zmm11 vmfadd231pd zmm4, [rbx+r13*8], zmm12 </pre>
ベースライン 1x	スピードアップ: 4.4x

### 13.17 FMA のレイテンシー

512 ビット・レジスターのポート方式で実行される場合、ポート 0 の FMA は 4 サイクルのレイテンシーを、ポート 5 の FMA は 6 サイクルのレイテンシーが科せられます。バイパスには、-2 (高速バイパス) から +1 サイクルの遅延がかかります。そのため、Skylake<sup>†</sup> マイクロアーキテクチャーの FMA ユニットで実行される命令のレイテンシーは、4-7 サイクルです。

命令は以下のグループに分類されます。

- Y グループ A の命令: vadd\*; vmfadd\*; vfnmsub\*; vfnmadd\*; vfnmsub\*; vmax\*; vmin\*; vmul\*; vscalef\*; vsub\*; vcvt\*; vgetexp\*; vfixupimm\*; vrange\*; vgetmant\*; vreduce\*; vcmp\*; vcomi\*; vdpp\*; vhadd\*; vhsb\*; vrndsclae\*; vround\*
- Y グループ B の命令: vpmaddubsw; vpmaddwd; vpmuldq; vpmulhrsw; vpmulhuw; vpmulhw; vpmullw; vpmuludq

すべての命令のソースが FMA ユニットから供給される場合、FMA ユニットは高速バイパスをサポートします。グループ A はポート 0 とポート 5 で 4 サイクルのレイテンシー、グループ B はポート 0 とポート 5 で 5 サイクルのレイテンシーを持ちます。

次の図は、すべての命令のソースが FMA ユニットから供給される高速バイパスの様子を示しています。

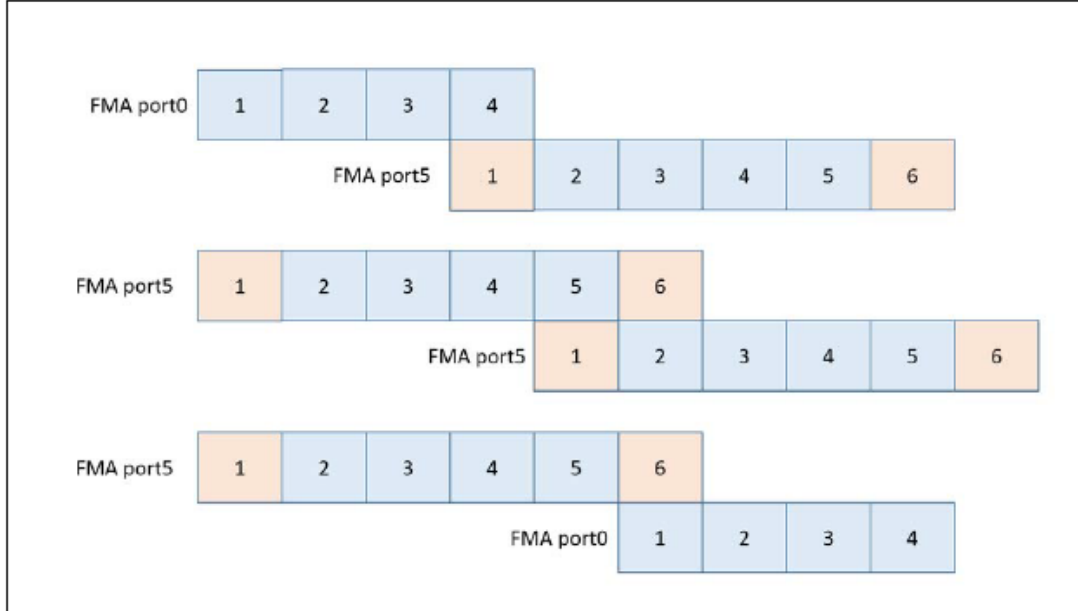


図 13-15 すべてのソースが FMA ユニットから供給される高速バイパス

青のボックスは計算サイクルを示します。肌色のボックスは、ポート 5 の FMA ユニット向けのデータ転送を表しています。

高速バイパスを利用せずに、すべてのソースが FMA ユニットから供給されない場合、グループ A の命令はポート 0 で 4 サイクル、ポート 5 で 6 サイクルのレイテンシーを持ちます。グループ B の命令には追加サイクルが科せられ、ポート 0 で 5 サイクル、ポート 5 で 7 サイクルのレイテンシーとなります。

表 13-8 に各種オプションにおける FMA ユニットのレイテンシーをまとめています。

表 13-8 FMA ユニットのレイテンシー

命令グループ	高速バイパス (FMA データ再利用)		高速バイパスなし (FMA データの再利用なし)	
	ポート 0	ポート 5	ポート 0	ポート 5
グループ A	4	4	4	6
グループ B	5	5	5	7

## 13.18 インテル® AVX 拡張またはインテル® AVX-512 拡張命令とインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) の混在

プロセッサ状態に影響する 2 つの命令グループがあります。

- Y グループ A: ベクトルレジスタ 0 から 15 のビット 128 - 511 をゼロに設定するか、それらを全く変更しない命令タイプ。
  - インテル® SSE 命令
  - 128 ビット・インテル® AVX 命令、128 ビット・インテル® AVX-512 命令
  - 256 ビット (ymm16-ymm31) インテル® AVX-512 命令
  - 512 ビット (zmm16-zmm31) インテル® AVX-512 命令
  - k0 - k7 マスクレジスタへ書き込むインテル® AVX-512 命令
  - GPR 命令
- Y グループ B: ベクトルレジスタ 0 から 15 のビット 128 - 511 を変更する命令タイプ。
  - 256 ビット (ymm0-ymm15) インテル® AVX 命令、インテル® AVX-512 命令
  - 512 ビット (zmm0-zmm15) インテル® AVX-512 命令

図 13-16 は、Skylake Server<sup>†</sup> マイクロアーキテクチャーにおける、インテル® AVX 命令またはインテル® AVX-512 命令とインテル® SSE 命令を混在させた際の遷移モデルを示しています。

実装は、Skylake<sup>†</sup> クライアントと同じですが、上位がダーティーなステート (Dirty Upper Staue) (2) でインテル® SSE 命令を実行すると、デスティネーション・レジスタの 128 - 511 ビットを保持する必要があり、命令はデスティネーション・レジスタと blend 命令のビット 128 - 511 との間に追加の依存性を持ちます。

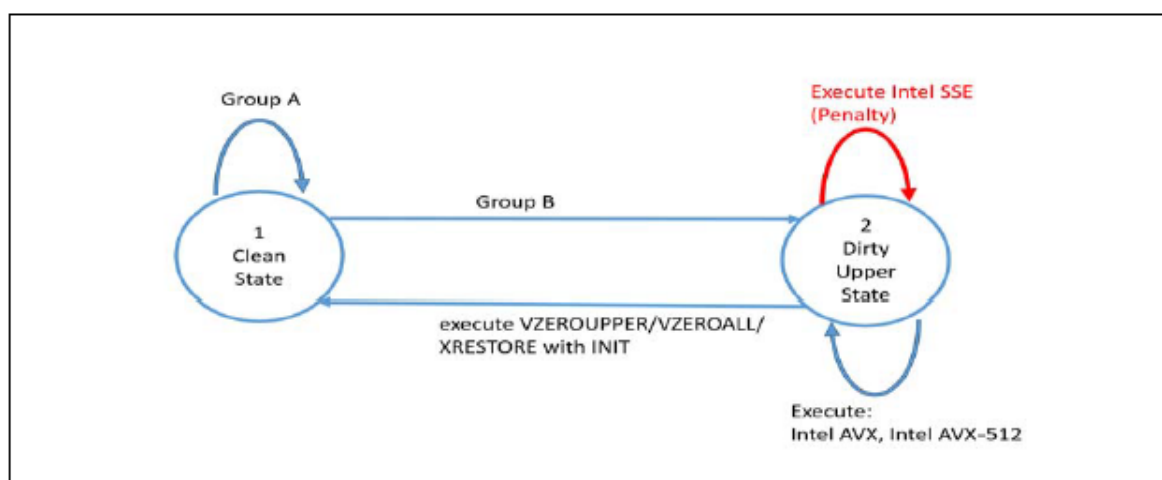


図 13-16 インテル® AVX またはインテル® AVX-512 命令をインテル® SSE 命令と混在する

推奨:

- Y グループ B の命令をインテル® SSE 命令と混在させる必要がある、またはその可能性がある場合、VZEROUPPER 命令を使用します。
- Y グループ B の命令が実行された後、さらにインテル® SSE 命令を含む関数呼び出しが行われる前に、VZEROUPPER 命令を追加します。
- Y グループ B の命令を使用する関数の終わりに VZEROUPPER 命令を追加します。
- Y スレッドが「上位ダーティーなステート」を引き継がないように、「クリーンステート」でなければスレッドを生成する前に VZEROUPPER 命令を追加します。

### 13.19 ZMM ベクトルコードと XMM/YMM コードの混在

Skylake<sup>+</sup> マイクロアーキテクチャーは、2 つのポート体系を持ちます。1 つは 256 ビット以下のレジスターを使用し、もう 1 つは 512 ビット・レジスターを使用するものです。

256 ビットのレジスターを使用する FMA 操作は、ポート 0 と 1 ヘディスパッチされ、SIMD 操作はポート 0、1、または 5 ヘディスパッチされます。512 ビット・レジスターを使用する操作は、FMA および SIMD 操作の両方がポート 0 と 5 ヘディスパッチされます。

リザベーション・ステーション (RS) の最大レジスター幅が、256 または 512 ポート体系を決定します。

インテル® AVX-512 でエンコードされた YMM 命令を使用する場合、命令は 256 ビット幅であると想定されま

す。512 ビット・ポート体系の結果は、XMM や YMM コードを 3 つのポート (0、1、5) ではなく 2 つのポート (0 と 5) ヘディスパッチするため、256 ビット・ポート体系と比較してスループットは低くそしてレイテンシーは長くなります。

例 13-20 256 ビット・コードと 256 ビットと 512 ビット・コードの混在

256 ビット・コードのみ	256 ビットと 512 ビット・コードの混在
<pre> Loop: vpbroadcastd ymm0, dword ptr [rsp] vfmadd213ps ymm7, ymm7, ymm7 vfmadd213ps ymm8, ymm8, ymm8 vfmadd213ps ymm9, ymm9, ymm9 vfmadd213ps ymm10, ymm10, ymm10 vfmadd213ps ymm11, ymm11, ymm11 vfmadd213ps ymm12, ymm12, ymm12 vfmadd213ps ymm13, ymm13, ymm13 vfmadd213ps ymm14, ymm14, ymm14 vfmadd213ps ymm15, ymm15, ymm15 vfmadd213ps ymm16, ymm16, ymm16 vfmadd213ps ymm17, ymm17, ymm17 vfmadd213ps ymm18, ymm18, ymm18 vpermd ymm1, ymm1, ymm1 vpermd ymm2, ymm2, ymm2 vpermd ymm3, ymm3, ymm3 vpermd ymm4, ymm4, ymm4 vpermd ymm5, ymm5, ymm5 vpermd ymm6, ymm6, ymm6 dec rdx jnl Loop                     </pre>	<pre> Loop: vpbroadcastd zmm0, dword ptr [rsp] vfmadd213ps ymm7, ymm7, ymm7 vfmadd213ps ymm8, ymm8, ymm8 vfmadd213ps ymm9, ymm9, ymm9 vfmadd213ps ymm10, ymm10, ymm10 vfmadd213ps ymm11, ymm11, ymm11 vfmadd213ps ymm12, ymm12, ymm12 vfmadd213ps ymm13, ymm13, ymm13 vfmadd213ps ymm14, ymm14, ymm14 vfmadd213ps ymm15, ymm15, ymm15 vfmadd213ps ymm16, ymm16, ymm16 vfmadd213ps ymm17, ymm17, ymm17 vfmadd213ps ymm18, ymm18, ymm18 vpermd ymm1, ymm1, ymm1 vpermd ymm2, ymm2, ymm2 vpermd ymm3, ymm3, ymm3 vpermd ymm4, ymm4, ymm4 vpermd ymm5, ymm5, ymm5 vpermd ymm6, ymm6, ymm6 dec rdx jnl Loop                     </pre>
ベースライン 1x	スローダウン: 1.3x

256 ビットのみコード例では、FMA はポート 0 と 1 ヘディスパッチされ、*permd* は broadcast 命令が 256 ビットであることから、ポート 5 ヘディスパッチされます。256 ビットと 512 ビットが混在するコード例では、broadcast は 512 ビット幅であるため、プロセッサは 512 ビット・ポート体系を使用して FMA をポート 0 と 5 へ、*permd* をポート 5 ヘディスパッチします。そのためポート 5 へのプレッシャーが高まります。

## 13.20 単一の FMA ユニットの備える場合

Skylake<sup>+</sup> マイクロアーキテクチャー・ベースのプロセッサには、2 つのインテル® AVX-512 FMA ユニット (ポート 0 と 5) を備えるものがある一方、同じアーキテクチャー・ベースのプロセッサでも単一のインテル® AVX-512 FMA ユニット (ポート 0) を持つものがあります。

2 つの FMA ユニットで実行されることを想定して最適化されたコードは、単一 FMA ユニット上での実行には最適でない可能性があります。

次のコードは、システムのインテル® AVX-512 FMA ユニットの数を判定する方法を示します。以下を含みます。

- Y インテル® AVX-512 のウォームアップ
- Y FMA 命令のみを実行する関数
- Y FMA 命令と shuffle 命令を実行する関数
- Y コードは、2 つのテスト結果を基にプロセッサが 1 つまたは 2 つの FMA ユニットの搭載するかを特定します。

テストの精度を高めるため 3 回実行されることに注意してください。

プログラムのオーバーヘッドを軽減するため、すべての関数呼び出しでこのテストを実行しないことを強く推奨します。インストールの一環として、または起動時に一度実行してください。

2 つのプロセッサの区別は、2 つのスループット・テスト間の比率に基づいています。2 つの FMA ユニットの備えるプロセッサは、FMA のみのテストを FMA と shuffle テストの 2 倍高速に実行できます。しかし、単独の FMA ユニットの持つプロセッサは、2 つのテストを同じスピードで実行します。

### 例 13-21 Skylake<sup>+</sup> マイクロアーキテクチャー・ベースのプロセッサの FMA ユニットの数を特定 (1 つもしくは 2 つ)

```
#include <string.h>
#include <stdlib.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdint.h>

static uint64_t rdtsc(void) {
    unsigned int ax, dx;
    __asm__ __volatile__ ("rdtsc" : "=a"(ax), "=d"(dx));
    return (((uint64_t)dx) << 32) | ax;
}

uint64_t fma_shuffle_tpt(uint64_t loop_cnt){
    uint64_t loops = loop_cnt;
    __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1, 1, 1, 1, 1};
    __declspec(align(64)) int shuf_vec[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
    __asm
    {
        vmovups zmm0, [one_vec]
        vmovups zmm1, [one_vec]
        vmovups zmm2, [one_vec]
        vmovups zmm3, [one_vec]
        vmovups zmm4, [one_vec]
        vmovups zmm5, [one_vec]
```

```

vmovups zmm6, [one_vec]
vmovups zmm7, [one_vec]
vmovups zmm8, [one_vec]
vmovups zmm9, [one_vec]
vmovups zmm10, [one_vec]
vmovups zmm11, [one_vec]
vmovups zmm12, [shuf_vec]
vmovups zmm13, [shuf_vec]
vmovups zmm14, [shuf_vec]
vmovups zmm15, [shuf_vec]
vmovups zmm16, [shuf_vec]
vmovups zmm17, [shuf_vec]
vmovups zmm18, [shuf_vec]
vmovups zmm19, [shuf_vec]
vmovups zmm20, [shuf_vec]
vmovups zmm21, [shuf_vec]
vmovups zmm22, [shuf_vec]
vmovups zmm23, [shuf_vec]
vmovups zmm30, [shuf_vec]
mov rdx, loops
loop1:
vfmadd231pd zmm0, zmm0, zmm0
vfmadd231pd zmm1, zmm1, zmm1
vfmadd231pd zmm2, zmm2, zmm2
vfmadd231pd zmm3, zmm3, zmm3
vfmadd231pd zmm4, zmm4, zmm4
vfmadd231pd zmm5, zmm5, zmm5
vfmadd231pd zmm6, zmm6, zmm6
vfmadd231pd zmm7, zmm7, zmm7
vfmadd231pd zmm8, zmm8, zmm8
vfmadd231pd zmm9, zmm9, zmm9
vfmadd231pd zmm10, zmm10, zmm10
vfmadd231pd zmm11, zmm11, zmm11
vpermd zmm12, zmm30, zmm30
vpermd zmm13, zmm30, zmm30
vpermd zmm14, zmm30, zmm30
vpermd zmm15, zmm30, zmm30
vpermd zmm16, zmm30, zmm30
vpermd zmm17, zmm30, zmm30
vpermd zmm18, zmm30, zmm30
vpermd zmm19, zmm30, zmm30
vpermd zmm20, zmm30, zmm30
vpermd zmm21, zmm30, zmm30
vpermd zmm22, zmm30, zmm30
vpermd zmm23, zmm30, zmm30
dec rdx
jg loop1
}
}

uint64_t fma_only_tpt(int loop_cnt){
uint64_t loops = loop_cnt;
__declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
__asm

```



```

{
vmovups zmm0, [one_vec]
vmovups zmm1, [one_vec]
vmovups zmm2, [one_vec]
vmovups zmm3, [one_vec]
vmovups zmm4, [one_vec]
vmovups zmm5, [one_vec]
vmovups zmm6, [one_vec]
vmovups zmm7, [one_vec]
vmovups zmm8, [one_vec]
vmovups zmm9, [one_vec]
vmovups zmm10, [one_vec]
vmovups zmm11, [one_vec]
mov rdx, loops
loop1:
vfmadd231pd zmm0, zmm0, zmm0
vfmadd231pd zmm1, zmm1, zmm1
vfmadd231pd zmm2, zmm2, zmm2
vfmadd231pd zmm3, zmm3, zmm3
vfmadd231pd zmm4, zmm4, zmm4
vfmadd231pd zmm5, zmm5, zmm5
vfmadd231pd zmm6, zmm6, zmm6
vfmadd231pd zmm7, zmm7, zmm7
vfmadd231pd zmm8, zmm8, zmm8
vfmadd231pd zmm9, zmm9, zmm9
vfmadd231pd zmm10, zmm10, zmm10
vfmadd231pd zmm11, zmm11, zmm11
dec rdx
jg loop1
}
}

```

```

int main()
{
int i;
uint64_t fma_shuf_tpt_test[3];
uint64_t fma_shuf_tpt_test_min;
uint64_t fma_only_tpt_test[3];
uint64_t fma_only_tpt_test_min;
uint64_t start = 0;
uint64_t number_of_fma_units_per_core = 2;
/*****
/* ステップ 1: ウォームアップ */
*****/
fma_only_tpt(100000);

/*****
/* ステップ 2: FMA と Shuffle TPT テストを実行 */
*****/
for(i = 0; i < 3; i++){
start = rdtsc();
fma_shuffle_tpt(1000);
fma_shuf_tpt_test[i] = rdtsc() - start;
}
}

```

```

/*****
/* ステップ 3: FMA のみの TPT テストを実行 */
/*****
for(i = 0; i < 3; i++){
    start = rdtsc();
    fma_only_tpt(1000);
    fma_only_tpt_test[i] = rdtsc() - start;
}

/*****
/* ステップ 4: 1 つの FMA か 2 つの FMA か決定 */
/*****
fma_shuf_tpt_test_min = fma_shuf_tpt_test[0];
fma_only_tpt_test_min = fma_only_tpt_test[0];
for(i = 1; i < 3; i++){
    if ((int)fma_shuf_tpt_test[i] < (int)fma_shuf_tpt_test_min) fma_shuf_tpt_test_min =
        fma_shuf_tpt_test[i];
    if ((int)fma_only_tpt_test[i] < (int)fma_only_tpt_test_min) fma_only_tpt_test_min =
        fma_only_tpt_test[i];
}
if(((double)fma_shuf_tpt_test_min/(double)fma_only_tpt_test_min) > 1.5){
    number_of_fma_units_per_core = 1;
}
printf("%d FMA server¥n", number_of_fma_units_per_core);
return 0;
}

```

## 13.21 シャッフルのためのギャザー/スカッター (G2S/STS)

### 13.21.1 スライドロードでシャッフルするためのギャザー

メモリー上の集約された要素間にデータの局所性がある場合、gather 命令をソフトウェア・シーケンスに置き換えることでパフォーマンスを向上できます。

この節では、最も一般的なスライドロードのパターンについて考えます。スライドロードとは、連続するロードがアクセスするメモリー上のオフセットが一定間隔のロードを指します。

次の例は、3 つの異なるコードで構造体配列 (AoS) から配列構造体 (SoA) への変換を行う様子を示しています。コードは、複素数配列内の実数と虚数要素を 2 つの異なる配列へ分離します。

次のような C コードについて考えてみます。

```

for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[i].real;
    Imaginary_buffer[i] = Complex_buffer[i].imag;
}

```

例 13-22 スライドロードでシャッフルするためのギャザーの例

代替 1: インテル® AVX-512 vpgatherdd	代替 2: インテル® AVX-512 vpermi2d を使用する G2S
<pre> loop: vpcmpeqb k1, xmm0, xmm0 vpcmpeqb k2, xmm0, xmm0 movsxd rdx, edx movsxd rdi, esi inc esi shl rdi, 0x7 vpxord zmm2, zmm2, zmm2 lea rax, ptr [r8+rdx*8] add edx, 0x20 vpgatherdd zmm2, k1, [rax+zmm1*4] vpxord zmm3, zmm3, zmm3 vpxord zmm4, zmm4, zmm4 vpxord zmm5, zmm5, zmm5 vpgatherdd zmm3, k2, [rax+zmm0*4] vpcmpeqb k3, xmm0, xmm0 vpcmpeqb k4, xmm0, xmm0 vmovups [r9+rdi*1], zmm2 vmovups [rcx+rdi*1], zmm3 vpgatherdd zmm4, k3, [rax+zmm1*4+0x80] vpgatherdd zmm5, k4, [rax+zmm0*4+0x80] vmovups [r9+rdi*1+0x40], zmm4 vmovups [rcx+rdi*1+0x40], zmm5 cmp esi, r14d jb loop                     </pre>	<pre> vmovups zmm4, [rdx+r9*8] vmovups zmm0, [rdx+r9*8+0x40] vmovups zmm5, [rdx+r9*8+0x80] vmovups zmm1, [rdx+r9*8+0xc0] vmovaps zmm2, zmm7 vmovaps zmm3, zmm7 vpermi2d zmm2, zmm4, zmm0 vpermt2d zmm4, zmm6, zmm0 vpermi2d zmm3, zmm5, zmm1 vpermt2d zmm5, zmm6, zmm1 vmovdqu32 [rcx+r9*4], zmm2 vmovdqu32 [rcx+r9*4+0x40], zmm3 vmovdqu32 [r8+r9*4], zmm4 vmovdqu32 [r8+r9*4+0x40], zmm5 add r9, 0x20 cmp r9, r10 jb loop                     </pre>
ベースライン 1x	スピードアップ: 4.8x

次の定数は zmm レジスターへロードされ、gather と permute のインデックスとして使用されます。

Zmm0 (代替 1)、zmm6 (代替 2)

```
__declspec(align(64)) const __int32 gather_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};
```

Zmm1 (代替 1)、zmm7 (代替 2)

```
__declspec(align(64)) const __int32 gather_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};
```

**推奨事項:** 最高のパフォーマンスを得るには、短い間隔のスライドロードをロードとパーミュートのシーケンスに置き換えます。

### 13.21.2 スライドストアでシャッフルするためのスキッター

以下は、シャッフルを行うスキッターの例であり、スキッターをパーミュートとストア命令に置き換える C のコードです。

```
for(int i=0;i<len;i++){
    Complex_buffer[i].real = Real_buffer[i];
    Complex_buffer[i].imag = Imaginary_buffer[i];
}
```

例 13-23 スライドストアでシャッフルするためのスカッターの例

代替 1: インテル® AVX-512 vscatterdps	代替 2: インテル® AVX-512 vpermi2d を使用する S2S
<pre>loop: vpcmpeqb k1, xmm0, xmm0 lea r11, ptr [r8+rcx*4] vpcmpeqb k2, xmm0, xmm0 vmovups zmm2, [rax+rsi*4] vmovups zmm3, [r9+rsi*4] vscatterdps [r11+zmm1*4], k1, zmm2 vscatterdps [r11+zmm0*4], k2, zmm3 add rsi, 0x10 add rcx, 0x20 cmp rsi, r10 jl loop</pre>	<pre>loop: vmovups zmm4, [rax+r8*4] vmovups zmm2, [r10+r8*4] vmovaps zmm3, zmm1 add r8, 0x10 vpermi2d zmm3, zmm4, zmm2 vpermt2d zmm4, zmm0, zmm2 vmovups [r9+rsi*4], zmm3 vmovups [r9+rsi*4+0x40], zmm4 add rsi, 0x20 cmp r8, r11 jl loop</pre>
ベースライン 1x	スピードアップ: 4.4x

次の定数がスカッターのインデックスとして使用されています。

Zmm1:

```
__declspec (align(64)) const __int32 scatter_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};
```

Zmm0:

```
__declspec (align(64)) const __int32 scatter_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};
```

次の定数がパーミュートのインデックスとして使用されています。

Zmm1:

```
__declspec (align(64)) const __int32 first_half[16] = {0, 16, 1, 17, 2, 18, 3, 19, 4, 20, 5, 21, 6, 22, 7, 23};
```

Zmm0:

```
__declspec (align(64)) const __int32 second_half[16] = {8, 24, 9, 25, 10, 26, 11, 27, 12, 28, 13, 29, 14, 30, 15, 31};
```

### 13.21.3 隣接するロードでシャッフルするためのギャザー

集約される要素が隣接するシーケンスにグループ化される場合、gather 命令をソフトウェア・シーケンスで置き換えることでパフォーマンスを向上できます。

次の例は、要素が隣接する場合どのようにベクトルをロードするかを示しています。

この場合、配列要素の順番はインデックス・バッファーに従って設定されることに注意してください。そのため、[13.21.1 節「スライドロードでシャッフルするためのギャザー」](#)で説明したソフトウェアの最適化は、このケースには適用されません。

次のような C コードについて考えてみます。

```
typedef struct{
    double var[4];
} ElemStruct;
```

```
const int* indices = Indices;
const ElemStruct *in = (const ElemStruct*) InputBuffer;
double* restrict out = OutputBuffer;

for (int i = 0; i < width; i++){
    for (int j = 0; j < 4; j++){
        out[i*4+j] = in[indices[i]].var[j];
    }
}
```

例 13-24 隣接するロードでシャッフルするためのギャザーの例

代替 1: vgatherdpd 実装	代替 2: ロードとマスク付きブロードキャスト
<pre>loop: vpbroadcastd ymm3, [r9+rsi*4] mov r15d, esi vpbroadcastd xmm2, [r9+rsi*4+0x4] add rsi, 0x2 vpbroadcastq ymm3{k1}, xmm2 vpmulld ymm4, ymm3, ymm1 vpadd ymm5, ymm4, ymm0 vpcmpeqb k2, xmm0, xmm0 shl r15d, 0x2 movsxd r15, r15d vpxord zmm6, zmm6, zmm6 vgatherdpd zmm6, k2, [r10+ymm5*1] vmovups [r11+r15*8], zmm6 cmp rsi, rdi jl loop</pre>	<pre>loop: movsxd r11, [r10+rcx*4] shl r11, 0x5 vmovupd ymm0, [r9+r11*1] movsxd r11, [r10+rcx*4+0x4] shl r11, 0x5 vbroadcastf64x4 zmm0{k1}, [r9+r11*1] mov r11d, ecx shl r11d, 0x2 add rcx, 0x2 movsxd r11, r11d vmovups [r8+r11*8], zmm0 cmp rcx, rsi jl loop</pre>
ベースライン 1x	スピードアップ: 2.2x

次の定数が vgatherdpd 実装に使用されています。

```
ymm0:
__declspec (align(64)) const __int32 index_inc[8] = {0, 8, 16, 24, 0, 8, 16, 24};
```

```
ymm1:
__declspec (align(64)) const __int32 index_scale[8] = {32, 32, 32, 32, 32, 32, 32, 32};
```

K1 レジスターの値は 0xF0 です。

## 13.22 データ・アライメント

この節では、インテル® AVX-512 命令におけるアライメントされたデータの利点を説明し、アライメントができない場合にパフォーマンスを改善するいくつかの方法を紹介します。この節では、いくつかの SAXPY カーネル例を使用して説明を行います。SAXPY はスカラーの「Alpha \* X + Y」アルゴリズムです。

以下の C コードは SAXPY の C 実装です。

```
for (int i = 0; i < n; i++)
{
    c[i] = alpha * a[i] + b[i];
}
```

## 13.22.1 64 バイトにデータをアライメント

ベクトル長に合わせてデータをアライメントすることを推奨します。最良の結果を得るには、インテル® AVX-512 命令を使用する場合はデータを 64 バイトにアライメントします。

インテル® AVX-512 でアライメントなしの 64 バイト・ロード/ストアを実行する場合、キャッシュラインが 64 バイトであるため、各ロード/ストアはキャッシュライン分割を引き起こします。32 バイト・レジスターを使用するインテル® AVX2 と比べて、キャッシュライン分割の頻度は倍になります。メモリー集約型のコードでキャッシュライン分割が頻発すると、パフォーマンスは低下します。

次の表は、メモリー集約型の SAXPY コードのパフォーマンスが、アライメントされていない入出力バッファーによってどのように影響されるかを示しています。表のデータは次のコードを基に算出されています。

```
__asm {
    mov rax, src1
    mov rbx, src2
    mov rcx, dst
    mov rdx, len
    xor rdi, rdi
    vbroadcastss zmm0, alpha

mainloop:
    vmovups zmm1, [rax]
    vfmadd213ps zmm1, zmm0, [rbx]
    vmovups [rcx], zmm1

    vmovups zmm1, [rax+0x40]
    vfmadd213ps zmm1, zmm0, [rbx+0x40]
    vmovups [rcx+0x40], zmm1

    vmovups zmm1, [rax+0x80]
    vfmadd213ps zmm1, zmm0, [rbx+0x80]
    vmovups [rcx+0x80], zmm1

    vmovups zmm1, [rax+0xC0]
    vfmadd213ps zmm1, zmm0, [rbx+0xC0]
    vmovups [rcx+0xC0], zmm1

    add rax, 256
    add rbx, 256
    add rcx, 256
    add rdi, 64
    cmp rdi, rdx
    jl mainloop
}
```

次の表は、各種オプションによる SAXPY のパフォーマンスへのデータ・アライメントの影響をまとめたものです。

表 13-9 SAXPY パフォーマンスとスピードアップへのデータ・アライメントの影響

SAXPY パフォーマンスへのデータ・アライメントの影響	スピードアップ
代替 1: ソースとデスティネーションはともに 64 バイトにアライメントされています。	ベースライン、1.0。
代替 2: ソースは 64 バイトにアライメントされ、デスティネーションは 4 バイト・オフセットを持ちます。	0.66x。
代替 3: ソースとデスティネーションは、アライメント境界からの 4 バイト・オフセットを持ちます。	0.59x。
代替 4: 一方のソースはアライメント境界からの 4 バイト・オフセットを持ち、もう一方のソースとデスティネーションは 64 バイトにアライメントされています。	0.77x。

### 13.23 動的メモリー割り当てとメモリーのアライメント

次の構造体について考えてみましょう。

```
float3_SOA {
    __declspec(align(64)) float x[16];
    __declspec(align(64)) float y[16];
};
```

構造体のメモリー割り当ては 64 バイトにアライメントされています。次のように使用します。

```
float3_SOA f;
```

次のような動的なメモリー割り当てを行う場合、`declspec` ディレクティブは無視され、64 バイトのアライメントは保証されません。

```
float3_SOA* stPtr = new float3_SOA();
```

この場合、動的なアライメントを保証する割り当てを使用するか、オペレーター `new` を再定義する必要があります。

**推奨事項:** 可能な限りデータを 64 バイトにアライメントし、次のガイドラインに従います。

Y インテル® コンパイラーで提供される `_mm_malloc` 組込み関数や Microsoft\* コンパイラーの `_aligned_malloc` を使用して動的なデータ割り当てをアライメントできます。次に例を示します。  
 // float 2048 要素のバッファを 64 バイトのアライメントで動的に割り当て。  
 InputBuffer = (float\*) \_mm\_malloc (2048\*sizeof(float), 64);

Y `__declspec(align(64))` を使用して静的なデータ割り当てをアライメントできます。次に例を示します。  
 // float 2048 要素のバッファを 64 バイトのアライメントで静的に割り当て。  
 \_\_declspec(align(64)) float InputBuffer[2048];

## 13.24 除算と平方根命令

VRSQRT14PS/VRSQRT14PD および VRCP14PS/VRCP14PD 命令を使用して、単精度除算と平方根の計算をスピードアップできます。これらの命令は、入力値の逆数平方根/逆数除算の近似 (14 ビット精度で) を生成します。

インテル® AVX-512 ではこれらの命令はパイプライン化され、次のような機能を持ちます。

- Y 256 ビットのベクトルの場合、4 サイクルのレイテンシーとサイクルごとに 1 命令のスループット。
- Y 512 ビットのベクトルの場合、6 サイクルのレイテンシーと 2 サイクルごとに 1 命令のスループット。

Skylake<sup>+</sup> マイクロアーキテクチャーでは、逆数平方根/逆数除算のパックド倍精度 (PD) VRSQRT14PD と VRCP14PD が導入されました。

ニュートンラフソン反復やほかの多項式近似で VRSQRT14PS/VRSQRT14PD および VRCP14PS/VRCP14PD 命令を使用することで、VDIVPS や VSQRTPS 命令 (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』を参照) と同じ精度を実現し、より高いスループットを達成できます。

完全な精度 (IEEE) が必要である場合、Skylake<sup>+</sup> マイクロアーキテクチャーの劇的なパフォーマンス向上により低レイテンシーと高スループットが達成された DIVPS と SQRTPS を使用します。表 13-12 に、Broadwell<sup>+</sup> と Skylake<sup>+</sup> マイクロアーキテクチャーにおけるこれらの命令のパフォーマンスの比較を示します。

### 注意

除算や平方根操作が大きなアルゴリズムの一部であるためレイテンシーが隠匿されるような場合、ニュートンラフソン近似は、そのほかの命令から発行されるマイクロオペレーションでパイプが満たされることで実行速度が低下する可能性があります。

次の節では、必要とする精度レベルに合わせた推奨される計算方法と命令を示します。

### 注意

値の近似誤りとその近似には 2 つの定義があります。

approx

$$\begin{aligned} \text{絶対誤差} &= |v - \text{approx}| \\ \text{相対誤差} &= |v - \text{approx}| / |v| \end{aligned}$$

本章では、“ビット数” エラーは相対的であり、絶対値のエラーではないとします。ここで比較する近似値  $v$  は、できるだけ正確で、倍精度でなければいけません。



### 13.24.1 除算と平方根命令の近似

表 13-10 Skylake<sup>+</sup> マイクロアーキテクチャーで推奨される DIV/SQRT ベースの操作 (単精度)

演算	精度	推奨される方法
除算	24 ビット (IEEE)	DIVPS
	23 ビット	RCP14PS + MULPS + 1 ニュートンラフソン反復
	14 ビット	RCP14PS + MULPS
逆数平方根	22 ビット	SQRTPS + DIVPS
	23 ビット	RSQRT14PS + 1 ニュートンラフソン反復
	14 ビット	RSQRT14PS
平方根	24 ビット (IEEE)	SQRTPS
	23 ビット	RSQRT14PS + MULPS + 1 ニュートンラフソン反復
	14 ビット	RSQRT14PS + MULPS

表 13-11 Skylake<sup>+</sup> マイクロアーキテクチャーで推奨される DIV/SQRT ベースの操作 (倍精度)

演算	精度	推奨される方法
除算	53 ビット (IEEE)	DIVPD
	52 ビット	RCP14PD + MULPD + 2 ニュートンラフソン反復
	26 ビット	RCP14PD + MULPD + 1 ニュートンラフソン反復
	14 ビット	RCP14PD + MULPD
逆数平方根	53 ビット (IEEE)	SQRTPD + DIVPD
	52 ビット	RSQRT14PD+2 N-R + エラー訂正または SQRTPD + DIVPD
	50 ビット	RSQRT14PD + 多項式近似
	26 ビット	RSQRT14PD+1 N-R
	14 ビット	RSQRT14PD
平方根	53 ビット (IEEE)	SQRTPD
	52 ビット	RSQRT14PD + MULPD + 多項式近似
	26 ビット	RSQRT14PD + MULPD + 1 N-R
	14 ビット	RSQRT14PD + MULPD

### 13.24.2 除算と平方根命令のパフォーマンス

Broadwell<sup>+</sup> と Skylake<sup>+</sup> マイクロアーキテクチャーにおけるベクトル除算と平方根操作のパフォーマンスを以下に示します。

表 13-12 インテル® AVX2 の 256 ビット除算と平方根命令のパフォーマンス

Broadwell <sup>+</sup> マイクロアーキテクチャー	DIVPS	SQRTPS	DIVPD	SQRTPD
レイテンシー	17	21	23	35
スループット	10	14	16	28
Skylake <sup>+</sup> マイクロアーキテクチャー	DIVPS	SQRTPS	DIVPD	SQRTPD
レイテンシー	11	12	14	18
スループット	5	6	16	12

表 13-13 インテル® AVX-512 の 512 ビット除算と平方根命令のパフォーマンス

Skylake <sup>+</sup> マイクロアーキテクチャー	DIVPS	SQRTPS	DIVPD	SQRTPD
レイテンシー	17	19	23	31
スルーブット	10	12	16	24

### 13.24.3 近似のレイテンシー

この節では、近似法、DIV、および SQRT 命令におけるレイテンシーとスルーブットについて説明します。以下に示す表から、ほとんどのケースにおいて近似法のスルーブット・ゲインは IEEE に対して (少なくとも) 倍以上であることが分かります。ここでは、簡単なループで除算または平方根を計算しています。

ループ反復に多くの計算 (除算と平方根以外) が含まれている場合、近似シーケンスのスルーブットの利点は減少します。

経験則から、ループ反復に 8-10 を超えない単精度操作、または 12-15 を超えない倍精度操作が含まれる場合、IEEE に近い精度の近似が推奨されます。次の表から、これらの正確な近似は、スルーブットの最適化のみに効果があることが分かります。精度を必要としない近似は、レイテンシーとスルーブットの両方を向上します。

ニュートンラフソン近似は、デノーマル入力、ゼロ、および無限大の特殊ケースに適用されないことに注意してください。また、デノーマルに近い入力では中間処理のアンダーフローにより、精度を失うこともあります。ゼロと無限大入力は、わずかな操作で比較的容易に解決できますが (以下のシーケンスで示すように)、デノーマル除数はパフォーマンスへの影響なしに対処することはできません。アンダーフローやオーバーフローのしきい値から離れた、「中央範囲」の入力に対し近似シーケンスは最も効率良く動作します。

次の表は、インテル® AVX-512 の除算と平方根命令の単精度のレイテンシーとスルーブットを、Skylake<sup>+</sup> マイクロアーキテクチャーの近似法と比較して示しています。

表 13-14 Skylake<sup>+</sup> マイクロアーキテクチャーにおける異なるベクトル幅で除算と平方根を計算する異なる手法のレイテンシー/スルーブット (単精度)

演算	方法	精度	256 ビット・インテル® AVX-512		512 ビット・インテル® AVX-512	
			組込み関数		組込み関数	
			スルーブット	レイテンシー	スルーブット	レイテンシー
除算 (a/b)	DIVPS	24 ビット (IEEE)	5	11	10	17
	RCP14PS + MULPS + 1 ニュートンラフソン反復	23 ビット	2	16	3	20
	RCP14PS + MULPS	14 ビット	1	8	2	10-12
平方根	SQRTPS	24 ビット (IEEE)	6	12	12	19
	RSQRT14PS + MULPS + 1 ニュートンラフソン反復	23 ビット	3	16	5	20
	RSQRT14PS + MULPS	14 ビット	2	9	3	12
逆数平方根	SQRTPS + DIVPS	22 ビット	11	23	22	36
	RSQRT14PS + 1 ニュートンラフソン反復	23 ビット	3.67	20	4.89	25
	RSQRT14PS	14 ビット			2	6

表 13-15 Skylake<sup>†</sup> マイクロアーキテクチャーにおける異なるベクトル幅で除算と平方根を計算する異なる手法のレイテンシー/スループット (倍精度)

演算	方法	精度	256 ビット・インテル® AVX-512 組込み関数		512 ビット・インテル® AVX-512 組込み関数	
			スループット	レイテンシー	スループット	レイテンシー
除算 (a/b)	DIVPD	53 ビット (IEEE)	8	14	16	23
	RCP14PD + MULPD + 2 ニュートンラフソン反復	22 ビット	3.2	27	4.7	28.4
	RCP14PD + MULPD + 1 ニュートンラフソン反復	26 ビット	2	16	3	20
	RCP14PS + MULPS	14 ビット	1	8	2	10-12
平方根	SQRTPD	53 ビット (IEEE)	6	12	12	19
	RSQRT14PD + MULPD + 多項式近似	22 ビット	4.82	24.54 <sup>1</sup>	6.4	28.48 <sup>1</sup>
	RSQRT14PD + MULPD + 1 N-R	23 ビット	3.76	17	5	20
	RSQRT14PD + MULPD	14 ビット	2	9	3	12
逆数平方根	SQRTPD + DIVPD	51 ビット	20	32	40	53
	RSQRT14PD + 2-NR + エラー訂正	52 ビット	5	29.38	6.53	34
	RSQRT14PD+2 N-R	50 ビット	3.79	25.73	5.51	30
	RSQRT14PD+1 N-R	26 ビット	2.7	18	4.5	21.67
	RSQRT14PD	14 ビット	1	4	2	6

注意:

1. コードシーケンスにはいくつかの FMA 命令 (4/6 の異なるレイテンシーを持つ) が含まれるため、丸めは行われていません。そのため、これらのシーケンスにおけるレイテンシーを考慮する必要はありません。

### 13.24.4 コード例

#### 13.24.4.1 単精度除算 24 ビット (IEEE)

```

float a = 10;
float b = 5;
__asm {
    vbroadcastss zmm0, a           // fill zmm0 with 16 elements of a
    vbroadcastss zmm1, b           // fill zmm1 with 16 elements of b
    vdivps zmm2, zmm0, zmm1       // zmm2 = 16 elements of a/b
}
    
```

### 13.24.4.2 単精度除算 23 ビット

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of b's
   Output:
   zmm3 = vector of a/b
*/
__asm {
    vrcp14ps zmm2, zmm1
    vmulps zmm3, zmm0, zmm2
    vmovaps zmm4, zmm0
    vfmadd231ps zmm4, zmm3, zmm1
    vfmadd231ps zmm3, zmm4, zmm2
}

```

### 13.24.4.3 単精度除算 14 ビット

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of b's
   Output:
   zmm2 = vector of a/b
*/
__asm {
    vrcp14ps zmm2, zmm1
    vmulps zmm2, k0, zmm0, zmm2
}

```

### 13.24.4.4 単精度逆数平方根 22 ビット

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of 1's
   Output:
   zmm2 = vector of 1/sqrt(a)
*/
double one = 1.0;
__asm {
    vbroadcastss zmm1, one // zmm1 = vector of 16 1's
    vsqrttps zmm2, zmm0
    vdivps zmm2, zmm1, zmm2
}

```

### 13.24.4.5 単精度逆数平方根 23 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm2 = vector of 1/sqrt(a)
*/
float half = 0.5;
__asm {
    vbroadcastss zmm1, half      // zmm1 = vector of 16 0.5's
    vrsqrt14ps zmm2, zmm0
    vmulps zmm3, zmm0, zmm2
    vmulps zmm4, zmm1, zmm2
    vfmadd231ps zmm1, zmm3, zmm4
    vfmsub231ps zmm3, zmm0, zmm2
    vfmadd231ps zmm1, zmm4, zmm3
    vfmadd231ps zmm2, zmm2, zmm1
}

```

### 13.24.4.6 単精度逆数平方根 14 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm2 = vector of 1/sqrt(a)
*/
__asm {
    vrsqrt14ps zmm2, zmm0
}

```

### 13.24.4.7 単精度平方根 24 ビット (IEEE)

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm2 = vector of sqrt(a)
*/
__asm {
    vsqrtps zmm2, zmm0
}

```

### 13.24.4.8 単精度平方根 23 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm0 = vector of sqrt(a)
*/
float half = 0.5;

__asm {
    vbroadcastss zmm3, half
    vrsqrt14ps zmm1, zmm0
    vfpclassps k2, zmm0, 0ch
    vmulps zmm2, zmm0, zmm1, {rn-sae}
    vmulps zmm1, zmm1, zmm3
    knotw k3, k2
    vfmadd231ps zmm0 {k3}, zmm2, zmm2
    vfmadd213ps zmm0 {k3}, zmm1, zmm2
}

```

### 13.24.4.9 単精度平方根 14 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm0 = vector of sqrt(a)
*/

__asm {
    vrsqrt14ps zmm1, zmm0
    vfpclassps k2, zmm0, 0eh
    knotw k3, k2
    vmulps zmm0 {k3}, zmm0, zmm1
}

```

### 13.24.4.10 倍精度除算 53 ビット (IEEE)

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of b's
   Output:
   zmm2 = vector of a/b
*/

__asm {
    vdivpd zmm2, zmm0, zmm1
}

```

### 13.24.4.11 倍精度除算 52 ビット

/\* 入力:

zmm15 = a のベクトル

zmm0 = b のベクトル

出力:

zmm0 = a/b のベクトル

```

*/

double One = 1.0;
__asm {
    vrcp14pd zmm1, zmm0
    vmovapd zmm4, zmm0
    vbroadcastsd zmm2, One
    vfmadd213pd zmm0, zmm1, zmm2, {rn-sae}
    vfpclasspd k2, zmm1, 1eh
    vfmadd213pd zmm0, zmm1, zmm1, {rn-sae}
    knotw k3, k2
    vfmadd213pd zmm4, zmm0, zmm2, {rn-sae}
    vblendmpd zmm0 {k2}, zmm0, zmm1
    vfmadd213pd zmm0 {k3}, zmm4, zmm0, {rn-sae}
    vmulpd zmm0, zmm0, zmm15
}

```

### 13.24.4.12 倍精度除算 26 ビット

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of b's
   Output:
   zmm3 = vector of a/b
*/
__asm {
    vrcp14pd zmm2, zmm1
    vmulpd zmm3, zmm0, zmm2
    vmovapd zmm4, zmm0
    vfmadd231pd zmm4, zmm3, zmm1
    vfmadd231pd zmm3, zmm4, zmm2
}

```

### 13.24.4.13 倍精度除算 14 ビット

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of b's
   Output:
   zmm2 = vector of a/b
*/
__asm {
    vrcp14pd zmm2, zmm1
    vmulpd zmm2, zmm0, zmm2
}

```

### 13.24.4.14 倍精度逆数平方根 51 ビット

```

/* Input:
   zmm0 = vector of a's
   zmm1 = vector of 1's
   Output:
   zmm0 = vector of 1/sqrt(a)
*/
__asm {
    vsqrtpd zmm0, zmm0
    vdivpd zmm0, zmm1, zmm0
}

```

### 13.24.4.15 倍精度逆数平方根 52 ビット

```

/* Input:
   zmm4 = vector of a's
   Output:
   zmm0 = vector of 1/sqrt(a)
*/
// duplicates x eight times
#define DUPE_DECL(x) x,x,x,x,x,x,x,x
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 One[] = { DUPE_DECL(0x3FF0000000000011) };
ALIGNTO(64) __int64 dc1[] = { DUPE_DECL(0x3FE0000000000011) };
ALIGNTO(64) __int64 dc2[] = { DUPE_DECL(0x3FD8000004600011) };
ALIGNTO(64) __int64 dc3[] = { DUPE_DECL(0x3FD4000005E00011) };

__asm {
    vbroadcast zmm4, big_num
    vmovapd zmm0, One
    vmovapd zmm5, dc1
    vmovapd zmm6, dc2
    vmovapd zmm7, dc3

    vrsqrt14pd zmm3, zmm4
    vfpclasspd k1, zmm4, 05ch
    vmulpd zmm1, zmm3, zmm4, {rn-sae}
    vfmadd231pd zmm0, zmm3, zmm1
    vfmsub231pd zmm1, zmm3, zmm4, {rn-sae}
    vfmadd213pd zmm1, zmm3, zmm0
    vmovups zmm0, zmm7
    vmulpd zmm2, zmm3, zmm1
    vfmadd213pd zmm0, zmm1, zmm6
    vfmadd213pd zmm0, zmm1, zmm5
    vfmadd213pd zmm0, zmm2, zmm3
    vorpd zmm0 {k1}, zmm3, zmm3
}

```



### 13.24.4.16 倍精度逆数平方根 50 ビット

```

/* Input:
   zmm3 = vector of a's
   Output:
   zmm4 = vector of 1/sqrt(a)
*/

// duplicates x eight times
#define DUP8_DECL(x) x,x,x,x,x,x,x,x

// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))

ALIGNTO(64) __int64 One[] = { DUP8_DECL(0x3FF0000000000001) };
ALIGNTO(64) __int64 dc1[] = { DUP8_DECL(0x3FE0000000000001) };
ALIGNTO(64) __int64 dc2[] = { DUP8_DECL(0x3FD80000046000011) };
ALIGNTO(64) __int64 dc3[] = { DUP8_DECL(0x3FD4000005E800011) };

__asm {
    vmovapd zmm5, One
    vmovapd zmm6, dc1
    vmovapd zmm8, dc3
    vmovapd zmm7, dc2

    vrsqrt14pd zmm2, zmm3
    vfpclasspd k1, zmm3, 05ch
    vmulpd zmm0, zmm2, zmm3, {rn-sae}
    vfmadd213pd zmm0, zmm2, zmm5
    vmulpd zmm1, zmm2, zmm0
    vmovapd zmm4, zmm8
    vfmadd213pd zmm4, zmm0, zmm7
    vfmadd213pd zmm4, zmm0, zmm6
    vfmadd213pd zmm4, zmm1, zmm2
    vorpd zmm4 {k1}, zmm2, zmm2
}

```

### 13.24.4.17 倍精度逆数平方根 26 ビット

```

/* 入力:
   zmm0 = a のベクトル
出力:
   zmm1 = 1/sqrt(a) のベクトル
*/

double half = 0.5;
__asm {
    vrsqrt14pd zmm1, zmm0
    vmulpd zmm0, zmm0, zmm1
    vbroadcastsd zmm3, half
    vmulpd zmm2, zmm1, zmm3
    vfmadd213pd zmm2, zmm0, zmm3
    vfmadd213pd zmm1, zmm2, zmm1
}

```

### 13.24.4.18 倍精度逆数平方根 14 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm2 = vector of 1/sqrt(a)
*/
__asm {
    vrsqrt14pd zmm2, zmm0
}

```

### 13.24.4.19 倍精度平方根 53 ビット (IEEE)

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm2 = vector of sqrt(a)
*/
__asm {
    vsqrtpd zmm2, zmm0
}

```

### 13.24.4.20 倍精度平方根 52 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm0 = vector of sqrt(a)
*/
double half = 0.5;
__asm {
    vbroadcastsd zmm4, half
    vrsqrt14pd zmm1, zmm0
    vfpclasspd k2, zmm0, 0ch
    vmulpd zmm2, zmm0, zmm1, {rn-sae}
    vmulpd zmm1, zmm1, zmm4
    knotw k3, k2
    vmovapd zmm3, zmm4
    vfmadd231pd zmm3, zmm1, zmm2, {rn-sae}
    vfmadd213pd zmm2, zmm3, zmm2, {rn-sae}
    vfmadd213pd zmm1, zmm3, zmm1, {rn-sae}
    vfmadd231pd zmm0 {k3}, zmm2, zmm2, {rn-sae}
    vfmadd213pd zmm0 {k3}, zmm1, zmm2
}

```

### 13.24.4.21 倍精度平方根 26 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm0 = vector of sqrt(a)
*/

// duplicates x eight times
#define DUP8_DECL(x) x,x,x,x,x,x,x,x

// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))

ALIGNTO(64) __int64 OneHalf[] = { DUP8_DECL(0x3fe0000000000001) };

__asm {
    vrsqrt14pd zmm1, zmm0
    vfpclasspd k2, zmm0, 0x0
    knotw k3, k2
    vmulpd zmm0 {k3}, zmm0, zmm1
    vmulpd zmm1, zmm1, ZMMWORD PTR [OneHalf]
    vfmadd213pd zmm1, zmm0, ZMMWORD PTR [OneHalf]
    vfmadd213pd zmm0 {k3}, zmm1, zmm0
}

```

### 13.24.4.22 倍精度平方根 14 ビット

```

/* Input:
   zmm0 = vector of a's
   Output:
   zmm0 = vector of sqrt(a)
*/

__asm {
    vrsqrt14pd zmm1, zmm0
    vfpclasspd k2, zmm0, 0xe
    knotw k3, k2
    vmulpd zmm0 {k3}, zmm0, zmm1
}

```

## 13.25 コンパイラーを利用するヒント

この節では、インテル® コンパイラーを使用して Skylake Server<sup>+</sup> 上で最高のパフォーマンスを得るために重要なコンパイラー・オプションについて説明します。コンパイラーのオプションとチューニングのヒントに関する詳細は、製品ページのドキュメントをご覧ください: <https://software.intel.com/en-us/intel-softwaretechnical-documentation> (英語)。例えば、インテル® C++ コンパイラー 17.0 デベロッパー・ガイドおよびリファレンスは以下で入手できます: <https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-referenceguide> (英語)。

多くのオプション名は、Windows\* では先頭に Q が付くことを除き、Linux\*、OS X\*、Windows\* で同じです。この場合ドキュメントでは、[Q]option-name のように記載されます。

デフォルトの最適化レベルは O2 です (デバッグオプションが指定される場合を除く)。O2 オプションは、ベクトル化を含む多くのコンパイラーによる最適化を有効にします。O3 オプションは、ループ構造が主体な HPC アプリケーションなどに推奨され、キャッシュを効率良く利用するループ・フュージョンやループ・ブロッキングなど、より積極的なループとメモリアクセスの最適化を有効にします。

Skylake Server<sup>†</sup> マイクロアーキテクチャー上で最高のパフォーマンスを発揮するには、プロセッサ固有のオプション [Q]xCORE-AVX512 を使用してコンパイルします。このオプションでコンパイルされた実行可能ファイルは、非インテル製プロセッサや、下位の命令セットをサポートするインテル製プロセッサ上では実行されません。

Skylake Server<sup>†</sup> マイクロアーキテクチャーと Knights Landing<sup>†</sup> マイクロアーキテクチャーベースのインテル® Xeon Phi™ プロセッサの両方で実行可能な共通バイナリーを作成する場合、[Q]xCOMMON-AVX512 オプションを使用します。このオプションは、ターゲット固有のオプション [Q]xCORE-AVX512 (Skylake Server<sup>†</sup>) と [Q]xMIC-AVX512 (Knights Landing<sup>†</sup>) で生成されたバイナリーに比べ、両方のマイクロアーキテクチャーでパフォーマンス上のコストを伴います。

また、Skylake Server<sup>†</sup> マイクロアーキテクチャー向けには、`-qopt-zmm-usage=low/high` (Linux\*) や `/Qopt-zmm-usage:low/high` (Windows\*) オプションを追加して `zmm` コードの生成を最適化できます。引数レベル `low` は、Skylake Server<sup>†</sup> マイクロアーキテクチャー上でインテル® AVX2 ISA からインテル® AVX-512 ISA へのスムーズな移行を可能にします。これは、エンタープライズ向けのアプリケーションで効果を発揮します。ZMM 命令向けのチューニングには、`#pragma omp simd simdlen()` などの明示的なベクトル構文を使用することを推奨します。引数レベル `high` は、より広いベクトル操作を使用して命令ごとの計算量を高めるため、ベクトル計算が主体の HPC コードなどのアプリケーションに適しています。Skylake Server<sup>†</sup> マイクロアーキテクチャーをターゲットとする [Q]xCORE-AVX512 と [Q]xCOMMON-AVX512 などより上位の CORE-AVX512/MIC-AVX512 を組み合わせたコンパイルでのデフォルトは `low` です。

また、[Q]ax ターゲットオプションを使用して、複数の命令セットをサポートする自動ディスパッチ・バイナリーを生成することもできます (コードサイズは大きくなります)。例えば、[Q]axCORE-AVX512,CORE-AVX2 オプションでアプリケーションをコンパイルすると、コンパイラーは Skylake Server<sup>†</sup> マイクロアーキテクチャーとインテル® AVX2 をターゲットとする特殊なコードを生成します。また、このバイナリーには、インテル® SSE2 をサポートする非インテル製プロセッサや、インテル製プロセッサで実行可能なデフォルトのコードパスも生成されます。実行時にアプリケーションは、自動的に動作環境のプロセッサを識別します。そして、検出されたプロセッサに最も適したコードパスが実行されます。指定されたターゲットと一致しない場合、デフォルトのコードパス (インテル® SSE2) が実行されます。また、指定されるオプションにかかわらず、コンパイラーは実行時のプロセッサ検出に基づくコードパスのディスパッチにより、`memset/memcpy` など最適化されたバージョンの呼び出しに置き換えるコードを挿入する可能性があることに注意してください。

`-qopt-report[n]` (Linux\*)、`/Qopt-report[:n]` (Windows\*) オプションは、コンパイラーによって適用された最適化のレポートを生成します。デフォルトではレポートは `.optprt` 拡張子のファイルへ書き込まれます。n には、0 (レポートなし) から 5 (最も詳しい) の詳細レベルを指定します。`.-qopt-report-phase=[list]` (Linux\*)、`/Qopt-report-phase[:list]` (Windows\*) オプションを使用して、コンパイラーが生成する最適化レポートに含まれる最適化フェーズを指定できます (`cg`, `ipo`, `loop`, `offload`, `openmp`, `par`, `vec`, `pgo`, `tcollect`, `all`)。デフォルトは `all` です。このレポートは、コンパイラーが適用した、または適用できなかったパフォーマンス最適化の詳細を知ること、インライン展開、OpenMP\* 並列化、ループの最適化 (ループ分割やアンロールなど)、およびベクトル化など各種最適化の相互関係を理解するのに役立ちます。

レポートはコンパイラーの静的解析を基に生成されます。このレポートは、インテル® Advisor やベクトル化アドバイザー (インテル® Advisor の機能) によるホットスポット解析や他の動的解析情報と組み合わせるとさらに有用です。一度情報を入手できると、コンパイラー・レポートのホットスポット (関数/ループの入れ子) に対する最適化情報を理解できます。コンパイラーが複数バージョンのループの入れ子を生成できるため、実際に実行されるバージョンを解析する際の関連付けに役立つことを覚えておいてください。コンパイラーのループ最適化におけるフェーズの入れ替えは、最適なベクトル化を可能にします。ループ最適化のパラメーターを理解することは、将来のパフォーマンス・チューニングに役立つでしょう。多くのケースでは、プラグマ、ディレクティブおよびオプションによってループの最適化を適切に制御できます。

アプリケーション・コードが OpenMP\* プラグマやディレクティブを含み、OpenMP\* ベースのスレッド化とベクトル化有効にするには、`-qopenmp` (Linux\*)、`/Qopenmp` (Windows\*) を指定してコンパイルします。あるいは、`-qopenmp-simd` (Linux\*)、`/Qopenmp-simd` (Windows\*) オプションを使用して OpenMP\* の SIMD ベクトル化機能だけを有効にできます。

コンパイラーによる自動ベクトル化が有効であるかを試すには、`-no-vec -no-simd -qno-openmp-simd` (Linux\*)、`/Qvec- /Qsimd- /Qopenmp-simd-` (Windows\*) オプションを使用してベクトル化を完全に無効化してみると良いでしょう。

効率良くベクトル化を行うには、データのアライメントが重要です。通常これは、プログラマーまたはアプリケーションによる 2 つの手順で行われます。

Y データをアライメントします。

Fortran プログラムをコンパイルする際に、`-align array64byte` (Linux\*)、`/align:array64byte` (Windows\*) オプションを使用して、ほとんどの配列の先頭アドレスを 64 バイト境界のメモリーアドレスにできます。C/C++ プログラムでは、メモリー割り当てに `__mm_malloc(n, 64)` 関数などを使用して 64 バイトにアライメントされたポインタを要求します。データ・アライメントの詳細については、<https://www.isus.jp/products/c-compilers/data-alignment-to-assist-vectorization/> をご覧ください。

Y 適切な句、プラグマ、およびディレクティブを使用してコンパイラーにアライメント情報を使えます。

`-O3 -xcore-avx512 -qopt-prefetch[=n]` (Linux\*)、`/O3 /QxCORE-AVX512 /Qopt-prefetch[:n]` (Windows\*) オプションを指定して、コンパイラーによるソフトウェア・データ・プリフェッチを有効にできます。ここで、`n` には 0 (プリフェッチを行わない) から 5 (最大限にプリフェッチを行う) の値を指定できます。`n=5` を指定すると、コンパイラーはループ内のインデックスやストライドを使用したロード/ストアに対し、ハードウェア・プリフェッチを無視して積極的にプリフェッチを行います。`n=2` を指定すると、コンパイラーによるプリフェッチの数を減らし、ハードウェア・プリフェッチャーがうまく処理できないとコンパイラーが判断した直接メモリーアクセスに対してのみプリフェッチが生成されます。特定のアプリケーションに対して最適なプリフェッチの用法を調査するため、`n=2` から 5 の値を試してみることが推奨します。また、`-qopt-prefetch-distance=n1[,n2]` (Linux\*)、`/Qopt-prefetch-distance: n1[,n2]` (Windows\*) オプションを使用して、アプリケーションのパフォーマンスをさらに細かく調整することができます。

Y `n1` には次の値が有効です: 0,4,8,16,32,64。

Y `n2` には次の値を指定します: 0,1,2,4,8。

ループカウントをコンパイラーに伝えることができないと、ホットスポット中で実行時に判明した比較的少ないループ回数の子は、インテル® AVX-512 のパフォーマンス引き出せないことがあります。多くの場合、コンパイラーはループ回数、ループのストライド、および配列範囲 (Fortran の多次元配列など) を知ることで適切なコードを生成し、高いパフォーマンスを発揮することが可能です。それができない場合、ループに対し `#pragma loop_count` を追加するのが有用であることがあります。

`-ipo` (Linux\*)、`/Qipo` (Windows\*) オプションを使用して、プロシージャー間の最適化を有効にできます。このオプションは、アプリケーションのすべてのソースファイルに、またはホットスポットを含むソースファイルを選択して最適化を適用できます。IPO は、複数のソースファイルにまたがるインライン展開や、その他のプロシージャー間の最適化を可能にします。特定の状況では、このオプションはコンパイル時間とコードサイズを大幅に増やすことがあります。コンパイラーによるインライン展開を制御するには、`-inline-factor=n` (Linux\*)、`/Qinline-factor:n` (Windows\*) オプションを利用できます。デフォルトで `n` は 100 です。これは、100% または 1 つのスケール要素を示します。例えば、200 と指定した場合、上限を定義するすべてのインライン展開オプションの値に係数 2 が掛けられ、より多くのインライン展開を有効にします。

`-prof-gen` と `-prof-use` (Linux\*)、`/Qprof-gen` と `/Qprof-use` (Windows\*) オプションを使用してプロファイルに基づく最適化 (PGO) を有効にできます。一般に、PGO を使用すると IPO の効率を高めます。`-fp-model name` (Linux\*)、`/fp:name` (Windows\*) オプションは、浮動小数点結果のパフォーマンス、精度、および一貫性間のトレードオフを制御します。`name` のデフォルトは `fast=1` です。`fast=2` に変更すると、精度と一貫性を少し犠牲にして、より積極的な最適化を可能にします。`name` に `precise` を指定すると、浮動小数点データの精度に影響する最適化を無効にします。`name` に `double`、`extended`、または `source` を指定すると、それぞれの精度で丸めを行います (`double` 53 ビット、`extended` 64 ビット、`source` ソースで定義)。浮動小数点の一貫性と再現性が必要な状況では、`-fp-model precise -fp-model source` (Linux\*)、`/fp:precise /fp:source` (Windows\*) オプションが推奨されます。

数学ライブラリー関数の精度を設定するには、`-fimf-precision=name` (Linux\*)、`/Qimf-precision=name` (Windows\*) オプションを使用できます。デフォルトでコンパイラーは、数学ライブラリー関数を呼び出すときに `medium` 精度を使用します。指定可能な `name` の値は、`high`、`medium` そして `low` です。精度を低下させると、特にベクトル化されたコードのパフォーマンスが向上する可能性があります。その逆もあり得ます。`-[no-]prec-div` および `-[no-]prec-sqrt` (Linux\*)、`/Qprec-div[-]` および `/Qprec-sqrt[-]` (Windows\*) オプションは、浮動小数点除算と平方根の精度を向上 [低下] させます。パフォーマンスが低下 [向上] することがあります。浮動小数点オプションの詳細は、<https://www.isus.jp/products/c-compilers/consistency-of-floating-point-results/> をご覧ください。

`-[no-]ansi-alias` (Linux\*)、`/Qansi-alias[-]` (Windows\*) オプションは、ANSI および ISO C 標準のエイリアシング規則を有効 [無効] にします。Linux\* ではデフォルトでこのオプションが有効化されますが、Windows\* では無効になっています。特に Windows\* 上の C++ プログラムでは、`/Qansi-alias` オプションを追加してコンパイラーが最適化を適用できるようにします。この最適化には、ANSI 標準の型ベースのディスアンビゲーションなどが含まれます。

最適化レポートで、コンパイル時間を短縮するために一部の最適化が無効化されていることが報告されている場合、`-qoverride-limits` (Linux\* のみ) オプションを使用して無効化を上書きして最適化を適用できます。特に大きなポディーを持つ関数呼び出しを行うアプリケーションでは、重要になることがあります。このオプションを追加すると、コンパイル時間が増加し、コンパイル時に使用されるメモリーも劇的に増えることがあります。

以下は、ファイン・チューニング向けの最適化でループレベルの制御に使用可能な例を示しています。コンパイラーのレポートで報告された特定の変換を無効にする方法を含みます。

- Y `#pragma simd reduction(+:sum)`  
ループ内のベクトル演算で変数 `sum` に `+` にリダクション操作が行われることをコンパイラーに指示します。
- Y `#pragma loop_count min(220) avg(300) max(380)`  
Fortran シンタックス: `!dir$ loop count(16)`
- Y `#pragma vector aligned nontemporal`
- Y `#pragma novector` // ベクトル化を抑制
- Y `#pragma unroll(4)`
- Y `#pragma unroll(0)` // ループアンロールを抑制
- Y `#pragma unroll_and_jam(2)` // 外部ループの前で
- Y `#pragma nofusion`
- Y `#pragma distribute_point`  
for ループの制御文の直後に記述されると、そのループの分割は抑制されます。  
Fortran シンタックス: `!dir$ distribute point`
- `#pragma prefetch *:<ヒント>:<距離>`  
ループ中のすべての配列に一定のプリフェッチ距離を適用します。
- `#pragma prefetch <変数>:<ヒント>:<距離>`  
それぞれの配列を細かく制御します。
- `#pragma noprefetch [<変数>]`  
プリフェッチを無効にします [特定の配列に対し]。
- `#pragma forceinline (recursive)`  
関数呼び出しの前に記述すると、コンパイラーに対する呼び出しチェーン全体を (再帰的に) インライン展開するヒントとなります。

## Optimization Notice

### FTC の最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

注意事項の改訂 #20110804

## 14.1 はじめに

インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) は、ロックベースのプログラミングモデルを維持する一方、ロックで保護されたクリティカル・セクションのパフォーマンスを向上することを目的としています。

インテル® TSX を利用すると、プロセッサは、スレッドをロックで保護されたクリティカル・セクションによりシリアル化する必要があるかどうかを動的に判断して、必要な場合にのみシリアル化を行います。これにより、ハードウェアは、Lock Elision (ロックの省略) として知られている手法を用いて、不要な動的同期によって損なわれているアプリケーションの並行性 (コンカレンシー) を生かすことができます。

ロックの省略により、ハードウェアは、開発者が指定したクリティカル・セクション (トランザクション領域とも呼ばれる) をトランザクション実行します。実行の際、ロックされた変数はトランザクション領域内で読み取られるだけで書き込まれません (したがって取得されない)。つまり、ロックされた変数はトランザクション領域で変更されないため、並行性を生かすことができます。

トランザクション実行に成功すると、ハードウェアはトランザクション領域内で行われたすべてのメモリー操作が、ほかの論理プロセッサからは瞬時に起こったように見えるようにします。プロセッサは、コミットに成功した場合のみ、ほかの論理プロセッサに見える領域内でアーキテクチャーの更新を行います。このプロセスは**アトミックコミット**とも呼ばれます。トランザクション領域内で行われた変更は、アトミックコミットが行われることでほかの論理プロセッサに見えるようになります。

成功したトランザクション実行ではアトミックコミットが保証されるため、プロセッサは明示的な同期を行うことなくプログラマーが指定したコード領域を安全だと推定して実行します。特定の実行で同期が不要だった場合、交差するスレッド間のシリアル化を行うことなく実行をコミットできます。

トランザクション実行が成功しなかった場合、プロセッサは更新をアトミックにコミットできません。安全だと仮定した実行に失敗すると、プロセッサは実行をロールバックし、プロセスはトランザクション・アポートと見なされます。トランザクションがアポートすると、プロセッサは領域で実行された更新をすべて破棄し、安全だと推定した実行が行われなかったように見えるようにアーキテクチャー上の状態を復元し、非トランザクションに実行を再開します。有効なポリシーに応じて、ロックの省略が再試行されるか、処理を進めるため明示的にロックが取得されます。

インテル® TSX では 2 つのプログラム・インターフェイスが用意されています。

- Y Hardware Lock Elision (HLE) は、従来のプロセッサと互換性のある命令セット拡張 (XACQUIRE および XRELEASE プリフィクス) です。
- Y Restricted Transactional Memory (RTM): 新しい命令セット・インターフェイス (XBEGIN および XEND 命令を包括) です。

従来のハードウェアでインテル® TSX 対応のソフトウェアを実行する場合は、HLE インターフェイスを使用してロックの省略を実装します。一方、従来のハードウェアに対応する必要がなく、より複雑なロック・プリミティブを扱う場合は、インテル® TSX の RTM インターフェイスによりロックの省略を実装します。新しい命令を使用する後者のケースでは、トランザクション実行が行われない場合に備えて、開発者はトランザクション・アポートが生じた後に実行する (省略されたロックを取得するコードを含む) 非トランザクション・パスを常に提供する必要があります。

さらに、インテル® TSX には、論理プロセッサがトランザクション実行しているかどうかをテストする XTEST 命令と、トランザクション領域をアポートする XABORT 命令も用意されています。



プロセッサは、さまざまな理由によりトランザクション実行をアボートします。最も多い原因は、トランザクション実行している論理プロセッサと別のプロセッサ間のデータアクセス競合によるものです。このようなアクセス競合はトランザクション実行の成功の妨げとなります。トランザクション領域内から読み取られたメモリーアドレスによりトランザクション領域の**読み取りセット**が構成され、トランザクション領域内へ書き込まれたアドレスによりトランザクション領域の**書き込みセット**が構成されます。インテル® TSX は、キャッシュラインの粒度で読み取りセットと書き込みセットを維持します。RTM を使用するロックの省略では、明示的にロックを取得する別のスレッドがある場合でもトランザクション実行するスレッドが正しく動作するように保証するため、省略されるロックのアドレスを読み取りセットに追加する必要があります。

別の論理プロセッサがトランザクション領域の書き込みセットの一部の場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットの一部の場所で書き込みを行うと、データアクセス競合が発生します。これは**データ競合**と呼ばれます。インテル® TSX は、キャッシュライン単位でデータ競合を検出するため、同じキャッシュラインに配置された無関係なデータの場所も競合として検出されます。トランザクション・アボートは、トランザクション・リソースの制限により発生することもあります。例えば、領域でアクセスしたデータ量が実装固有の処理能力を超えた場合です。CPUID や IO などの命令は、実装により常にトランザクション実行をアボートすることがあります。

インテル® TSX のインターフェイスに関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 16 章で確認できます。

以降のセクションでは、インテル® TSX 命令を使用するソフトウェア開発者向けのガイドラインを説明します。ガイドラインでは、プリフィクス・ヒント (HLE) や新しい命令 (RTM) を通じて、ロックで保護されたクリティカル・セクションの並行性を生かせるようにロックを省略するインテル® TSX 命令の使用法に注目します。インテル® TSX 命令はロックの省略以外にも利用できますが、それらの使用法はここでは説明しません。

以後**ロックの省略**という用語は、ロックを省略する HLE ベースまたは RTM ベースの実装のいずれかを指します。

### 14.1.1 最適化の概略

ここからは、インテル® TSX 命令を使用してロックを省略する際に、マルチスレッド・アプリケーションを最適化およびチューニングするための推奨事項を示します。インテル® TSX は、ロックを取得した後のアプリケーションの動作を見逃しがちなマイクロカーネルの代わりに、アプリケーションのパフォーマンスを向上させます（「14.2 アプリケーション・レベルのチューニングと最適化」を参照）。この後のセクションでは、インテル® TSX を使用してロックの省略を行う同期ライブラリーを作成する方法（「14.3 インテル® TSX 対応の同期ライブラリーの開発」を参照）、インテル® TSX のパフォーマンス監視機能を効果的に使用する方法（「14.4 インテル® TSX のパフォーマンス監視サポートを利用する」を参照）、そして初期実装のパフォーマンス・ガイドライン（「14.5 パフォーマンス・ガイドライン」を参照）についても述べていきます。

最初に、すべてのクリティカル・セクションのロックを有効化してから、問題のあるクリティカル・セクションを特定することを推奨します。この「ボトムアップ」アプローチによってアプリケーションの評価とチューニングが単純化され、開発者は適切なクリティカル・セクションに注目できます。インテル® TSX のチューニングに関連するリソースは、<http://www.intel.com/software/tsx> (英語) で入手できます。

## 14.2 アプリケーション・レベルのチューニングと最適化

アプリケーションは通常、**同期ライブラリー**を利用してクリティカル・セクションに関連するロック取得とロック解放機能を実装しています。これらのアプリケーションでインテル® TSX ベースのロックの省略を活用する最も簡単な方法は、インテル® TSX 対応の同期ライブラリーを利用することです。既存のライブラリーがインテル® TSX 命令 (14.2.1 を参照) の利点を活用できるかもしれません。既製品で、インテル® TSX 対応のライブラリーが利用できない場合、14.3 節「インテル® TSX 対応の同期ライブラリーの開発」で、インテル® TSX 未対応のロック・ライブラリーをインテル® TSX 命令を使うように拡張する方法を示しています。インテル® TSX 対応の同期ライブラリーと従来の同期ライブラリーは、互換性を持って使用できます。

これらのライブラリーを使用するアプリケーションは、アプリケーションを変更することなくインテル® TSX を使用できますが、トランザクション実行のコミット率を増加させ、トランザクション・アボートによる無駄な実行サイクルを抑える基本的なチューニングとプロファイリングを行うことでパフォーマンスを向上できます。チューニングでは、最初にプロファイリングツール (14.4 節「インテル® TSX のパフォーマンス監視サポートを利用する」を参照) を用いてアプリケーションのトランザクション動作の特性を把握することを推奨します。プロファイリングツールは、ハードウェアに実装されているパフォーマンス監視機能とサンプリング機能を利用して、アプリケーションのトランザクション動作に関する詳細な情報を提供します。ツールは、パフォーマンス・モニタリング・カウンターやプリサイズ・イベント・ベース・サンプリング (PEBS) メカニズムなどのプロセッサ機能が使用します。詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章を参照してください。

インテル® TSX 対応の同期ライブラリーを使用するアプリケーションは、従来の同期ライブラリーを使用する場合と同じ動作になる必要がありますが、インテル® TSX によりレイテンシーが短縮されてスレッド間の同期が以前よりも速くなるため、コードの潜在的なバグが表面化する可能性があります。

## 14.2.1 既存の TSX 対応ロック・ライブラリー

このセクションでは、ロックの省略のためすでにインテル® TSX に対応している既製品のライブラリーについてまとめられています。これらは完全なものではなく、2015 年前半時点のスナップショットを示しています。ここで紹介するすべてのライブラリーが、完全にチューニングされているわけではありません。

### 14.2.1.1 プログラムの変更なしにロックの省略を可能にするライブラリー

- Y Linux\* では、GNU\* glibc 2.18 で PTHREAD\_MUTEX\_DEFAULT タイプの pthread ミューテックスのロック省略のサポートが追加されました。glibc 2.19 では読み取り/書き込みミューテックスのロック省略のサポートが加えられました。ロックの省略が有効であるかどうかは、ライブラリーのコンパイル時に `--enable-lock-elision=yes` パラメーターが設定されているかどうかによって依存します。
- Y Java\* JDK 8u20 以降では、`-XX:+UseRTMLocking` オプションが有効である場合に同期セクションで適応型の省略がサポートされます。
- Y インテル® Composer XE 2013 SP1 以降では、OpenMP\* の `omp_lock_t` 向けのロックの省略がサポートされています。ロックの省略を有効にするには、“`export KMP_LOCK_KIND=adaptive`” を設定します。

### 14.2.1.2 プログラムの修正を必要とするライブラリー

- Y インテル® スレディング・ビルディング・ブロック (インテル® TBB) では、`speculative_spin_rw_mutex` でロックの省略をサポートしています。この新しいロックタイプを使用するようにプログラムを変更する必要があります。
- Y gcc 4.8 以降では、インテル® TSX によるソフトウェア・トランザクション・メモリーの実装の加速をサポートします。
- Y Concurrency Kit は、`ck_elide` ラッパーでスピンロックのロック省略をサポートします。
- Y DPDK ライブラリーは、スピンロックと read-write ロックのロック省略をサポートします (“\_tm” サフィックス付きの `lock/unlock` 呼び出しを介して)。

## 14.2.2 初期のチェック

いくつかの簡単なチェックを行うことで、チューニングの労力を軽減できます。特に、優れたライブラリー実装の利用とクリティカル・セクション内部の統計収集の扱いについては重要です。

- Y インテル® TSX 対応の優れた同期ライブラリーを使用します。アプリケーションは、インテル® TSX 対応の同期ライブラリーを直接使用する必要があります。インテル® TSX 対応ライブラリーの上に独自のカスタム・ライブラリーを実装すると、トランザクション領域を識別できないことがあります。インテル® TSX 向けに同期ライブラリーを作成する方法は、14.3 節をご覧ください。

- Y クリティカル・セクション内部の統計を収集しないようにします。クリティカル・セクション（および同期ライブラリー自身）は共有のグローバル統計カウンターを使用することがありますが、これらのカウンターはデータ競合やトランザクション・アボートを引き起こします。アプリケーションには通常、これらの統計収集を無効にするフラグが用意されているので、初期のチューニング段階でこれを無効にしておく、本来のデータ競合に注目しやすくなります。

### 14.2.3 アプリケーションの実行とプロファイル

マルチスレッド・アプリケーションにおけるスレッドの相互作用を視覚化することは困難です。最初のステップでは、インテル® TSX 対応の同期ライブラリーを使ってアプリケーションを実行し、パフォーマンスを測定します。次に、プロファイリング・ツールで結果を解析します。つまり、プロファイリング・ツールによりトランザクション実行されたアプリケーション・サイクルを測定し、アプリケーションのトランザクション実行の割合を把握すべきです（14.4 節「インテル® TSX のパフォーマンス監視サポートを利用する」を参照）。

トランザクション実行サイクルの比率が低くなる要因はいくつかあります。

アプリケーションがクリティカル・セクションで同期をほとんど使用していない、この場合、ロックの省略の利点は得られません。

- Y アプリケーションの同期ライブラリーがすべてのプリミティブにインテル® TSX を使用していない、アプリケーションが、クリティカル・セクションのロックの一部に内部カスタム関数やカスタム・ライブラリーを使用している場合、この問題が発生します。これらのロックの実装をトランザクション実行に変更する必要があります（「14.4.2 無効化するロックを特定してすべてのロックが無効化されることを確認する」を参照）。
- Y アプリケーションが、ロックの省略に対応した同期ライブラリーで提供されるものとは異なる、より高いレベルのロック構造（本書ではメタロックと呼んでいます）を使用している。このような場合、構造をロックの省略にさせる必要があります（14.3.7 節「省略するロックを特定してすべてのロックが省略されることを確認する」を参照）。
- Y プログラムが LOCK プリフィクス命令をクリティカル・セクション以外で使用している。この場合、アルゴリズムがトランザクション実行に対応していない限り、インテル® TSX を利用する利点はありません。このようなロック目的以外での使用法は本書では触れていません。

インテル® TSX パフォーマンス・チューニングの「ボトムアップ」アプローチでは、チューニング手法を次のようにモジュール化できます。

- Y ロックをすべて特定します。
- Y ロックをすべて省略するインテル® TSX 同期ライブラリーを使用してプログラムを（変更せずに）実行します。
- Y プロファイリング・ツールでトランザクション実行を測定します。
- Y 必要に応じて、トランザクション・アボートの原因を調べて対処します。

### 14.2.4 トランザクション・アボートを最小限に抑える

**データ競合**はキャッシュ・コヒーレンス・プロトコルを介して検出されます。データ競合はトランザクションのアボートにつながります。初期の実装では、データ競合を検出するスレッドはトランザクションをアボートします。

HLE ベースのトランザクション実行がトランザクションのアボートに遭遇すると、現在の実装ではハードウェアは HLE 実行を開始した XACQUIRE プリフィクス付きの命令を再開しますが、XACQUIRE プリフィクスは無視されます。その結果、ロックの省略なしで再実行し、明示的にロックが取得されます。RTM ベースのトランザクション実行がトランザクションのアボートに遭遇すると、現在の実装ではハードウェアは XBEGIN 命令で提供される命令アドレスから再開します。

初期のインテル® TSX 実装は制限付きの入れ子をサポートします。RTM は 7 階層の入れ子レベルをサポートします。HLE は 1 階層の入れ子レベルをサポートします。これは実装依存であり、同じ世代のプロセッサ・ファミリーの今後の実装では変更される可能性があります。

『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』の第 16 章では、トランザクションのアボートを引き起こすさまざまな原因が詳しく説明されています。インテル® TSX 命令とプリフィクスに関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2B』で確認できます。

プロファイリング・ツールのパフォーマンス監視機能を使用して、アボートしたトランザクション実行に費やされたサイクル数を計算できます。すべてのトランザクション・アボートがパフォーマンスを低下させるとは限りません。別のスレッドが取得したロックを待機するために実行がストールしている場合や、トランザクション実行にデータ・プリフェッチ効果がある場合もあります。

プロファイリング・ツールは、PEBS を用いてアボートしたトランザクション領域を認識し、相対的なコストに関する情報を提供します (14.4 節「インテル® TSX のパフォーマンス監視サポートを利用する」を参照)。次に、トランザクション・アボートの一般的な原因と対策を示します。

**チューニングの推奨事項 4:** プロファイリング・ツールを用いて、パフォーマンス低下の多くの原因であるトランザクション・アボートを識別します。

トランザクション・アボートの原因には次のようなものがあります。

- Y データアクセス競合によるアボート。
- Y ロック変数の競合によるアボート。
- Y リソースバッファの超過によるアボート。
- Y HLE インターフェイス固有の制限によるアボート。
- Y 『Intel® Architecture Instruction Set Extensions Programming Reference』の第 8 章で説明されているその他のアボート。

#### 14.2.4.1 データ競合によるトランザクション・アボート

別の論理プロセッサがトランザクション領域の書き込みセットの一部の場所で読み取りを行うか、トランザクション領域の読み取りセットまたは書き込みセットの一部の場所で書き込みを行うと、データ競合が発生します。初期の実装では、データ競合は、キャッシュライン単位で処理されるキャッシュ・コヒーレンス・プロトコルにより検出されず。

ここでは、トランザクション・アボートの原因となるさまざまなデータ競合について説明します。アプリケーションにもともと内在する競合は回避できるものもあります。

##### フォルス・シェアリングによる競合

異なる変数を同じキャッシュライン (64 バイト) へ配置して複数のスレッドで個別に書き込みを行うと、フォルス・シェアリングが発生します。ハードウェアはキャッシュライン単位でデータ競合をチェックするため、アドレスがオーバーラップしていなくてもこれらの変数は同じアドレスを持つように見え、不要なトランザクション・アボートを引き起こします。

このフォルス・シェアリングの問題はインテル® TSX 固有の問題ではありません。キャッシュ・コヒーレンス・プロトコルはシステム内でキャッシュラインを移動しますが、これには大きなオーバーヘッドが伴います。少なくとも変数の 1 つが異なるスレッドによって頻繁に書き込まれる場合、異なる変数を同じキャッシュラインに配置するべきではありません。

**チューニングの推奨事項 5:** 競合する変数はパディングを追加して別々のキャッシュラインに配置します。

**チューニングの推奨事項 6:** 可能な場合、フォルス・シェアリングが最小限になるようにデータ構造を再構成します。

## 真の共有による競合

このトランザクション・アボートはフォルス・シェアリングによるものではなく、競合データが実際に共有されている場合に発生しますが、ソフトウェアを変更することで軽減できることがあります。次のセクションで、このような競合への対処方法を説明します。

## 統計管理による競合

一部のソフトウェアは、複数のスレッド間で共有されるグローバル統計カウンターを使用しています。例えば、クリティカル・セクションのロックを取得したか、または取得された回数をカウントする同期ライブラリーなどがあります。また、複数のスレッドによってアクセスされるグローバル変数やオブジェクトの数をカウントするような場合も考えられます。これらの統計はトランザクション・アボートの原因となります。このようなトランザクション・アボートに対処するには、統計の使用目的を理解する必要があります。

プログラムのロジックに影響しない場合は、統計を無効にしたり、条件付きでスキップすることができます。例えば、クリティカル・セクションのシリアル実行の回数をカウントするケースについて考えてみます。ロックが省略されない場合、実行はすでにシリアル化されているため、クリティカル・セクション内で統計が更新されます。ロックが省略されている場合、省略されているロックをカウントしても意味がありません。この統計が役立つのは、ロックが省略されなかった場合だけです。その場合、ソフトウェアは統計を利用してシリアル化のレベルを追跡できます。ロックを省略しない実行の場合（つまり、シリアル化されている場合）のみ、XTEST 命令で統計を更新できます。これらの統計はプログラムの開発中にのみ有用であり、製品版では無効にできます。

しかし、場合によっては、統計を無効にしたり、スキップできないこともあります。これらの統計を論理スレッドごとに維持することで、(フォルス・シェアリングを回避しつつ) 不要なトランザクション・アボートを回避できます。このアプローチでは、統計の読み取り時にすべてのスレッドの結果をまとめる必要があります。これはまた、スレッド間の通信を最小限に抑えることで、インテル® TSX 命令を利用しなくてもアプリケーションのパフォーマンスを向上できます。

このほかにも、クリティカル・セクションの外に統計を移動したり、アトミック操作で統計を更新する方法があります。これらのアプローチではトランザクション・アボートは軽減されますが、アトミック操作によるオーバーヘッドや、通信のオーバーヘッドは軽減されません。

**チューニングの推奨事項 7:** グローバル統計は操作ごとに更新する代わりにサンプリングすることもできます。

**チューニングの推奨事項 8:** クリティカル・セクションでは不要な統計は避けます。

**チューニングの推奨事項 9:** クリティカル・セクションでは統計をスレッド単位で維持することを検討します。

プログラマーは、共有グローバル統計によるトランザクション・アボートを軽減するため最適なアプローチを選択する必要があります。初期テストでは、すべてのグローバル統計を無効にすると、グローバル統計が問題かどうかを判断しやすくなります。

## データ構造内の資源管理による競合

データ構造内の資源管理による操作も、データ競合を引き起こす原因の 1 つです。例えば、データ構造は、構造内のエントリー数を追跡するために変数を持つことがあります。これは、統計カウンターと同様の影響があり、不要なトランザクション・アボートを引き起こします。

状況によっては（例えば、ヒープの再構成を引き起こすエントリーの数など）、アトミック更新を使用することでクリティカル・セクション外に更新を移動できます。

また別のケースでは、データ競合が発生するタイミングを減らすアプローチを採用できることもあります (14.2.4.1 節「データ競合によるトランザクション・アボート」を参照)。

## メモリー割り当ての競合

クリティカル・セクションでメモリー割り当てが行われることがあります。その場合、スレッドのローカル領域にフリーリストを保持し、割り当て済みメモリーのフォルス・シェアリングを回避するスレッド対応のメモリー・アロケーション・ライブラリーを使用することを推奨します。

### 条件付き書き込みリダクションによる競合

一般的なソフトウェア・パターンには、値がほとんど変わらない共有変数やフラグの更新が含まれます。そのような操作は、(値が同じ場合でも) キャッシュラインを更新するためキャッシュラインへの書き込み許可が必要になります。そのような操作では、共有変数へアクセスするほかのスレッドでトランザクション・アボートが発生します。このようなデータ競合は、値が同じ場合はストアを実行せずに必要な場合のみ更新を行うことで回避できます (例 14-1 を参照)。

例 14-1 条件付き更新とデータ競合の軽減

<pre>state = true; // 毎回更新 var  = flag;</pre>	<pre>if (state != true) state = true; if (!(var &amp; flag)) var  = flag;</pre>
---	---

### データ競合範囲の軽減

ここで示す手法では、頻発する実際のデータ競合によるトランザクション・アボートを回避できないケースがあります。その場合、データ競合が発生する範囲を短くすることを目標にすべきです。例えば、実際に競合するメモリーアクセスをクリティカル・セクションの最後に移動してタイミングを短縮します。

## 14.2.4.2 トランザクション・リソースの制限によるトランザクション・アボート

インテル® TSX 実装は、共通のトランザクション領域を実行する十分なリソースを提供しますが、トランザクション領域の実装の制限やデータ使用量によりトランザクションがアボートすることがあります。アーキテクチャーがトランザクション実行に必要なリソースを保証したり、トランザクション実行の成功を保証することはありません。

プロセッサは、L1D (L1 データ) キャッシュ内の読み取りセットアドレスと書き込みセットアドレスの両方を追跡します。

読み取りセットアドレスの追い出し (eviction) は、第 2 レベルの構造でそのラインが追跡される可能性があるため、即座にトランザクション・アボートにつながるとは限りません。現在の実装では、第 2 レベルの構造は追い出された読み取りセットアドレスを確立的に追尾します。その結果、他のスレッドからのアクセスによって偽りの一致が生じ、必要がないトランザクションのアボートが発生する可能性があります。このような偽りの競合の割合は、異なるスレッドからのアドレスストリームと正確なハードウェア実装によります。Broadwell<sup>†</sup> マイクロアーキテクチャーの実装では第 2 レベルの構造が改善されています。偽りの競合の割合は、将来の実装では減少すると考えられます。

アーキテクチャーはバッファリングに関し何も保証しないため、ソフトウェアはどのような保証も期待してはいけません。

Haswell<sup>†</sup>、Broadwell<sup>†</sup> および Skylake<sup>†</sup> マイクロアーキテクチャーの L1 データキャッシュの連想性は 8 です。これはこの実装では、同じキャッシュセットに割り当てられる 9 番目の位置への書き込みを実行するトランザクションがアボートすることを意味します。しかし、このマイクロアーキテクチャーの実装が、同じセットへのわずかなアクセスではアボートしないことが保証されることを意味するものではありません。

さらに、インテル® ハイパースレッディング・テクノロジーが有効である構成では、L1 キャッシュが同じコアの 2 つの論理プロセッサ間で共有されるため、同じコアの一方の論理プロセッサの操作が追い出しを引き起こし、有効な読み取りと書き込みセットのサイズを大幅に減少させます。

プロファイラーを利用して、処理能力の制限により頻繁にアボートするトランザクション領域を特定できます (14.4.4 節「プロファイリング・ツールを利用してアボートを分類する」を参照)。ソフトウェアは、そのようなトランザク

ション領域内でデータの過剰なアクセスを避けるべきです。一般に、大量のデータアクセスには時間がかかるため、そのようなアボートは多くの実行サイクルを無駄にします。

クリティカル・セクションでのデータ使用量はアルゴリズムの変更により軽減できることもあります。例えば、ソートされた配列には、リニア検索ではなくバイナリー検索を利用することで、クリティカル・セクション内でアクセスするアドレスの数を減らすことができます。

トランザクション領域の特定のコードパスで大量のデータにアクセスすることが想定される場合、アルゴリズムで (XABORT 命令により) 早期にトランザクション・アボートを強制したり、無効化されたロックを取得してアボートせずに非トランザクション実行に遷移できます (「14.2.6 節「アボートが頻発するトランザクション領域またはトランザクション・パスへの対応」を参照)。

トランザクション領域内の動作が原因で、処理能力によるアボートが発生することがあります。例えば、アプリケーションが初めてダイナミック・ライブラリー関数を呼び出す場合、ソフトウェア・システムはダイナミック・リンカーを呼び出してシンボルを解決する必要があります。初めての呼び出しがトランザクション領域内で行われる場合、大量のデータアクセスが生じて通常アボートが発生します。このアボートは、ダイナミック・ライブラリー関数が初めて呼び出されるときのみ発生します。これが頻繁に発生する場合は、非トランザクション実行中にトランザクション実行パスが使用されていないことが原因である可能性が高いと考えられます。

### 14.2.4.3 ロック省略固有のトランザクション・アボート

データ競合に加えて、ロック自体の競合によってトランザクション・アボートが生じることもあります。その場合、クリティカル・セクションのトランザクション実行と非トランザクション実行がオーバーラップするタイミングを検出する必要があります。インテル® TSX を利用してロックを省略する場合、ロックは読み取りセットに追加されます。これは、HLE では自動的に行われますが、RTM ではソフトウェア・ライブラリーで明示的に行う必要があります。これにより、明示的にロックを取得するほかのスレッドとの競合を確認できます。これは、アボートし、再開して、最終的にロックを取得するというトランザクション実行の自然な流れの一部です。

HLE と RTM を利用するロックの省略では、このようにロック変数に対する二次的な競合が原因でアボートが発生することがよくあります。トランザクション・スレッドのアボートは通常非トランザクション実行に遷移し、その過程で明示的にロックを取得します。このロックの取得により、ほかのトランザクション実行スレッドをシリアル化しなければならなくなりアボートします。

RTM の場合、ロックが解放されるのを待って取得を試みることで、フォールバック・ハンドラーはこれらの二次的なアボートを減らせる可能性があります (14.3.5 節「RTM フォールバック・ハンドラーのガイドライン」を参照)。

### 12.2.4.4 HLE 固有のトランザクション・アボート

一部のトランザクション・アボートは、HLE ベースのロックの省略でのみ発生します。これらについては、次のセクションで説明します。

#### サポートされていないロックの省略パターン

トランザクション実行を正常にコミットするには、ロックがある特性を持ち、ロックへのアクセスが特定のガイドラインに従っていなければなりません。XRELEASE プリフィクス命令では、ロックの省略の値を、対応する XACQUIRE プリフィクス命令 (ロックの取得) が実行される前の値に復元する必要があります。

これにより、ハードウェアは書き込みセットに追加することなく、安全にロックを省略できます。XRELEASE プリフィクス命令 (ロックの解放) と XACQUIRE プリフィクス命令 (ロックの取得) のデータサイズとデータアドレスは、どちらも一致していなければなりません。また、ロックはキャッシュ境界をまたぐことはできません。例えば、アドレス A への XACQUIRE プリフィクス命令 (ロックの取得) の後に別のアドレス B への XRELEASE プリフィクス命令 (ロックの解放) がある場合、アドレス A とアドレス B は一致しないためアボートします。

## サポートされていない HLE 領域内のロック変数へのアクセス

通常、ロック変数は HLE 領域からアボートせずに読み取ることができます。しかし、ある種の一般的でないアクセスはトランザクション・アボートを引き起こすことがあります。例えば、省略されたロック変数へのアライメントされていないアクセスや一部がオーバーラップするアクセスは、トランザクション・アボートとなります。その場合、省略されたロック変数へ適切にアライメントされたアクセスが行われるように、ソフトウェアを変更すべきです。

HLE ベースのトランザクション領域内で省略されたロックへ書き込みを行う場合は、必ず XRELEASE プリフィクス命令を使用します。そうしないと、トランザクション・アボートが発生します。

### 14.2.4.5 その他のトランザクション・アボート

プログラマーは、トランザクション領域内ですべての命令を安全に使用でき、すべての特権レベルでトランザクション領域を使用できます。しかし、一部の命令は常にトランザクション実行をアボートさせ、実行を非トランザクション・パスに安全かつシームレスに遷移させます。そのようなトランザクション・アボートは、プロファイリング・ツールによって収集される PEBS レコードのトランザクション・アボート・ステータスには命令アボート (Instruction Aborts) として表示されます (14.4 節「インテル® TSX のパフォーマンス監視サポートを利用する」を参照)。

この命令の一覧は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル』に記載されています。一般的な例として、x87 およびインテル® MMX® 命令のアーキテクチャー・ステートに対する操作、セグメント、制御、デバッグレジスターを更新する操作、IO 命令、SYSENTER、SYSCALL、SYSEXIT、SYSRET などのリング遷移を引き起こす命令が挙げられます。

プログラマーは、トランザクション領域内では、x87/インテル® MMX® 命令の代わりに、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)/インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) 命令を使用すべきです。ただし、トランザクション領域内でインテル® SSE 命令とインテル® AVX 命令を混在させる場合は注意が必要です。XMM レジスターにアクセスするインテル® SSE 命令と YMM レジスターにアクセスするインテル® AVX 命令の混在使用は、トランザクション・アボートの原因となります。VZERoupper 命令もアボートの原因になることがあるため、この命令はクリティカル・セクションの前に移動すべきです。

特定の 32 ビット呼び出し規約は、引数と戻り値の受け渡しに x87 ステートを使用しています。プログラマーは別の呼び出し規約を利用するか、あるいは関数をインライン展開することを検討すべきです。long double 型などの一部のデータ型は x87 命令を使用するため避けるべきです。

命令ベースによるアボートに加えて、各種ランタイムイベントによりトランザクション実行がアボートされるケースもあります。

トランザクション実行中に非同期イベント (NMI、SMI、INTR、IPI、PMI、その他) が発生すると、トランザクション実行はアボートされ、非トランザクション実行に移行します。そのようなアボートの発生率は、オペレーティング・システムのバックグラウンド・ステートに依存します。例えば、オペレーティング・システムのタイマーを用いて割り込みをかけると、トランザクション・アボートの原因になることがあります。

トランザクション実行中に同期例外イベント (#BR、#PF、#DB、#BP/INT3 など) が発生すると、トランザクション実行はコミットされず、非トランザクション実行が必要になります。これらのイベントは、発生しなかったように処理されます。

ページフォルト (#PF) は通常、プログラムの起動時に発生することが最も多いイベントです。ページが初めて割り当てられるため、この間にトランザクション領域がアボートする確率が高くなります。このようなアボートは、プログラムが定常状態になると発生しなくなります。ただし、実行時間の非常に短いプログラムではこのアボートが頻発することがあります。同様の現象は、大きなメモリー領域が割り当てられた際にも起こります。

トランザクション領域内のメモリーアクセスを行うには、プロセッサが参照するページ・テーブル・エントリーの Accessed フラグと Dirty フラグをセットする必要があります。この処理は、ページへの初回アクセス時と書き込み



時に行われます。現在の実装では、これらの処理はトランザクション・アボートを引き起こします。非トランザクション・モードで再実行するとこれらのビットが適切に更新され、通常、後続のトランザクション実行ではこれが原因のトランザクション・アボートが発生しません。このトランザクション・アボートは、PEBS レコードのトランザクション・アボート・ステータスには命令アボート (Instruction Aborts) として表示されますが、頻発しない限り特に注意する必要はありません。

さらに、実装固有の条件やバックグラウンド・システム・アクティビティーがトランザクション・アボートの原因になることもあります。例えば、システムのキャッシュ階層によるアボート、プロセッサのマイクロアーキテクチャー実装との微妙な相互作用、システムタイマーによる割り込みなどが挙げられます。このようなアボートは、インテル® TSX を使用するロックの省略ではごくまれです。

**チューニングの推奨事項 10:** プログラム起動時のトランザクション領域では、定常状態よりも高い確率でアボートが発生します。

**チューニングの推奨事項 11:** バックグラウンド・アクティビティーにより、まれにオペレーティング・システムのサービスがトランザクション・アボートの原因になることがあります。

## 14.2.5 トランザクション実行専用のコードパスの使用

インテル® TSX を使用するプログラマーは、非トランザクション・フォールバック・パスとは異なる、トランザクション領域でのみ実行されるコードを記述できます。これは RTM (フォールバック・ハンドラーを利用) や HLE と XTEST 命令を使用することで可能になります。

ただし、トランザクション実行時に実行されるコードが、非トランザクション実行時に実行されるコードと大きく異なる場合は注意が必要です。(命令およびデータの) ページフォルトなどのイベントや、アクセスビットとダーティービットを変更するページ操作は、トランザクション実行を繰り返しアボートする可能性があります。そのため、非トランザクション・フォールバック・パスでもこれらの操作が実行されるようにしなければなりません。そうしないと、そのトランザクション領域はいつまでも成功できません。ロックの省略により、アプリケーションのトランザクション・パスと非トランザクション・パスは同じになり、唯一の違いは同期ライブラリーだけなので、これは一般的な問題ではありません。

XTEST 命令は、トランザクション実行時にアボートの原因になる可能性の高い不要なコードシーケンスをスキップするのに利用できます。また、巻き戻された (unwind) コードや実際にロックを取得したときのみ必要なエラー処理コード (デッドロック検出など) をスキップして最適化を実装するのにも利用できます。

**チューニングの推奨事項 12:** トランザクション実行専用のコードパスは単純にしてインライン展開します。

**チューニングの推奨事項 13:** トランザクション実行専用のコードパスは最小限に抑えます。

## 14.2.6 アボートが頻発するトランザクション領域またはトランザクション・パスへの対応

一部のトランザクション領域は高い確率でアボートし、これまでに示した手法ではうまくアボートを減らすことができないことがあります。その場合は、以降のセクションに記載する手法を検討してみると良いでしょう。

### 14.2.6.1 アボートしないで非省略実行へ遷移する

システムコールや IO 操作など、トランザクション・アボートが避けられないこともあります。

トランザクション実行パスでアボートが避けられない場合、ロックの省略に RTM を利用しているソフトウェアでは、非トランザクション実行に遷移しロックの取得を試みて、成功した場合はトランザクション実行をコミットすることができます。例 14-2 に簡単な例を示します。実際のコードでは入れ子処理などの追加が必要になります。

### 例 14-2 アボートしないで非省略実行へ遷移する

```
/* RTM トランザクションでトランザクション実行がアボートした場合は */
/* 非トランザクション実行でロックを取得する */
<オリジナルのロック取得コード>
_xend(); /* コミット */
/* アボート処理を行う */
```

## 14.2.6.2 早期のアボート強制

トランザクション領域内のアボートを引き起こすパスで、早い段階に PAUSE 命令または XABORT 命令を挿入し、強制的にトランザクション・アボートの発生を試みることができます。これにより、破棄が必要な作業を最小限に抑えることが可能です。

## 14.2.6.3 選択したロックを省略しない

ロックの省略によりアプリケーションのパフォーマンスが低下し、どの手法でもトランザクション・アボートを軽減できない場合、トランザクション・アボートの発生率が高く、コストの高い特定のロックの省略をソフトウェアで無効にできます。その場合、アボートの発生率が高くてパフォーマンスが向上することがあるため、アプリケーション・レベルのパフォーマンス・メトリックを用いて検証すべきです。

## 14.3 インテル® TSX 対応の同期ライブラリーの開発

このセクションでは、インテル® TSX 命令を使用して、既存の同期ライブラリーをロックの省略に対応させる方法を示します。

### 14.3.1 HLE プリフィックスの追加

プログラマーは、クリティカル・セクションを保護するロックの取得に使用する命令の前に XACQUIRE プリフィックスを使用します。プログラマーは、クリティカル・セクションを保護するロックの解放に使用する命令の前に XRELEASE プリフィックスを使用します。この命令にはロックに対する書き込みも含まれます。

この命令により、同じロックの XACQUIRE プリフィックスが付加されたロック取得操作の前の値にロックの値が戻された場合、プロセッサはロックの解放に関連付けられている外部書き込み要求を省略し、書き込みセットにロックのアドレスを追加しません。

### 14.3.2 省略に適したクリティカル・セクションのロック

ライブラリー自体がデータ競合の原因にならないようにする必要があります。ライブラリーのデータ競合の一般的な例を次に示します。

- Y ロックの所有権フィールドの競合
- Y ロック関連の統計の競合

ロックの省略に HLE を利用する場合、プログラマーは既存のコードパスにトランザクション実行を追加する必要があります (HLE では、ロックが省略される場合も、されない場合も実行されるコードパスは同じであるため)。また、共有された場所への書き込み操作は、そのロック変数に対するロックの取得/解放命令によってのみ行われることを確認すべきです。共有された場所へのほかの書き込み操作は、通常、ロック省略ライブラリーを利用して共通のロックを省略する 2 つのスレッド間でデータ競合を引き起こします。これは、複数のスレッドを使って共有ロックで保護された空のクリティカル・セクションを繰り返し実行するテストによって簡単に特定できます。

### 14.3.3 ロックの省略における HLE または RTM の使用

プロセッサが HLE 拡張と RTM 拡張をサポートしているかどうかは、CPUID 情報によって判断できます。ただし、HLE プリフィクス (XACQUIRE と XRELEASE) は、プロセッサが HLE をサポートしているかどうかを確認しなくても使用できます。HLE をサポートしていないプロセッサはこれらのプリフィクスを無視し、トランザクション実行に入らずにコードを実行します。一方、RTM 命令 (XBEGIN、XEND、XABORT) を使用するアプリケーションは、プロセッサが RTM 命令をサポートしているかどうかを確認する必要があります。RTM 命令をサポートしていないプロセッサで実行すると、#UD (未定義オペコード) 例外が発生します。XTEST 命令も、CPUID 情報をチェックして HLE か RTM のいずれかがサポートされていることを確認する必要があります。どちらもサポートされていない場合、この命令も #UD 例外を引き起こします。繰り返し確認しなくても済むように、CPUID 情報のある変数に格納しておくとも良いでしょう。

HLE では、トランザクション実行中のプロセッサがクリティカル・セクションでロックの値を読み取ると、プロセッサがロックを取得したように見えます (トランザクション実行のものではない値が返されます)。そのため、HLE 実行と HLE プリフィクスなしの実行が機能的に同じになります。

RTM インターフェイスを利用して、プログラマーはより複雑な同期アルゴリズムを記述したり、トランザクション・アボート後の再試行ポリシーを制御することができます。RTM ベースのロック実装では、複数のコードパスを持つラッパーとして実装することが推奨されます。つまり、1 つのパスで RTM ベースのロックを実行し、別のパスで RTM ベースではないロックを実行します (14.3.4 節「ロック省略に RTM を使用するラッパーの例」を参照)。通常、RTM ベースではないロックのコードを変更する必要はありません。一度だけ再試行するプリミティブを使うことでパフォーマンスが向上する可能性があります。このプリミティブにより、スレッドはロックが解放された後にロックの省略を再試行できます。

RTM 命令は明示的にロックに関連付けられていないため、ロックの省略に RTM 命令を使用するソフトウェアは、トランザクション領域内でロックの状態を確認し、ロックがフリーの場合のみトランザクション実行を継続すべきです。さらに、ロックがフリーでないときの再試行ポリシーも定義する必要があります。

HLE との違いは、RTM ベースのクリティカル・セクション内でロックを読み取ると、ロックはフリーのままであり、取得されていないかのように見えることです。そのため、ロックの値を返すライブラリー関数は、トランザクション実行をアボートして非トランザクション実行の値を返す必要があります (14.3.9 節「RTM ベースのライブラリーで省略されたロックの値を読み取る」を参照)。HLE 命令では、明示的にロックアドレスが関連付けられており、正しい値が返されることがハードウェアによって保証されているため、このような状況は発生しません。

**ユーザー/ソース・コーディング規則 36:** ロックの省略に RTM を利用する場合は、常にトランザクション領域内でロックをテストします。

**チューニングの推奨事項 14:** ラッパーでロック変数を読み取れない場合、RTM ラッパーは使用しないようにします。

### 14.3.4 ロックの省略に RTM を使用するラッパーの例

このセクションでは、RTM 命令を使用してロックの省略を実装するラッパーを記述する方法を示します。既存のロック実装 (省略なし) をラッパーで囲み、ラッパー内に新しいパスを追加してロックの省略を実装します。ラッパーでロックを省略する場合と、しない場合のそれぞれのコードパスを記述します。ロックを省略しないで取得するパスは、ロックを省略するパスが成功しなかったときのみ実行されます。このアプローチでは、ロックを省略しない場合のパスは変更されません。これは、チケットロックや reader-writer (読み取り/書き込み) ロックなどのさまざまなロックに適用できます。

例 14-3 にコードシーケンスの例を示します (使用されている組込み関数の説明は、14.7 節「インテル® TSX 用の一般的な組込み関数」を参照してください)。

例 14-3 ロックの取得/解放プリミティブに RTM を使用するラッパーの例

```

void rtm_wrapped_lock(lock) {
    if (_xbegin() == _XBEGIN_STARTED) {
        if (lock is free)
            /* 読み取りセットにロックを追加する */
            return; /* トランザクション実行 */
        _xabort(0xff);
        /* 0xff はロックがフリーでないことを示す */
    }
    /* トランザクション・アボートの後に実行されるコード */
    original_locking_code(lock);
}

void rtm_wrapped_unlock(lock) {
    /* ロックがフリーの場合は、ロックが省略されたと仮定する */
    if (lock is free)
        _xend(); /* コミット */
    else
        original_unlocking_code(lock);
}

```

例 14-3 の `_xabort()` はロックがフリーでない場合、トランザクション実行を終了します。代わりに、`_xend()` を使用しても同様の動作になります。ただし、プロファイリング・ツールは `_xabort()` 操作とアボートコード `0xff` (ソフトウェア規約) を簡単に認識し、ロックがフリーでなかったケースであると判断できます。`_xend()` が使用されると、プロファイリング・ツールは、このケースとロックの省略に成功したケースを区別することができません。

上記の例は、1 回だけ再試行し、トランザクション・アボートのさまざまな原因は区別しないという基本ポリシーを示すために単純化されています。より高度な実装では、トランザクション・アボートの原因に関する情報に基づいてロックごとに省略を試すかどうかを決定するヒューリスティックを追加します。また、ロックがフリーでない場合、ブロックした後にロックの省略を再試行するコードを追加します。これには、同期ライブラリーへのわずかな変更が必要になることがあります。

プログラミング・エラーが原因で、スレッドが解放済みのロックを解放しようとする場合があります。このエラーはすぐに明らかにならないことがあります。しかし、前述のラッパーでロックの解放関数を前述の RTM 対応のライブラリーに置換すると、XEND 命令がトランザクション領域外で実行されます。この場合、ハードウェアは #GP 例外を発生します。一般に、このエラーはオリジナルのアプリケーションで修正したほうが良いでしょう。代替手段として、ソフトウェアでエラーになったコードパスを保持しようとする場合は XTEST で XEND を保護できます。

### 14.3.5 RTM フォールバック・ハンドラーのガイドライン

RTM 用のフォールバック・ハンドラーでは、RTM ベースのトランザクション実行が成功しなかったときに実行されるコードパスを記述します。インテル® TSX はトランザクション実行の成功を保証していないため、RTM フォールバック・ハンドラーは単にトランザクション実行の再試行を繰り返すのではなく、処理を進めることを保証しなければなりません。

**チューニングの推奨事項 15:** ロックの省略に RTM を利用する場合、ロックを取得することで処理が進むことを簡単に保証できます。

フォールバック・ハンドラーが明示的にロックを取得すると、そのロックを省略する他のすべてのトランザクション実行スレッドがアボートし、実行がシリアル化されます。これは、ロックがトランザクション領域の読み取りセットにあることを保証することで達成されます。

ソフトウェアは、EAX レジスターのアボート情報を参照して、トランザクション実行を再試行する場合とフォールバックして明示的にロックを取得する場合のヒューリスティックを作成できます。例えば、`_XABORT_RETRY` ビット

が設定されていないと、トランザクション実行の再試行は他のアボートを引き起こす可能性が高くなります。フォールバック・ハンドラーは、このような場合とロックがフリーでない場合 (例えば、\_XABORT\_EXPLICIT ビットが設定されているが、\_XABORT\_CODE()<sup>12</sup> が「ロックがビジー」状態であることを示す 0xff を返す場合) を区別する必要があります。ロックがフリーでない場合は待機後に再試行しなければなりません。

アボートの原因がデータ競合 (\_XABORT\_CONFLICT) であれば、ほとんどのデータ競合は一時的であるため、時間をおいて再試行することでパフォーマンスが向上する可能性があります。ただし、このような再試行の回数は制限し、無制限に繰り返すべきではありません。

ハイパースレッディングが有効である構成では、容量アボート (\_XABORT\_CAPACITY) に対する少数の再試行は有効である場合があります。L1 キャッシュは HT スレッド間で共有されるリソースであり、一方のスレッドが他のスレッドのデータを押し出す可能性があります。再試行には合理的な成功の見込みがあります。この場合、ステータスコードの \_XABORT\_RETRY ビットを無視する必要があります。それ以外ではいかなる理由があっても \_XABORT\_RETRY ビットを無視すべきではありません。

一般にコア数が多く複数のソケットを持つシステムでは再試行の回数が増加します。

一般に、ロックがフリーでない場合、フォールバック・ハンドラーはロックがフリーになるまでトランザクション実行の再試行を待機すべきです。そうすることでロックの省略なしで非トランザクション実行にとどまるのを防ぐことができます。このような状況は、フォールバック・ハンドラーが、ロックがフリーでもトランザクション実行を試みることでできないために生じます (14.3.8 節「永続的な非省略実行を回避する」を参照してください)。

**ユーザー/ソース・コーディング規則 37:** RTM アボートハンドラーは有効なテスト済みの非トランザクション・フォールバック・パスを提供する必要があります。

**チューニングの推奨事項 16:** ロックがビジー状態の場合は、ロックがフリーになるまで再試行を待機します。

## 14.3.6 インテル® TSX による省略に適したロックの実装

このセクションでは、一般的なロック・アルゴリズムにおいて、インテル® TSX 命令を使用するロックの省略に適したバージョンを実装するアプローチについて述べています。このセクションで触れないアルゴリズムにも同様のアプローチを適用できます。

### 14.3.6.1 HLE を使用する単純なスピンロックの実装

スピンロックは、単純でよく使用されているロック・アルゴリズムです。このアルゴリズムでは、スレッドはロックがフリーかどうかを確認してから、LOCK プリフィクス命令でロックを取得します。ロックがフリーでない場合、スレッドはロックがフリーになるまで (通常は、ロックの値を保持するローカル・データ・キャッシュの読み取り操作により) スピンして待機します。

例えば、値がゼロのときはロックがフリーで、そうでない場合はほかのスレッドによって取得されていると仮定します。ロックは通常のストア命令によって解放されます。

<sup>12</sup> \_XABORT\_CODE は RTM アボートコードを示す xabort ステータスにアクセスします。

例 14-4 は、C11 規格に似た gcc 4.8 以降の**アトミック組込み関数**を用いています。ここでは、推奨されるアプローチに従って gcc 4.8+ 組込み関数を使用したスピンロックを実装しています。このスピンロックで HLE を有効にするのに必要な変更は、`__ATOMIC_HLE_ACQUIRE` フラグと `__ATOMIC_HLE_RELEASE` フラグの追加だけです。残りのコードは HLE を利用しない場合と同じです。

例 14-4 GCC 4.8 以降で HLE を使用するスピンロックの例

```
#include <immintrin.h> /* _mm_pause() に必要 */
/* ロックを 0 に初期化 */
void hle_spin_lock(int *lock)
{
    while (__atomic_exchange_n(lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) != 0)
    { int val;
      /* 再試行する前にロックがフリーになるのを待機する*/
      do {
          _mm_pause(); /* アボートのスペキュレーション */
          __atomic_load_n(lock, &val, __ATOMIC_CONSUME);
      } while (val == 1);
    }
}

void hle_spin_unlock(int *lock)
{
    __atomic_clear(lock, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
}
```

以下に、同じスピンロックを Windows\* 用の C/C++ コンパイラー (Microsoft\* Visual Studio\* 2012 とインテル® C++ コンパイラー 17.0) の組込み関数を用いて実装する例を示します。

例 14-5 インテル® コンパイラーと Microsoft\* コンパイラーの組込み関数で HLE を使用するスピンロックの例

```
#include <intrin.h> /* _mm_pause() に必要 */
#include <immintrin.h> /* HLE 組込み関数に必要 */
/* ロックを 0 に初期化 */
void hle_spin_lock(int *lock)
{
    while (_InterlockedCompareExchange_HLEAcquire(&lock, 1, 0) != 0){
        /* 再試行する前にロックがフリーになるのを待機する */
        do {
            _mm_pause(); /* アボートのスペキュレーション */
            /* コンパイラーによる命令の並べ替えと待機ループのスキップを防ぎ
            IA 上で追加のフェンス命令が生成されないようにする */
            _ReadWriteBarrier();
        } while (lock == 1);
    }
}

void hle_spin_unlock(int *lock)
{
    _Store_HLERelease (lock, 0);
}
```

HLE スピンロックのアセンブラー実装については、14.7 節「インテル® TSX 用の一般的な組込み関数」を参照してください。

### 14.3.6.2 インテル® TSX を使用する reader-writer (読み取り/書き込み) ロックの実装

reader-writer (読み取り/書き込み) ロックは、クリティカル・セクションの大部分が読み取り専用のおかげによく使用されます。このロックはクリティカル・セクションの読み取りアクセスがシリアル化されるのを防止しますが、共有された場所に対してはアトミック操作 (LOCK プリフィクスが付加された XADD または CMPXCHG のことが多い) を要求し、複数の読み取り間で通信が必要になります。ロックの省略は、読み取りと競合しない書き込みが通信することなく同時に処理できることを除いて、基本的にすべてのロックが reader-writer (読み取り/書き込み) ロックと同様の動作となります。

前述のラッパーアプローチにより、RTM を使って reader-writer (読み取り/書き込み) ロックを省略できます。ロックの省略との唯一の違いは、reader-writer (読み取り/書き込み) ロックのアルゴリズムでは通常、読み取りと書き込みの両方の状態を確認してロックがフリーかどうかを判断していることです。そのため、読み取り/書き込みのそれぞれの状態を別々のキャッシュラインに配置できれば、読み取りのトランザクション実行と非トランザクション実行を並列に行えます。読み取りは、書き込みの状態がフリーであることを確認するだけで済みます。

HLE を利用する場合、ロックが省略されるときと、されないときのコードパスは同じになります。一部の reader-writer (読み取り/書き込み) ロックの実装では実際のクリティカル・セクションではなく、読み取り/書き込み状態の保護にロックを使用しています。この場合、まずロックを 1 つのアトミック操作から成る高速なパスに変更する必要があります。このパスでは、ロック変数が配置されているキャッシュラインは変更すべきではありません。具体的には、読み取りの数と書き込みの数を 1 つのフィールドにまとめ、ロックの取得/解放関数に LOCK プリフィクスを付加した XADD 命令や CMPXCHG 命令を使ってこのフィールドをアトミックに確認/更新します。HLE プリフィクス (XACQUIRE と XRELEASE) を、LOCK プリフィクスが付加されたこれらの命令に追加します。興味深いことに、このアプローチは、インテル® TSX を使用しなくても reader-writer (読み取り/書き込み) ロックのパフォーマンスを向上させます。また、RTM ラッパーを使用することで、トランザクション実行用と非トランザクション実行用に異なるロックの取得パスが用意されるため、ロック構造の変更を回避できます。

**チューニングの推奨事項 17:** 読み取り/書き込みロックでは基本ブロックのロックではなく、ロック操作全体を省略します。

### 14.3.6.3 インテル® TSX を使用するチケットロックの実装

チケットロックも一般的なアルゴリズムです。チケットロックはスピンロックのバリエーションで、共有する場所でスピンしてロックがフリーになったら取得する代わりに、チケットによってどのスレッドがクリティカル・セクションに入ることができるかを決定します。

前述のラッパーアプローチにより、RTM を使ってチケットロックを省略することができます (14.3.4 節「ロックの省略に RTM を使用するラッパーの例」を参照)。

一部のチケットロックの実装は、チケットの値が増えることを想定します。そのようなロックは、取得前と取得後のロックの値が同じでなければならないとする HLE 要件を満たしません。

**チューニングの推奨事項 18:** RTM を使用してチケットロックを省略します。

### 14.3.6.4 インテル® TSX を使用するキューベースのロックの実装

一般に、ロックの省略は複数のスレッドが共通のクリティカル・セクションへ同時に入り、コミットを試みることを前提としています。公正なロックでは、スレッドが先着順にクリティカル・セクションに入り、解放しなければなりません。この 2 つの考え方は、場合によっては対立するよう見えますが、一般的な目的はこれよりも柔軟性があります。

キューベースのロックはスレッドがロック要求のキューを作成する公正なロック方式です。これはチケットロックの亜種とも言えます。

一部の実装では、初期の LOCK プリフィクス命令によりキューが作成されます。その際、その命令に HLE XACQUIRE プリフィクスを追加してロックを省略できます。トランザクション・アボートが発生しなかった場合、ロックの解放後にキューは空になります。しかし、トランザクション・アボートが発生し、アボートしたスレッドがロックを明示的に取得している場合（つまり、キューが生成されている）、後続のスレッドはキューに追加され、ロックが解放されると先頭のスレッドのみがロックの省略を試みます。さらに、別のスレッドがクリティカル・セクションに到達してキューに追加されると、トランザクション実行中のスレッドはアボートし、キューが空になるまで非トランザクション実行になる可能性があります。

これは、キューを使用してロックの省略を試みる実装でのみ発生します。スピンフェーズを省略し最初のスピンの失敗した後にのみキューで待機する適応スピン・スリープ・ロックのような、最初のアトミック操作の後にのみキューを作成する実装には当てはまりません。この問題は、ラッパーを使用する実装 (RTM を使用するものなど) でも存在しません。これらの実装では、キューの処理でスレッドはロックの省略を試みません。

**チューニングの推奨事項 19:** 最初のアトミック操作でキューを実装するロックに RTM ラッパーを使用します。

### 14.3.7 インテル® TSX を使用するアプリケーション固有のメタロックの省略

一部のアプリケーションは、同期ライブラリーを利用してメタロックと呼ばれる独自のロックを構築します。このアプローチでは、アプリケーションは同期ライブラリーのロックによりメタロックのデータを保護します。データを更新したら、ロックを解放します。これは、14.3.6.2 節「インテル® TSX を使用する reader-writer (読み取り/書き込み) ロックの実装」のアプローチに似ています。

アプリケーションは、メタロックを保持しているときにクリティカル・セクションを実行し、メタロックを解放しているときは同期ライブラリーのロックを使ってメタロックデータを保護します。このシーケンスでは同期ライブラリーのロックを省略しても意味がありません。同期ライブラリー内のコードではなく、メタロック自体を省略してアプリケーションのコードをトランザクション実行すべきです。プロファイリング・ツールによってクリティカル・セクションを特定し、(14.3.4 節「ロックの省略に RTM を使用するラッパーの例」に似た) RTM ラッパーによりロックの省略中のメタロックを回避できます。

次のメタロックの実装例について考えてみます。

例 14-6 メタロックの例

```
void meta_lock(Metalock *metalock) {
    __lock(metalock->lock);
    /* ロックのためメタロックの状態を変更 */
    unlock(metalock->lock);
}

void meta_unlock(Metalock *metalock) {
    lock(metalock->lock);
    /* メタロック状態を解放する */
    unlock(metalock->lock);
}

meta_lock(metalock);
/* クリティカル・セクション */
meta_unlock(metalock);
```

上記のコードは次のコードに置き換えることができます。



例 14-7 RTM を使用するメタロックの例

```

void rtm_meta_lock(Metalock *metalock) {
    if (_xbegin() == _XBEGIN_STARTED){
        if (meta_state_is_all_free(metalock))
            return;
        _xabort(0xff);
    }
    meta_lock(metalock);
}

void rtm_meta_unlock(Metalock *metalock) {
    if (meta_state_is_all_free(metalock))
        _xend();
    else
        meta_unlock(metalock);
}

rtm_meta_lock(metalock);
/* クリティカル・セクション */
rtm_meta_unlock(metalock);

```

チューニングの推奨事項 20: メタロックでは基本ブロックのロックではなく、外側のロックをすべて省略します。

14.3.8 永続的な非省略実行を回避する

トランザクション・アボートが起こると、最終的にロックを省略しない非トランザクション実行に遷移します。これは、処理が継続することを保証します。ただし、特定の状況下と一部のロック取得アルゴリズムでは、スレッドがロックの省略を試みずに非トランザクション実行に留まることがあります。これはパフォーマンスの妨げとなります。

この状況を理解するため、HLE を使用する単純なスピンロックの実装例について考えてみます (同様のシナリオは RTM でも試せます)。ロックは値が 0 の場合はフリーで、値が 1 の場合は別のスレッドによって取得されていることを意味します。

HLE を利用するロックの取得シーケンスは、例 14-8 のように記述できます。

例 14-8 HLE を利用するロックの取得/解放シーケンス

```

mov eax,$1
Retry:
XACQUIRE; xchg LockWord,eax
cmp eax,$0 # 値が 0 ならロックの取得に成功
jz Locked
SpinWait:
cmp LockWord, $1
jz SpinWait# 値がまだ 1 のまま
jmp Retry# ロックがフリーなので取得を試みる
Locked:
XRELEASE; mov LockWord,$0

```

スレッドがロックを省略できないときは、省略なしでロックを取得します。ほかのスレッドが同じロックの取得を試みる場合、"XACQUIRE; xchg lockWord, eax" 命令を実行して、ロック操作を省略しトランザクション実行に入ります。しかし、この時点でロックはほかのスレッドによってすでに取得されているため、このスレッドはトランザクション実行中に SpinWait ループに入ります。

これは、トランザクション実行中にハードウェアがクリティカル・セクション・ロックだと認識できず、ロック変数に対するアトミック操作と見なすためです。ハードウェアはロックがフリーでない、ということの意味を理解できません。

ロックを取得しているスレッドがロックを解放すると、そのロックへの書き込み操作によって、現在その場所をスピンしているスレッドのトランザクションがアボートします (ロックの解放操作とトランザクション実行中のスレッドによるロックの読み取りループの間で競合が発生するため)。一度アボートすると、スレッドはロックを省略しないで実行を再開します。これは、すべてのスレッドで起こる可能性があります。スレッドはトランザクション実行中にスピンしますが、ロックが解放されるとロックの省略なしで非トランザクション実行します。これは、ほかにロックの取得を試みるスレッドがなくなるまで繰り返されます。つまり、スレッドは非トランザクション実行に留まることになります。

この問題への簡単な対処法は、SpinWait ループで PAUSE 命令 (アボートを引き起こす) を使用することです。これは、インテル® TSX を使用しない場合でもロックの解放の待機に推奨されるアプローチです。PAUSE 命令は SpinWait ループの非トランザクションへの移行を強制し、ロックが解放されたらスレッドがロックを省略できるようにします。

例 14-9 HLE を使用したスピンウェイトの例

```

mov eax,$1
Retry:
  XACQUIRE; xchg LockWord,eax
  cmp eax,$0# 値が 0 ならロックの取得に成功
  jz Locked
SpinWait:
  pause
  cmp LockWord, $1
  jz SpinWait# 値がまだ 1 のまま
  jmp Retry# ロックがフリーなので取得を試みる
Locked:

```

**チューニングの推奨事項 21:** HLE スピンロックの待機ループには常に PAUSE 命令を使用します。

### 14.3.9 RTM ベースのライブラリーで省略されたロックの値を読み取る

一部の同期ライブラリーは、ロックの値を読み取るインターフェイスを提供しています。RTM を使用してロックを省略するライブラリーは、ロックが読み取られるだけでライブラリー内へ書き込まれないため、省略を行うスレッドがロック変数を取得したかどうかを正確に判断できないことがあります。

場合によっては、ライブラリーのインターフェイスはロックが取得されているかどうかをチェックするだけの単純なテストで、ソフトウェアに正当性チェックを提供します。RTM ベースのライブラリーを使用して関数に正しい値を確実に渡すには、トランザクション実行をアボートして明示的にロックを取得する必要があります。具体的には、XABORT 命令 (`_xabort(0xfe)` を使用して) でアボートを強制します。フォールバック・ハンドラーは `0xfe` コードでこの状況を特定し、読み取りを排除する最適化を行います。また、`_xtest()` 組込み関数で不要なトランザクション・アボートを回避できます。

```
assert(is_locked(my_lock)) => assert(_xtest() || is_locked(my_lock))
```

省略された同期ライブラリー向けの効率良いプリミティブは、取得されたロックや進行中のロックの省略を結合します。次に例を示します。

```
bool is_atomic(lock) { return _xtest() || is_locked(lock); }
```

また、動作を想定できる場合、ロック変数は関数の一部として読み取ることができます。例えば、ロックを取得する `try-lock` インターフェイスは、スレッドがロックの取得を 1 回だけ試みてロックがフリーかどうかを示す値を返します。これは、ロックを取得するためスピンを繰り返すスピンロックとは対照的です。一般に `try-lock` は問題になりませんが、入れ子の `try-lock` により返される値に対しソフトウェアが暗黙の仮定を行っていることがあります。RTM ベースの実装ではロックが省略されるため、ロックがフリーであることを示す値が返されます。ソフトウェアでこの値に対して暗黙の仮定を行っている場合、同期ライブラリーは XABORT 命令で (`_xabort(0xfd)` を使用して) トランザク

ション・アボートを強制できます。ただし、これは一部のプログラムで不要なアボートを引き起こします。プログラムでこのような暗黙の仮定を行うことは推奨されません。また、このような暗黙の仮定が行われることはまれであるため、try-lock の同期ライブラリーではアボートしないことを推奨します。

### 14.3.10 HLE と RTM を混在させる

HLE と RTM は、一般的なトランザクション実行機能に対する 2 つのソフトウェア・インターフェイスの選択肢を提供します。RTM 内で HLE を使用したり、HLE 内で RTM を使用する場合は実装固有です。第 4 世代インテル® Core™ プロセッサの初期実装では、HLE と RTM を混在させるとトランザクション・アボートにつながります。以降のプロセッサでは動作が変わる可能性があります。トランザクション・コミットのセマンティクスは維持されます。

一般に HLE と RTM はロックを省略するという目的は同じですが、ソフトウェア・インターフェイスが異なるため、アプリケーションでは混在しないようにすべきです。しかし、ロックの省略を実装するライブラリー関数は、呼び出し回数があるかどうか、また呼び出し関数が RTM あるいは HLE でロックを省略するライブラリー関数を呼び出しているかどうかを把握していない可能性があります。

これらの状態は、\_xtest() 関数によりソフトウェアで対応できます。例えば、ライブラリーがトランザクション領域内で呼び出されたかどうか、そしてロックがフリーかどうかを確認できます。トランザクション領域内で呼び出された場合は、新しいトランザクション領域を開始しないようにできます。ロックがフリーでない場合、ライブラリーは \_xabort(0xff) 関数で結果を返すことができます。この場合、ロックの解放時に呼び出される関数は、取得操作がスキップされたことを認識できなければいけません。

例 14-10 にこの概念を示します。

例 14-10 HLE と RTM を混在させる概念上の例

```
// ロックの取得シーケンス
// 関数またはスレッドのローカルを使用する
bool lock_in_transactional_region = false;
if (_xtest() && my lock is free) { /* すでにトランザクション領域内である */
    lock_in_transactional_region = true;
} else {
// ロックがフリーの場合は取得し、そうでない場合はアボートする
}
// ロックの解放シーケンス
if (!lock_in_transactional_region) {
    // ロックを解放する
}
```

## 14.4 インテル® TSX のパフォーマンス監視サポートを利用する

インテル® TSX を使用するアプリケーションの解析は、パフォーマンス・カウンターベースのプロファイルにより、トランザクション実行の動作とトランザクション・アボートの原因を明らかにします。インテル® TSX で優れたパフォーマンスを得るには、プロファイリング・ツールで収集したデータを基にアボートを最小限に抑えるチューニングが必要です。通常、アプリケーションのインストールメンテーションよりも、パフォーマンス・カウンターを使用するほうが簡単でコード変更が少ないため推奨されます。『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章と第 19 章に、現在の実装でサポートされる各種パフォーマンス・イベントに関する情報が記載されています。

一般に、プロファイリング・ツールは周期的に割り込みをかけて情報を収集しますが、この割り込みによってトランザクション・アボートするため、プロファイリングはトランザクション実行に影響します。そのため、プロファイリングではこの影響を最小限に抑えるべきです。ただし、トランザクション・アボートのみをプロファイリングする場合は問題になりません。

プログラムの起動時に一度しか実行されないイベントも多数あります。複雑なプログラムのプロファイリングでは起動時のプロファイリングをスキップすることで、これらのイベントによる不要なデータの収集を大幅に減らすことができます。

Linux\* perf、インテル® Performance Counter Monitor、およびインテル® VTune™ Amplifier を含むプロファイリングツールは、インテル® TSX をサポートしています。http://www.intel.com/software/tsx (英語) をご覧ください。

### 14.4.1 トランザクション成功を測定する

最初のステップは、アプリケーションのトランザクション成功を測定することです。これは、次の 3 つのカウンターと Unhalted\_Core\_Cycles イベントを設定することで測定できます。

1. 固定サイクルカウンター (IA32\_FIXED\_CTR0) を使用して FixedCyclesCounter を測定します。
2. IA32\_PERFEVTSEL2 に IN\_TX フィルターと IN\_TXCP フィルターを設定して、IA32\_PMC2 の CyclesInTxCP を測定します。
3. MSR IA32\_PERFEVTSELx (x= 0, 1, 3) に IN\_TX フィルターを設定して、対応するカウンターの CyclesInTxOnly を測定します。

サンプリングはトランザクション・アボートを引き起こす可能性があるため、これらのサイクルの測定にはサンプリングではなくカウンターを使用します。この 3 つの値から、合計サイクル数、トランザクション実行で費やされたサイクル数、アボートされたトランザクション領域で費やされたサイクル数を計算できます。

```
CyclesTotal = FixedCycleCounter
%CyclesTransactionalAborted = ((CyclesInTxOnly - CyclesInTxCP) / CyclesTotal) * 100.0
%CyclesTransactional = (CyclesInTx / CyclesTotal) * 100.0
%CyclesNonTransactional = 100.0 - %CyclesTransactional
```

CyclesTransactional がほぼゼロの場合、アプリケーションはロックベースの同期を使用していないか、インテル® TSX の Hardware Lock Elision (HLE) が有効な同期ライブラリーを使用していません。後者の場合、インテル® TSX 対応の同期ライブラリーを使用すべきです (14.3 節「インテル® TSX 対応の同期ライブラリーの開発」を参照してください)。

CyclesTransactionalAborted が CyclesTransactional と比較して少ない場合、トランザクションの成功率は高く、追加のチューニングは必要ありません。

CyclesTransactionalAborted が CyclesTransactional とほぼ同じ (で少なくない) 場合、ほとんどのトランザクション領域がアボートしていて、HLE は役立ちません。次のステップでは、トランザクション・アボートの原因を特定して、トランザクション・アボート数を減らします (14.2.4 節を参照してください)。

### 14.4.2 省略するロックを特定してすべてのロックが省略されることを確認する

このステップは、トランザクション実行で費やされたサイクルが少ない場合に有効です。これは、省略されるロックが少ないことが原因の可能性があります。MEM\_ OP\_RETIRED.LOCK\_LOADS イベントをカウントし、RTM\_RETIRED.START イベントまたは HLE\_RETIRED.START イベントと比較すべきです。ロックのロード数が、開始されたトランザクションの数よりも大幅に多い場合、すべてのロックが省略として認識されていない可能性があります。MEM\_μop\_RETIRED.LOCK\_LOADS の PEBS バージョンでサンプリングを行い、失われたロックを特定できます。ただし、この手法は省略の対象になっていないメタロックを迅速に検出するには効果的ではありません (14.3.7 節「インテル® TSX を使用するアプリケーション固有のメタロックの省略」を参照してください)。また、MEM\_μop\_RETIRED.LOCK\_LOADS イベントのコールグラフをプロファイリングすることで、アプリケーション・レベルのクリティカル・セクションをトランザクション実行するためインテル® TSX を使用すべき高レベルの同期ライブラリーを特定できます。

### 14.4.3 トランザクション・アボートのサンプリング

ハードウェア実装は、トランザクション・アボートをサンプリングするため PEBS プリサイズイベントを定義しています (HLE の場合は HLE\_RETIRED.ABORTED、RTM の場合は RTM\_RETIRED.ABORTED)。これは実行中のすべてのトランザクション・アボートを正確にプロファイルすることを可能にします。PEBS を有効にしてサンプリングし、トランザクション・アボートが発生するコードの位置を特定します。PEBS ハンドラー (プロファイリングツールの一部) は、PEBS レコードの EventingIP フィールドを用いてトランザクション・アボートの正確なコード位置を報告します。

次のステップでは最も一般的なトランザクション・アボートについて検証し対処します。トランザクション・アボートをサンプリングすることで追加のアボートが発生することはありません。

### 14.4.4 プロファイリング・ツールを利用してアボートを分類する

トランザクション・アボートのプロファイリングにより生成される PEBS レコードには、トランザクション・アボートの原因に関する追加情報を示す TX Abort Information フィールドがあります。TX Abort Information の下位 32 ビットは Cycles\_Last\_TX と呼ばれ、アボート前の最後のトランザクション領域で費やされたサイクル数を示します。このデータからトランザクション・アボートのおよそのコストが分かります。

$$\text{RelativeCostOfAbortForIP} = \text{SUM}(\text{Cycles\_Last\_TX\_For\_IP})$$

トランザクション・アボートにはパフォーマンスを低下させないものもあれば、パフォーマンスに大きく影響するものもあります。プログラマーは、この情報を基にどのトランザクション・アボートに注目すべきかを判断できます。

PEBS レコードの詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の 18.10.5.1 節を参照してください。

アボートを分類できるように、プロファイリング・ツールはアボートコストを表示できなければいけません。

**チューニングの推奨事項 22:** 最もコストの高いアボートを最初に検証します。

**チューニングの推奨事項 23:** TX Abort Information にはトランザクション・アボートに関する追加情報が含まれています。

PEBS レコードの Instruction\_Abort ビット (ビット 34) が設定されている場合、トランザクション・アボートの原因を命令に直接関連付けることができます。それらのアボートに対して、PEBS レコードはトランザクション・アボートの原因となった命令アドレスを記録します。ページフォルト (通常プログラムを終了させるものやプログラム起動時のワーキングセットでフォルトになるものを含む) などの例外もこのカテゴリーに含まれます。

PEBS レコードの Non\_Instruction\_Abort ビット (ビット 35) が設定されている場合、アボートの原因は PEBS レコードで報告された命令アドレスの命令ではない可能性があります。例えば、ほかのスレッドとの間でデータ競合が発生した場合が考えられます。この場合、Data\_Conflict ビット (ビット 37) も設定されます。別の例として、トランザクション実行する読み取りセット/書き込みセットの容量の制限によるトランザクション・アボートが挙げられます。これは、Capacity\_Write (ビット 38) フィールドと Capacity\_Read (ビット 39) フィールドに記録されます。

データ競合によるアボートは、トランザクション領域内のどの命令でも発生する可能性があります。そのため、クリティカル・セクション全体にわたって競合の原因を調査したほうが良いでしょう。PEBS によって報告される EventingIP ではなく、リターン IP (アボートコードの IP) とコールグラフに注目すべきです。通常リターン IP はロックがインライン展開されていない限り、同期ライブラリーを指しているため、呼び出し元からクリティカル・セクションを特定できます。

容量が原因の場合、クリティカル・セクション全体にわたってメモリー使用量を減らすように変更する必要があるの、クリティカル・セクション全体 (ReturnIP のプロファイリング) を調査すると良いでしょう。

**チューニングの推奨事項 24:** 命令のアボートは早期に分析すべきですが、プログラム起動後に発生するコストの高いもののみ分析します。

**チューニングの推奨事項 25:** データ競合や容量の制限によるアボートは、アボート時に報告される命令アドレスだけでなく、クリティカル・セクション全体を調査します。

**チューニングの推奨事項 26:** プロファイラーは、命令が原因でないアボートイベントでは ReturnIP とコールグラフの表示を、命令が原因のアボートイベントでは EventingRIP の表示をサポートしなければなりません。

**チューニングの推奨事項 27:** プロファイリング・ツールはすべての PEBS TX Abort 情報ビットを表示できなければなりません。

### 14.4.5 RTM フォールバック・ハンドラー向けの XABORT 引数

RTM ベースのトランザクション領域のアボートに XABORT 命令が使用されると、EAX レジスターを介してフォールバック・ハンドラーに命令オペランドが渡されます。この情報は、RTM 用の PEBS ベースのプロファイリング・ツールでも提供されます。プロファイリング・ツールはこの情報を使用してさまざまな XABORT ベースのトランザクション・アボートを分類できます。アボートステータスを定義することは、優れたフォールバック・ハンドラーを記述する上でも役立ちます。

次の表に、ここで使用するアボートステータスの定義を示します。

表 14-1 RTM アボートステータスの定義

アボートコード	説明
0xff	テスト時にロックがフリーでなかったことが原因の XABORT ベースのアボート (「14.3.4 ロック省略に RTM を使用するラッパーの例」を参照)
0xfe	省略されたロックの値がテストされたことが原因の XABORT ベースのアボート (「14.3.9 RTM ベースのライブラリーで省略されたロックの値を読み取る」を参照)
0xfd	入れ子の try-lock 内で発生した XABORT ベースのアボート (14.3.9 を参照)
0xfc: 0xf0	予約済み

**チューニングの推奨事項 28:** プロファイリング・ツールは RTM アボートコードを表示できなければなりません。

### 14.4.6 トランザクション・アボートのコールグラフ

プロファイリング・ツールは、パフォーマンス監視情報を収集する際に割り込みをかけます。この割り込みはトランザクション・アボートの原因になります。つまり、プロファイリング・ツールはトランザクション・アボートが発生した後にのみ情報を収集することが可能であり、トランザクション領域内で発生したスタック上の関数呼び出しは把握できず、トランザクション実行の開始時のコールグラフのみ見ることができるといことです。PEBS でトランザクション・アボートをサンプリングする場合、RIP フィールドにはアボート後の命令ポインターが、EventingIP フィールドにはアボート時のトランザクション領域内の命令ポインターが含まれます。すべてのサンプリングがトランザクション・アボートの原因となるため、非アボートイベントのサンプリングでも同じことが言えます。

アボートの種類に応じて、ReturnIP または EventingIP のいずれかをプロファイリングすると良いでしょう。プロファイリング・ツールによって収集されるスタック・コールグラフは常に ReturnIP と関連付けられています。この情報と EventingIP を組み合わせると、トランザクション領域内に関数呼び出しが含まれず、連続していないように見えることがあります (EventingIP は最下位レベルの呼び出し元と関連付けられていない可能性があります)。アボートの原因を理解するためトランザクション領域内の関数呼び出しに関する情報が必要な場合は、LBR (最後の分岐レコードの略、14.4.7 節「LBR とトランザクション・アボート」を参照) または SDE ソフトウェア・エミュレーション (14.4.8 節「インテル® SDE によるインテル® TSX ソフトウェアのプロファイリングとテスト」を参照) を使用できます。

**チューニングの推奨事項 29:** プロファイラーは ReturnIP と EventingIP を表示できなければなりません。

**チューニングの推奨事項 30:** スタック・コールグラフは常に ReturnIP に関連付けられており、EventingIP と一緒に見た場合、連続していないように見えることがあります。

**チューニングの推奨事項 31:** トランザクション領域内の関数呼び出しを確認するには LBR またはインテル® SDE を利用します。

### 14.4.7 LBR とトランザクション・アポート

LBR (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の 17.4 節を参照) は、トランザクション実行とアポートに関する情報を提供します。一般に LBR はインテル® TSX と互換性があります。通常のコールグラフが利用できない場合、LBR を使用することでトランザクション内の情報が得られます。lcall フィルターをコールグラフの代わりに使用できます。ただし、LBR コールグラフ・スタック (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の 17.8 節) はインテル® TSX と互換性がなく、完全な情報が得られないことがあります。

**チューニングの推奨事項 32:** プロファイリング・ハンドラーはアポート時に LBR をサンプリングし、その結果を報告できなければなりません。

### 14.4.8 インテル® SDE によるインテル® TSX ソフトウェアのプロファイリングとテスト

インテル® Software Development Emulator (インテル® SDE) ツール (<http://software.intel.com/en-us/articles/intel-software-development-emulator> (英語)) は、ハードウェアに実装される前に新しい命令セット拡張をソフトウェア開発に利用できます。このツールは、新しい命令を利用するソフトウェアの広範なテスト、デバッグ、解析にも役立ちます。

インテル® SDE の各種機能によって、インテル® TSX 命令を使用するプログラムの機能テスト、プロファイル、デバッグが行えます。このツールは一般的なトランザクション・アポートの詳細な情報と、ハードウェアで直接利用できない追加のプロファイリング機能をもたらします。エミュレーションによる非常に大きなオーバーヘッドが発生するため、このツールでランタイムおよび絶対的なパフォーマンス特性を得るべきではありません。

14.4.4 節「プロファイリング・ツールを利用してアポートを分類する」で述べたとおり、アポートの原因がデータ競合やリソースの制限によるものでない限り、ハードウェアはアポートの原因となった命令の正確なアドレスを報告します。インテル® SDE はそのような命令の正確な命令アドレスと命令に関する追加情報を提供します。また、アプリケーションのソースコード位置、ソースファイル名、行番号、コールスタック、命令が操作したデータアドレスの情報も提供します。さらに、犠牲となったトランザクション (競合でアポートされた) 向けに、競合するメモリアクセスが行われたソースコードの位置も出力できます。

次のオプションを指定することで、これらの情報を得られます。

```
-tsx -hle_enabled 1 -rtm-mode full -tsx_stats 1 -tsx_stats_call_stack 1
```

フォールバック・ハンドラーは EAX レジスターからアポートの原因を判断します。インテル® SDE ツールにエミュレーター・パラメーターとして特定の EAX レジスター値を渡すと、トランザクション・アポートを強制できます。開発者はさまざまな EAX 値でフォールバック・ハンドラーをテストできます。このモードでは、すべての RTM ベースのトランザクション実行は、パラメーターとして渡された EAX レジスター値で直ちにアポートします。これは、未解決のページフォルトや同様の操作が原因でトランザクション実行がアポートするケース (EAX = 0) の機能テストに有効です。

次のオプションを指定することで、これらの情報を得られます。

```
-tsx -rtm-mode abort -rtm_abort_reason EAX
```

インテル® SDE は、容量アボートのデバッグに有効な命令とメモリアクセスのログ取得機能を備えています。インテル® SDE のログデータから、容量オーバーフローを引き起こしている不均等 (non-uniform) キャッシュセットがあるか調査するため、キャッシュセットの過密度を診断できます。この洗練されたログデータは、アボートの原因を診断する際に利用できます。ログ取得機能は、以下のオプションを指定して有効にできます。

```
-tsx_debug_log 3 -tsx_log_inst 1 -tsx_log_file 1
```

さらに、インテル® SDE は、トランザクション内部の機能的なデバッグを行うため標準デバッガー (gdb や Microsoft\* Visual Studio\*) とともに使用できます。

### 14.4.9 HLE 固有のパフォーマンス監視イベント

インテル® TSX のパフォーマンス・イベントには HLE 固有のトランザクション・アボート条件が含まれています。これらのイベントは、14.2.4.4 節「HLE 固有のトランザクション・アボート」に示す原因のアボートを追跡します。多くの場合、これらのアボートは同期ライブラリーの実装の問題により発生します。インテル® TSX 対応の同期ライブラリーでは、これらのイベントを測定してその値が無視できるくらいになるまでライブラリーを改善すると良いでしょう。

TX\_MEM.ABORT\_HLE\_STORE\_TO\_ELIDED\_LOCK は、XRELEASE プリフィクスを持たないストア操作が、省略バッファーで省略されたロックの操作を行ったために発生したトランザクション・アボートの数をカウントします。これは多くの場合、ロックの解放命令に XRELEASE プリフィクスがないことが原因です。

TX\_MEM.ABORT\_ELISION\_BUFFER\_NOT\_EMPTY は、トランザクション実行をコミットする XRELEASE プリフィクスが付加されたロックの解放命令が、省略されたロックを持つ省略バッファーを見つけたために発生したトランザクション・アボートの数をカウントします。これは多くの場合、省略されなかった (つまり、省略バッファーにない) ロックに対して XRELEASE を実行するコードシーケンスで発生します。

TX\_MEM.ABORT\_HLE\_ELISION\_BUFFER\_MISMATCH は、XRELEASE ロックが省略バッファーのアドレスと値の条件を満たさないために発生したトランザクション・アボートの数をカウントします。これは、例えば XRELEASE 操作によって書き込まれる値が、同じロックに対する XACQUIRE 操作で読み取られた値と異なる場合に発生します。

TX\_MEM.ABORT\_HLE\_ELISION\_UNSUPPORTED\_ALIGNMENT は、トランザクション領域の読み取りが省略バッファーのロックにアクセスしたが、読み取れなかったために発生したトランザクション・アボートの数をカウントします。これは通常、アクセスが適切にアライメントされていないか、アクセスが部分的にオーバーラップしているか、あるいは読み取り操作のリニアアドレスが省略されたロックと異なるが物理アドレスは同じ場合に発生します。これらのイベントの発生は非常にまれです。

### 14.4.10 インテル® TSX の有用なメトリックを計算する

ここでは、パフォーマンス・イベントを使って有用な指標を計算する式を示します。イベントのカウントをそのまま利用できることもありますが、場合によってはカウンターのデータを基に計算が必要になります。

次の式は、HLE または RTM トランザクション実行が開始された回数を計算します。ここでは、すべての入れ子の領域を 1 つの領域にまとめています。

```
#HLE Regions Started: HLE_RETIRE.COMMIT + HLE_RETIRE.ABORTED
#RTM Regions Started: RTM_RETIRE.COMMIT + RTM_RETIRE.ABORTED
```

次の式は、アボートした HLE または RTM トランザクション実行の比率を計算します。

```
%AbortedHLE = 100.0 * (HLE_RETIRE.ABORTED/HLE_RETIRE.START)
%AbortedRTM = 100.0 * (RTM_RETIRE.ABORTED/RTM_RETIRE.START)
```



次の式は、トランザクション領域で費やされたサイクル数の平均を計算します (CyclesInTX の計算については 14.4.1 節「トランザクションの成功を測定する」を参照)。

$$\begin{aligned} \text{AvgCyclesInHLE} &= \text{CyclesInTX} / \text{HLE\_RETIRED\_START} \\ \text{AvgCyclesInRTM} &= \text{CyclesInTX} / \text{RTM\_RETIRED\_START} \\ \text{AvgCyclesInTX} &= \text{CyclesInTX} / (\text{HLE\_RETIRED\_START} + \text{RTM\_RETIRED\_START}) \end{aligned}$$

次の式は、データ競合によりアボートした HLE または RTM トランザクション実行の比率を計算します。

$$\begin{aligned} \% \text{AbortedHLEDataConflict} &= \text{TX\_MEM.ABORT\_CONFLICT} / \text{HLE\_RETIRED\_START}; \\ \% \text{AbortedRTMDataConflict} &= \text{TX\_MEM.ABORT\_CONFLICT} / \text{RTM\_RETIRED\_START}; \\ \% \text{AbortedTXDataConflict} &= \text{TX\_MEM.ABORT\_CONFLICT} / (\text{HLE\_RETIRED\_START} + \text{RTM\_RETIRED\_START}); \end{aligned}$$

次の式は、トランザクション・ストアのリソースの制限によりアボートした HLE または RTM トランザクション実行の数を計算します。

$$\% \text{AbortedTXStoreResource} = \text{TX\_MEM.ABORT\_CAPACITY\_WRITE}$$

Broadwell<sup>+</sup> と Skylake<sup>+</sup> マイクロアーキテクチャーをベースとするプロセッサでは、“TX\_MEM.ABORT\_CAPACITY\_WRITE” イベントは、読み取りまたは書き込みによるアボートをカウントする TX\_MEM.ABORT\_CAPACITY で置き換えられます。

次の式は、リソースの制限によりアボートした HLE または RTM トランザクション実行の合計数を計算します。L1 データキャッシュから追い出されたトランザクション読み取りは、直ちにアボートにならない可能性があるため区別されず。

$$\begin{aligned} \% \text{AbortedHLEResource} &= \text{HLE\_RETIRED.ABORTED\_MISC1} - \text{TX\_MEM.ABORT\_CONFLICT} \\ \% \text{AbortedRTMResource} &= \text{RTM\_RETIRED.ABORTED\_MISC1} - \text{TX\_MEM.ABORT\_CONFLICT} \\ \% \text{AbortedTXResource} &= (\text{HLE\_RETIRED.ABORTED\_MISC1} + \text{RTM\_RETIRED.ABORTED\_MISC1}) - \text{TX\_MEM.ABORT\_CONFLICT} \end{aligned}$$

HLE\_RETIRED.ABORTED\_MISC1 は、14.4.9 節「HLE 固有のパフォーマンス監視イベント」で示したいくつかのイベントの影響を受けることがあります。正確な結果を得るためには、まずこれらを最小限に抑えるようにロック・ライブラリーをチューニングする必要があります。

HLE\_RETIRED.ABORTED\_MISC1 は、HLE\_RETIRED.ABORTED\_MIEM としても知られています。同様に、RTM\_RETIRED.ABORTED\_MISC1 は RTM\_RETIRED.ABORTED\_MEM とも呼ばれます。

## 14.5 パフォーマンスのガイドライン

第 4 世代インテル® Core™ プロセッサはインテル® TSX をサポートする最初のプロセッサです。トランザクション実行には実装固有のオーバーヘッドが伴います。パフォーマンスは、将来のマイクロアーキテクチャーで改善される可能性があります。インテル® TSX の初期実装はアプリケーションのクリティカル・セクションにおける一般的な用途を想定しています。そのため、このようなオーバーヘッドは相殺され、通常アプリケーション・レベルのパフォーマンスには影響しません。

しかし、考慮すべきいくつかのガイドラインがあります。

**チューニングの推奨事項 33:** インテル® TSX はクリティカル・セクション向けに設計されているため、XBEGIN/XEND 命令と XACQUIRE/XRELEASE プリフィックスのレイテンシーは、LOCK プリフィックス命令のレイテンシーと一致するように意図されています。これらの命令のレイテンシーは通常のロード操作とは異なることに注意してください。

トランザクション領域の実行には実装固有のオーバーヘッドがあります。そのほとんどは固定コストで、残りはさまざまな動的コンポーネントによるものです。このオーバーヘッドはクリティカル・セクションのサイズやメモリー使用量とはほとんど関係なく、通常マイクロアーキテクチャーのアウトオブオーダー実行によって相殺されます。しかし、第 4 世代インテル® Core™ プロセッサ実装では、特定のシーケンスでこのオーバーヘッドが大きくなる可能性があります。特にクリティカル・セクションが非常に小さく、タイトなループ内にある場合（例えば、マイクロベンチマークで行われる処理など）、オーバーヘッドが大きくなります。実際のアプリケーションでは、通常このような動作は見られません。

このオーバーヘッドは大きなクリティカル・セクションでは相殺されますが、非常に小さなクリティカル・セクションでは相殺されません。オーバーヘッドを減らす簡単なアプローチの 1 つは、クリティカル・セクションの早期にトランザクション・キャッシュラインへアクセスすることです。コミットのオーバーヘッドは、Broadwell<sup>†</sup> マイクロアーキテクチャーベースのプロセッサでは軽減されています。

## 14.6 デバッグのガイドライン

インテル® TSX を使用するロック省略の実装はアプリケーションのセマンティクスを変更しません。つまり、アボートされたトランザクション実行中に更新されたすべてのアーキテクチャー・ステートは、ハードウェアによって自動的に破棄されます。アプリケーションにトランザクション実行でのみ実行される新しいコードパスを追加する際は注意が必要です（14.2.5 節「トランザクション実行専用のコードパスの使用」を参照）。

ただしロックの省略では、データ競合が起こった場合にのみスレッド間の通信が発生するため、スレッド間の実行タイミングが変わります。そのためロックが通常よりも速く行われるように見えることがあります。このタイミングの違いはアプリケーションに潜在的な問題をもたらす可能性があります。この潜在的な問題はインテル® TSX 固有のものでなく、新しい世代のすべてのハードウェアで見られます。

コードのインストルメンテーションは、マルチスレッド・ソフトウェアのデバッグでよく使用される手法です。タイミング関連の問題をデバッグする場合と同様に、コードのインストルメンテーションを行う際は、タイミングを大きく変えたり、不要なアボートを引き起こさないように注意する必要があります。スレッドごとにバッファを利用し、実行をトレースし特定のイベントを記録できます。タイムスタンプの取得には RDTSC 命令を使用できます。バッファの出力はクリティカル・セクション外で行うべきです。

トランザクション・アボートは、トランザクション領域内で更新されたメモリー状態をすべて破棄します。この情報はインストルメンテーションでなければトレースできません。トランザクション領域内の問題は、プロファイリング・ツールではトランザクション・アボートとして検出され、LBR 情報から制御フローを再構成できます。インテル® プロセッサ・トレースをサポートするプロセッサでは、トランザクション内部の制御フローを完全に追跡して記録することを可能にするトレースログを使用できます。このトレースには、トランザクションの開始、コミット、およびアボートを示すマーカーが含まれます。

通常の `assert()` 関数はトランザクション・アボートになり、その出力情報はトランザクション領域外では利用できません。RTM 命令を使用することで `assert` 関数は、トランザクション実行を終了し、その影響を可視化して、`assert` 関数でプログラムを終了することができます。次に例を示します。

```
assert(x) => if (!x) { while (_xtest()) _xend(); assert(0); }
```

## 14.7 インテル® TSX 用の一般的な組み込み関数

新しいアセンブラー (GNU\* binutils 2.23、Microsoft\* Visual Studio\* 2012) はインテル® TSX 命令をサポートしています。以前のツールチェーンではインテル® TSX 命令をバイト値として表現します。

### 14.7.1 RTM C 組み込み関数

新しい C/C++ コンパイラー (gcc 4.8、Microsoft\* Visual Studio\* 2012、インテル® C++ コンパイラー 17.0) は、`immintrin.h` ヘッダーファイルで RTM 組み込み関数を定義しています。RTM は新しい命令セットであり、CPUID 命

令で RTM 機能フラグを確認してから使用するべきです (『Intel® Architecture Instruction Set Extensions Programming Reference』の第 8 章を参照)。

## `_xbegin()`

`_xbegin()` はトランザクション領域を開始して、トランザクション領域に入ると `_XBEGIN_STARTED` を返し、そうでない場合はアボートコードを返します。`_xbegin()` の戻り値が `_XBEGIN_STARTED` (0 でない値) であることを確認することが重要です。ゼロはアボートコードです。値が `_XBEGIN_STARTED` でない場合、リターンコードには、`_xabort()` によって渡される各種ステータスビットとオプションの 8 ビット定数が含まれます。

以下に、有効なステータスビットを示します。

- Y `_XABORT_EXPLICIT`: `_xabort()` によって発生したアボート。`_XABORT_CODE(status)` には、`_xabort()` へ渡された値が含まれます。
- Y `_XABORT_RETRY`: このビットが設定されている場合は、再試行によりトランザクション領域をコミットできる可能性があります。設定されていないければ、再試行しても成功する確率が低いままです。
- Y `_XABORT_CAPACITY`: 容量のオーバーフローによるアボートです。
- Y `_XABORT_DEBUG`: デバッグトラップによるアボートです。
- Y `_XABORT_NESTED`: 入れ子のトランザクションで発生したアボートです。

## `_xend()`

`_xend()` はトランザクションをコミットします。

## `_xtest()`

`_xtest()` は、コードが現在トランザクション実行中の場合は真を返します。HLE でも利用できます。

## `_xabort()`

`_xabort(constant)` は現在のトランザクションをアボートします。`constant` は 8 ビットの定数でなければなりません。この定数は `_xbegin()` によって返されるステータスコードに含まれており、`_XABORT_EXPLICIT` フラグが設定されている場合、`_XABORT_CODE()` でアクセスできます。推奨される利用法については 14.4.5 節を参照してください。

gcc 4.8 以降では、`-mrtm` コンパイラー・オプションを指定してこれらの組み込み関数を有効にする必要があります。

### 14.7.1.1 古い gcc\* 互換コンパイラーによる RTM 組み込み関数のエミュレート

`immintrin.h` で RTM 組み込み関数をサポートしていない古い gcc 互換コンパイラーでは、例 14-11 に示す等価なインライン・アセンブラーを利用できます。

例 14-11 古い gcc 互換コンパイラーによる RTM 組み込み関数のエミュレート

```

/* immintrin.h でこのインターフェイスをサポートしている新しいツールでは不要 */
#define _XBEGIN_STARTED (~0u)
#define _XABORT_EXPLICIT (1 << 0)
#define _XABORT_RETRY (1 << 1)
#define _XABORT_CONFLICT (1 << 2)
#define _XABORT_CAPACITY (1 << 3)
#define _XABORT_DEBUG (1 << 4)
#define _XABORT_NESTED (1 << 5)
#define _XABORT_CODE(x) (((x) >> 24) & 0xff)
#define __force_inline __attribute__((__always_inline__)) inline

static __force_inline int _xbegin(void)
{
    int ret = _XBEGIN_STARTED;
    asm volatile(".byte 0xc7,0xf8 ; .long 0" : "+a" (ret) :: "memory");
    return ret;
}

static __force_inline void _xend(void)
{
    asm volatile(".byte 0x0f,0x01,0xd5" ::: "memory");
}

static __force_inline void _xabort(const unsigned int status)
{
    asm volatile(".byte 0xc6,0xf8,%P0" :: "i" (status) : "memory");
}

static __force_inline int _xtest(void)
{
    unsigned char out;
    asm volatile(".byte 0x0f,0x01,0xd6 ; setnz %0" : "=r" (out) :: "memory");
    return out;
}

```

14.7.2 gcc\* およびその他の Linux\* 互換コンパイラーの HLE 組み込み関数

Linux\* および互換システムでは、HLE は gcc 4.8 および古い形式の C11 のアトミック・プリミティブ拡張として実装されています。HLE XACQUIRE を使用するにはメモリーモデル引数に \_\_ATOMIC\_HLE\_ACQUIRE フラグを設定し、HLE XRELEASE を使用するには \_\_ATOMIC\_HLE\_RELEASE フラグを設定します。

メモリーモデルは、\_\_ATOMIC\_HLE\_ACQUIRE では \_\_ATOMIC\_ACQUIRE 以上、\_\_ATOMIC\_HLE\_RELEASE では \_\_ATOMIC\_RELEASE 以上でなければなりません。失敗メモリーモデルと成功メモリーモデルを含む操作 (\_\_atomic\_compare\_exchange\_n など) では、HLE フラグは成功メモリーモデルでのみサポートされます。

HLE は、IA アトミック命令に直接変換可能なアトミック操作でのみサポートされます。次の場合はサポートされません。

- ÿ 32 ビットのターゲット上の 8 バイト値
- ÿ 16 バイト値
- ÿ 加算/減算を除く、結果にアクセスするフェッチ命令または命令フェッチ
- ÿ \_\_atomic\_store と \_\_atomic\_clear は、\_\_ATOMIC\_HLE\_RELEASE のみサポート

### 14.7.2.1 gcc 4.8 による HLE 組み込み関数の生成

gcc 4.8 のいくつかのバージョンでは、コンパイラーの不具合により、アトミック組み込み関数を用いて HLE ヒントを生成するには最適化レベル -O2 以上を指定しなければなりません。

### 14.7.2.2 C++11 atomic のサポート

gcc 4.8 は C++11 の <atomic> ヘッダーをサポートしています。このヘッダーで定義されているメモリーモデルは、C アトミック・インターフェイスに似た HLE フラグで拡張されています。2 つの新しいフラグ `__memory_order_hle_acquire` と `__memory_order_hle_release` が定義されています。C アトミック組み込み関数の制限が適用されます。

例 14-12 に C++ を使用した HLE 組み込み関数の例を示します。

例 14-12 HLE 組み込み関数の C++ の例

```
#include <atomic>
#include <immintrin.h>
using namespace std;
atomic_flag lock;
for (;;) {
    if (!lock.test_and_test(memory_order_acquire|__memory_order_hle_acquire) {
        // HLE によるロックの省略を使用するクリティカル・セクション
        lock.clear(memory_order_release|__memory_order_hle_release);
        break;
    } else {
        // ロックを取得できなかったため待機して再試行する
        while (lock.load())
            _mm_pause(); // ロックがビジーなためトランザクション領域をアボートする
    }
}
```

### 14.7.2.3 古い gcc\* 互換コンパイラーによる HLE 組み込み関数のエミュレート

これらの組み込み関数をサポートしていない古いコンパイラーではインライン・アセンブリーを利用できます。例 14-13 に、`__atomic_exchange_n(&lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE)` をエミュレートする例を示します。

表 14-13 古い GCC コンパイラーでの HLE 組み込み関数のエミュレーション

```
#define XACQUIRE ".byte 0xf2; " /* XACQUIRE をサポートしない古いアセンブラー向け*/
#define XRELEASE ".byte 0xf3; "

static inline int hle_acquire_xchg(int *lock, int val)
{
    asm volatile(XACQUIRE "xchg %0,%1" : "+r" (val), "+m" (*lock) :: "memory");
    return val;
}

static void hle_release_store(int *lock, int val)
{
    asm volatile(XRELEASE "mov %0,%1" : "r" (val), "+m" (*lock) :: "memory");
}
}
```

### 14.7.3 Windows\* C/C++ コンパイラーの HLE 組み込み関数

Windows\* C/C++ コンパイラー (Microsoft Visual Studio 2012 およびインテル® C++ コンパイラー 17.0) は、HLE プレフィクスを持つ固有の atomic 組み込み関数を提供しています。例 14-14 を参照してください。

例 14-14 インテルと Microsoft コンパイラーによる HLE 組み込み関数のサポート

アトミックな比較-交換操作:

```
long _InterlockedCompareExchange_HLEAcquire(long volatile *Destination, long Exchange, long
Comparand);
__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *Destination, __int64
Exchange, __int64 Comparand);
void * _InterlockedCompareExchangePointer_HLEAcquire(void * volatile *Destination, void *
Exchange, void *Comparand);
long _InterlockedCompareExchange_HLERelease(long volatile *Destination, long Exchange, long
Comparand);
__int64 _InterlockedCompareExchange64_HLERelease(__int64 volatile *Destination, __int64
Exchange, __int64 Comparand);
void * _InterlockedCompareExchangePointer_HLERelease(void * volatile *Destination,
void * Exchange, void *Comparand);
```

アトミックな加算:

```
long _InterlockedExchangeAdd_HLEAcquire(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *Addend, __int64 Value);
long _InterlockedExchangeAdd_HLERelease(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *Addend, __int64 Value);
```

HLE プリフィクス付きのストア組み込み関数:

```
void _Store_HLERelease(long volatile *Destination, long Value);
void _Store64_HLERelease(__int64 volatile *Destination, __int64 Value);
void _StorePointer_HLERelease(void * volatile *Destination, void * Value);
```

組み込み関数の詳細については、コンパイラーのドキュメントを参照してください。

## 第 16 章

# Goldmont<sup>†</sup> および Silvermont<sup>†</sup> マイクロアーキテクチャー向けのソフトウェア最適化

次世代の Intel Atom<sup>®</sup> プロセッサ・ファミリーは Goldmont<sup>†</sup> マイクロアーキテクチャーをベースにしています。Goldmont<sup>†</sup> マイクロアーキテクチャーは、Silvermont<sup>†</sup> マイクロアーキテクチャーを引き継いでいます。Silvermont<sup>†</sup> マイクロアーキテクチャー・ベースの Intel Atom<sup>®</sup> プロセッサは、タブレット、携帯電話、そして PC からマイクロサーバーまで、幅広いデバイスで利用されています。この章では、現在および最近の世代の Intel Atom<sup>®</sup> プロセッサ向けソフトウェアのコーディング手法を説明しています。この章におけるソフトウェア最適化の推奨事項は、Silvermont<sup>†</sup> と Airmont<sup>†</sup> マイクロアーキテクチャーから始まる最近の世代の Intel Atom<sup>®</sup> プロセッサのマイクロアーキテクチャーに注目します。これらの推奨事項は、第 3 章「一般的な最適化のガイドライン」で説明する x86 向けの一般的なコーディングの推奨に加えて考慮されるべきです。Intel Atom<sup>®</sup> プロセッサの最近の世代のマイクロアーキテクチャー向けの CPUID シグネチャー DisplayFamily\_DisplayModel を確認する方法は、16.3 節をご覧ください。Intel<sup>®</sup> プロセッサ・ファミリー全体のマイクロアーキテクチャーに関する DisplayFamily\_DisplayModel シグネチャーの詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 4』の第 2 章の表 2-1 で確認できます。

## 16.1 最近の Intel Atom<sup>®</sup> プロセッサ世代のマイクロアーキテクチャー

### 16.1.1 Goldmont<sup>†</sup> マイクロアーキテクチャー

Goldmont<sup>†</sup> マイクロアーキテクチャーは、Silvermont<sup>†</sup> マイクロアーキテクチャー (16.1.2 節を参照) の成功を基に、次の拡張を提供します。

- Y 3 ワイド・スーパーカラー・パイプラインのアウトオブオーダー実行エンジン。
  - デコーダーはサイクルごとに 3 命令をデコード可能。
  - マイクロコード・シーケンサーは、アロケーションのためサイクルごとに 3 つの  $\mu\text{op}$  をリザーベーション・ステーションへ送出可能。
  - リタイアメントは、サイクルあたり 3 ピークレートをサポート。
- Y 命令デコーダーからフェッチ・パイプラインを分離することで、分岐予測を強化。
- Y 大きなアウトオブオーダー実行ウィンドウとバッファーにより、整数、FP/SIMD、およびメモリー命令タイプに渡り、より深いアウトオブオーダー実行が可能となります。
- Y 完全なアウトオブオーダー・メモリー実行とディスアンビゲーション。Goldmont<sup>†</sup> マイクロアーキテクチャーは、サイクルごとに 1 つのロードと 1 つのストアを実行できます (つまり 2 つの操作)。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、サイクルごとに 1 つのロードまたは 1 つのストアを実行できました。メモリー実行パイプラインはまた、4KB ページで 512 エントリーに拡張された第 2 レベルの TLB を含みます。
- Y Goldmont<sup>†</sup> マイクロアーキテクチャーの整数実行クラスターは、3 つのパイプラインを提供し、サイクルごとに最大 3 つの簡単な ALU 操作を実行できます。
- Y SIMD 整数と浮動小数点命令は、128 ビット幅のエンジンで実行されます。多くの命令のスループットとレイテンシーが改善されています。例えば、PSHUFQ 命令は 1 サイクルのスループット (Silvermont<sup>†</sup> マイクロアーキテクチャーでは 5 サイクルでした) で、その他多くの SIMD 命令は倍のスループットを提供します。詳細は表 16-14 をご覧ください。
- Y Goldmont<sup>†</sup> マイクロアーキテクチャーでは、暗号化/復号 (AES) とキャリーなしの乗算 (PCLMULQDQ) を加速する命令のスループットとレイテンシーが、かなり改善されました。
- Y Goldmont<sup>†</sup> マイクロアーキテクチャーは、ハードウェアによって加速された安全なハッシュ・アルゴリズムをサポートする新しい命令 SHA1 と SHA256 を提供します。
- Y Goldmont<sup>†</sup> マイクロアーキテクチャーはまた、NIST SP800-90C 標準に準拠する乱数生成のため RDSEED 命令をサポートします。
- Y 電力効率を高めるため、PAUSE 命令のレイテンシーが最適化されました。

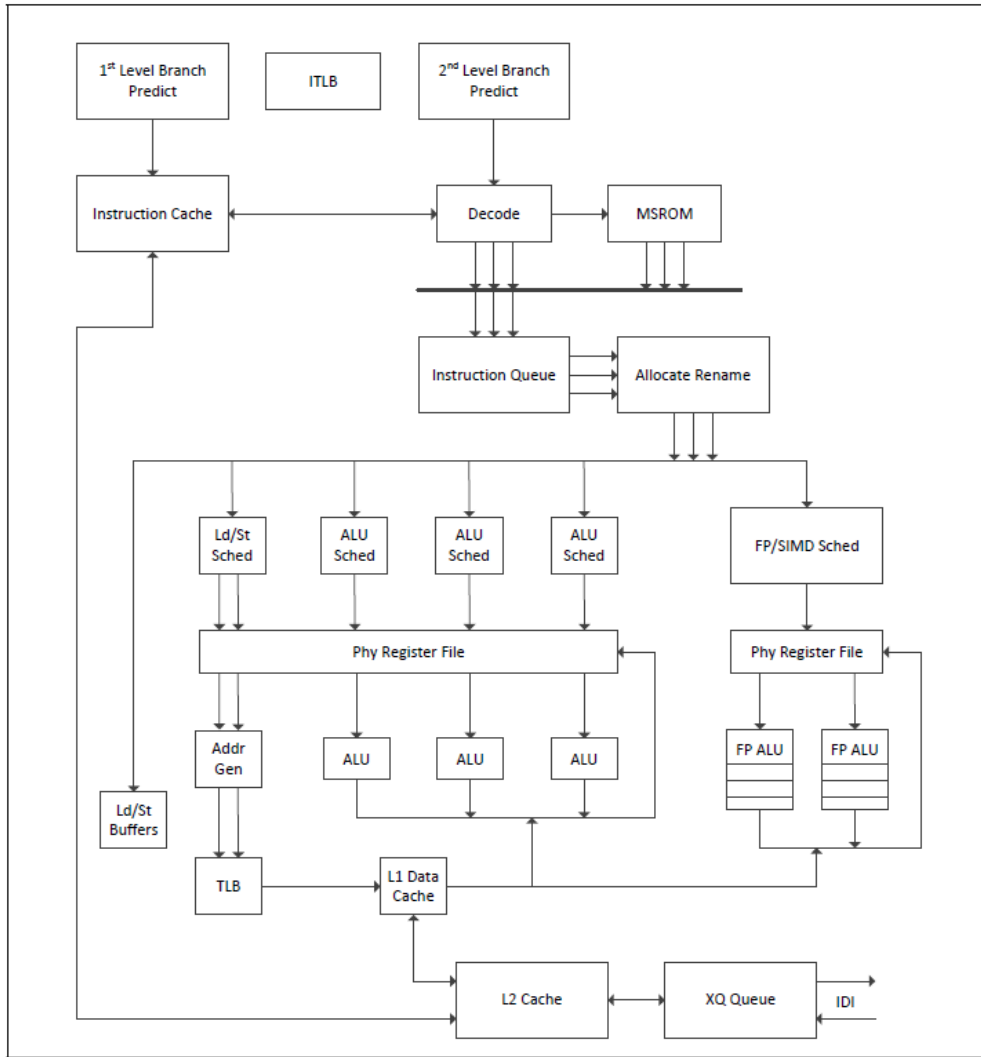


図 16-1 Goldmont<sup>+</sup> マイクロアーキテクチャーの CPU コア・パイプラインの機能

Goldmont<sup>+</sup> マイクロアーキテクチャーのフロントエンド・クラスター (FEC) では、Silvermont<sup>+</sup> マイクロアーキテクチャーの FEC に対し多くの拡張が行われています。表 16-1 にこれらの拡張をまとめています。

表 16-1 フロントエンド・クラスター機能の比較

Feature	Goldmont Microarchitecture	Silvermont Microarchitecture
Number of Decoders	3	2
Max Throughput of Decoders	20 Bytes per cycle	16 Bytes per cycle
Fetch and Icache Pipeline	Decoupled	Coupled
ITLB	48 entries, large page support	48 entries
Branch Mispredict Penalty	12 cycles	10 cycles
L2 Predecode Cache	16K	NA

FEC は、アロケーション、リネーミングおよびリタイアメント (ARR) クラスターを介して OOO 実行エンジンに接続されています。μop のスケジューリングは、異なるクラスター (IEC、FPC、MEC) にまたがって分散されタリザベーション・ステーションによって扱われます。それぞれのクラスターは、ARR から複数の μop を受け取るため固有のリザベーション・ステーションを保持しています。表 16-2 は、Goldmont<sup>+</sup> マイクロアーキテクチャーと Silvermont<sup>+</sup> マイクロアーキテクチャーのアウトオブオーダーの特徴を比較したものです。



表 16-2  $\mu\text{op}$  をスケジューリングする際の分散リザーベーション・ステーションの比較

Cluster	Goldmont Microarchitecture	Silvermont Microarchitecture
IEC Reservation	3x distributed for each port	2x distributed for each port
	Out-of-order within each IEC RSV and between IEC, across FPC, MEC	Out-of-order within each IEC RSV and between IEC, across FPC, MEC
FPC Reservation	1x unified to ports 0, 1	2x distributed for each port
	Out-of-order within FPC RSV and across IEC, MEC	In order within each FPC RSV; out-of-order between FPC, across IEC, MEC
MEC Reservation	1x unified to ports 0, 1	1x to port 0
	Out-of-order within MEC RSV and across IEC, FPC	In order within each MEC RSV; out-of-order across IEC, FPC

メモリーを参照して整数/FP リソースを必要とする命令は、メモリー  $\mu\text{op}$  が MEC クラスタへ送られ、整数/FP  $\mu\text{op}$  が IEC/FPC クラスタへ送られます。そしてリソースが利用可能になると、表 16-2 で示すヒューリスティックに従ってアウトオブオーダー実行を開始できます。表 16-3 は、それぞれのクラスタに対するポートと実行ユニットのマッピングを示します。

表 16-3 Goldmont<sup>†</sup> マイクロアーキテクチャーの機能ユニットの割り当て

Cluster	Port 0	Port 1	Port 2
IEC	ALU0, Shift/Rotate, LEA with no index, F2I, converts/cmp, store_data	ALU1, Bit processing, JEU, IMUL, IDIV, POPCNT, CRC32, LEA, I2F, store_data	ALU2, LEA <sup>1</sup> , I2F, flag_merge
FPC	SIMD ALU, SIMD shift/Shuffle, SIMD mul, STTNI/AESNI/PCLMULQDQ/SHA; FP_mul, Converts, F2I convert	SIMD ALU, SIMD shuffle, FP_add, F2I compare	
MEC	Load_addr	Store_addr	

**注意:**

1. インデックスなしの LEA は、ポート 0、1 もしくは 2 で実行できます。有効なインデックスとディスプレースメントを持つ LEA は複数のマイクロオペレーション ( $\mu\text{op}$ ) に分けられ、ポート 1 と 2 の両方を使用します。有効なインデックスを持つ LEA はポート 1 で実行されます。

MEC は固有の MEC RSV を保持しており、ポート 0 と 1 を経由するすべてのロードとストアのスケジューリングを行います。ロードとストア命令は、インオーダーまたはアウトオブオーダーでアドレス生成フェーズを通過します。アウトオブオーダーでアドレス生成のスケジューリングが可能である場合、メモリー実行パイプラインはロードバッファとストアバッファを使用してアドレス生成パイプラインから分離されます。

アウトオブオーダー実行では、ロードが未知のストアに先行することができるため、メモリー順序の問題とパイプラインフラッシュを引き起こす可能性があります。Goldmont<sup>†</sup> マイクロアーキテクチャーでサポートされるメモリー・ディスアンビゲーション (一義化) は、ロード実行の潜在的な問題を追跡して最小化します。

問題となったメモリー操作 ( $\mu\text{TLB}$  ミスや利用できないリソースなど) は、再実行のためロードやストアバッファに戻されます。後続の命令をすべてストールする代わりに、(問題が発生していない) より新しい命令の実行を継続できます。問題が解決されると、問題を引き起こした命令はロード/ストアバッファから再発行 (状況によっては、リタイアメントで再発行) されます。ロードミスが、データキャッシュが非ブロッキングであることによる問題と考えられる場合、ライトコンバイン・バッファ (WCB) を使用して複数の未処理のミスを吸収することができます。

表 16-4 MEC リソースの比較

MEC Resource	Goldmont Microarchitecture	Silvermont Microarchitecture
L1 Data Cache	24KB	24 KB
uTLB	32 entries	32 entries
DTLB (4KB page)	512 entries	128 entries
DTLB (2M/4M page)	32 entries	16 entries
Load-use Latency	3 cycles	3 cycles
Pipeline	1x load + 1x store	1x share by load/store
AGEN	Out-of-order	In order
WCBs	8	8
Addressing	39-bit physical, 48-bit linear	36-bit physical, 48-bit linear

### 16.1.2 Silvermont<sup>†</sup> マイクロアーキテクチャー

Intel Atom<sup>®</sup> プロセッサ E3000 と C2000 シリーズは、Silvermont<sup>†</sup> マイクロアーキテクチャーをベースにしています。Silvermont<sup>†</sup> マイクロアーキテクチャーは、タブレット、携帯電話、そして PC からマイクロサーバーまで、幅広いコンピューター・デバイスで利用できます。インテル® 64 アーキテクチャーと IA-32 アーキテクチャーのサポートに加えて、Silvermont<sup>†</sup> マイクロアーキテクチャーでは主に次の点が拡張されています。

- Y 整数命令のアウトオブオーダー実行、および非整数命令とメモリー命令間の実行順序を分離しています。対照的に、45nm と 32nm の Intel Atom<sup>®</sup> マイクロアーキテクチャー (付録 D を参照) では、インオーダー実行が厳守され、命令レベルの並列性が制限されていました。
- Y 非ブロッキング命令における複数の未処理ミスの許容 (8 回まで)。前世代のプロセッサでは、1 つのメモリー命令で問題が発生すると (例えば、キャッシュミスなど)、その問題が解決されるまで後続のすべての命令がストールしましたが、新しいマイクロアーキテクチャーでは最大 8 つの未処理参照が許容されます。
- Y 2 コアのもジュールシステム設計。フロントサイド・バスの代わりにポイントツポインのインターフェイスを使って、新しい内蔵メモリー・コントローラーに接続された L2 キャッシュを共有します。
- Y インテル® SSE4.1、インテル® SSE4.2、インテル® AES New Instructions (インテル® AES-NI)、PCLMULQDQ が追加されています。

図 16-2 に Silvermont<sup>†</sup> マイクロアーキテクチャーの基本パイプライン機能を示します。シングルスレッドのパフォーマンスを向上するため、メモリークラスターと実行クラスター的设计が大幅に見直されている一方、これまでと同様に、小さなフォームファクターで低消費電力を実現する取り組みが行われています。各パイプラインには、リザーベーション・ステーション (RSV) と呼ばれる専用のスケジューリング・キューがあります。浮動小数点命令とメモリー命令はそれぞれのキューからプログラム順にスケジュールされ、整数命令はそれぞれのキューからアウトオブオーダーでスケジュールされます。

これは、整数命令がインオーダー実行であった前世代とは対照的です。アウトオブオーダー・スケジューリングにより、これらの命令ではソースやリソースが利用できない場合に発生するストールを許容することができます。メモリー命令は、アドレスの生成 (AGEN) をインオーダーで行い、スケジューリング・キューからインオーダーでスケジュールしなければいけませんが、実行はアウトオブオーダーで行うことができます。

(SIMD 整数、SIMD 浮動小数点、x87 浮動小数点を含む) 非整数命令も、それぞれのスケジューリング・キューからプログラム順にスケジュールされますが、これらは個別のスケジューリング・キューなので、ほかのスケジューリング・キューにある命令とは切り離して実行することができます。

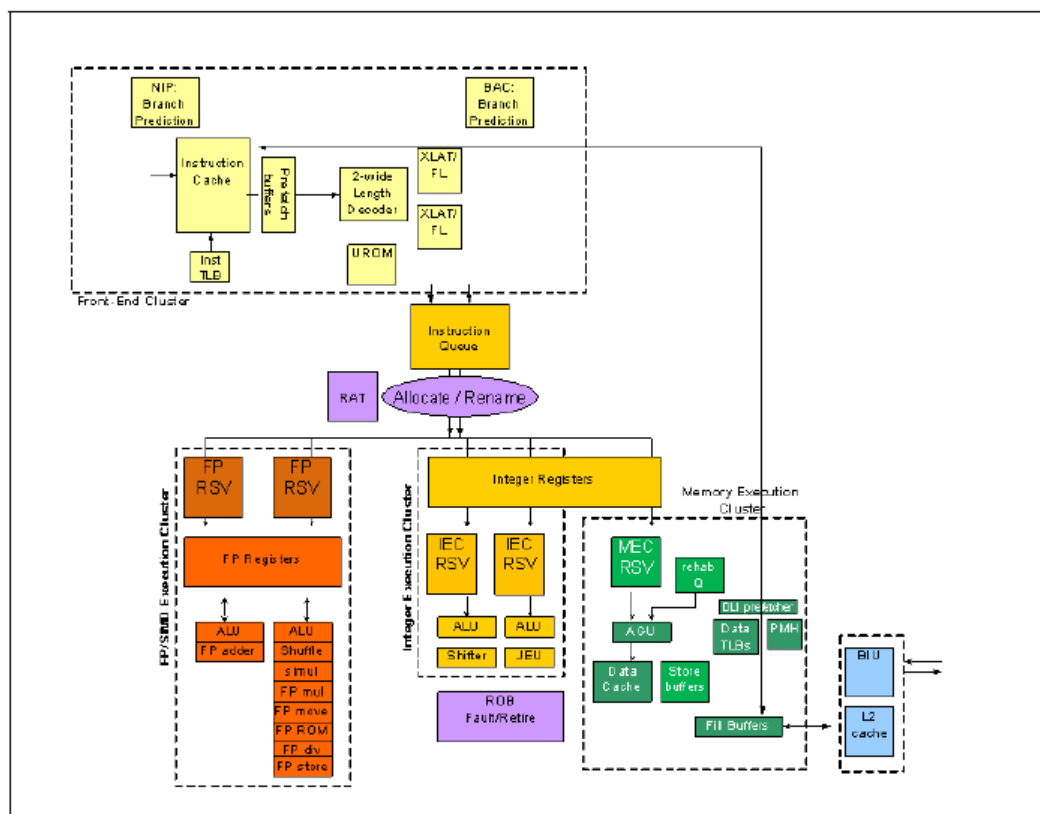


図 16-2 Silvermont+ マイクロアーキテクチャーのパイプライン

Silvermont+ マイクロアーキテクチャーは、アウトオブオーダー・スケジューリングにより、多様なフォームファクターの（例えば、携帯電話、タブレットからマイクロサーバーにわたる）プラットフォーム・パフォーマンスを最大限に引き出し、消費電力と面積コストを最小限に抑えるように設計されています（つまり、パフォーマンス/電力/コスト効率を最大化しています）。共有 L2 キャッシュを装備したマルチコア・アーキテクチャーを採用しているため、インテル® ハイパースレッディング・テクノロジーはサポートされません。クラスターレベルの機能については、この節の後半で説明します。

図 16-2 に薄黄色で示されているフロントエンド・クラスター (FEC) は、同時に 2 命令を処理できるデコード・パイプラインであり、消費電力が最適化されています。FEC はメモリーから命令をフェッチしデコードを行います。このとき、命令キャッシュからのプリデコード情報を利用することで、コストのかかる命令長の検出をデコード時に行わないようにしています。フロントエンドには分岐ターゲットバッファー (BTB) と高度な分岐予測ハードウェアがあります。

フロントエンドは、アロケーション、リネーミング、およびリタイアメント (ARR) クラスターを介して OOO 実行エンジンに接続されています (図 16-2 の紫色)。ARR は、FEC からマイクロオペレーション (μop) を受け取り、リソースチェックを行います。レジスター・エイリアス・テーブル (RAT) は、論理レジスターから物理レジスターへのリネームを行います。リオーダーバッファー (ROB) は、プログラム順に操作を並べ替えて実行 (リタイア) します。また、割り込み、例外、アシスト時には実行を停止して、マイクロコードに対するプログラム制御を実行します。

Silvermont+ マイクロアーキテクチャーは分散スケジューリングを採用しているため、リネーム処理後にマイクロオペレーション (μop) はさまざまなクラスター (IEC: 整数実行クラスター、MEC: メモリー実行クラスター、FPC: 浮動小数点クラスター) に送られ、スケジューリングされます (図 16-2 では FP RSV、IEC RSV、MEC RSV として示されています)。

FPC RSV と IEC RSV は 2 セット (各ポートに 1 つずつ) あり、MEC RSV は 1 セットあります。各 RSV は、ARR クラスタからサイクルごとに最大 2 マイクロオペレーション (μop) を受け取り、実行準備が整ったものから実行ユニットへディスパッチします。

分散型リザベーション・ステーションの概念をサポートするため、整数実行を要求する load-op (ロード-実行) 型や load-op-store (ロード-実行-ストア) 型のマクロ命令は、MEC RSV に送られるメモリー操作と、IEC RSV に送られる整数実行操作に分割する必要があります。IEC スケジューラーは、各 IEC RSV から実行準備が整っている最も古い命令を選択します。一方、MEC スケジューラーと FPC スケジューラーは、それぞれの RSV から最も古い命令を選択します。MEC クラスタと FPC クラスタはインオーダー・スケジューラーを採用していますが、FPC RSV の新しい命令は、別の FPC RSV や IEC RSV、MEC RSV にあるより古い命令よりも前に実行できます。

各実行ポートには固有の機能ユニットがあります。表 16-5 の Silvermont<sup>†</sup> マイクロアーキテクチャーの機能ユニットのポートへの割り当てを示します。図 16-2 では IEC がオレンジ、MEC が緑、FPC が赤で示されています。前世代の Intel Atom<sup>®</sup> マイクロアーキテクチャーと比べると、Silvermont<sup>†</sup> マイクロアーキテクチャーでは IEC に整数乗算ユニット (IMUL) が追加されています。

表 16-5 Silvermont<sup>†</sup> マイクロアーキテクチャーの機能ユニットの割り当て

	Port 0	Port 1
IEC	ALU0, Shift/Rotate Unit, LEA with no index	ALU1, Bit processing unit, Jump unit, IMUL, POPCNT, CRC32, LEA <sup>1</sup>
FPC	SIMD ALU, SIMD shift/Shuffle unit, SIMD FP mul/div/cvt unit, STTNI/AESNI/PCLMULQDQ unit, RCP/RSQRT unit, F2I convert unit	SIMD ALU, SIMD FPadd unit, F2I convert unit
MEC	Load/Store	

**注意:**

1. 有効なインデックスとディスプレースメントを持つ LEA は複数のマイクロオペレーション (μop) に分けられ、両方のポートを使用します。有効なインデックスを持つ LEA はポート 1 で実行されます。

メモリー実行クラスタ (MEC) (図 16-2 に緑色で示されている) は、32 ビットと 36 ビットの物理アドレスモードをサポートします。Silvermont<sup>†</sup> マイクロアーキテクチャーは、2 つのレベルからなるデータ TLB を実装しており、スモールページとラージページ (2MB または 4MB) をサポートしています。第 1 レベルのマイクロ TLB (μTLB) は小さく、より大きな第 2 レベルの TLB (DTLB) にバックアップされます。命令 TLB ミスとデータ TLB ミスはどちらもハードウェア・ページ・ウォーカーによって処理されます。

MEC には、すべてのロードとストアのスケジューリングを行う MEC RSV もあります。ロード命令とストア命令は、後のパイプラインでメモリーを並べ替えなくても済むように、プログラム順にアドレス生成処理が行われます。そのため、不明なアドレスによって新しいメモリー命令がストールします。(μTLB ミスやリソースが利用できないなどの) 問題が発生したメモリー操作は RehabQ (修復キュー) と呼ばれる別のキューに配置されるため、後続の命令をすべてストールする代わりに、(問題が発生していない) より新しい命令の実行を継続できます。問題が発生した命令は、問題が解決した後に RehabQ から再発行されます。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、データ・キャッシュミスは 8 回までブロックされないため、ロードミスはそれほど問題と見なされません。

バスクラスタ (BIU) の L2 キャッシュは、プロセッサ・コア外部とのすべての通信を処理します。この L2 キャッシュは最大 1MB で、前世代の Intel Atom<sup>®</sup> マイクロアーキテクチャーと比べるとレイテンシーが最適化されています。前世代の Intel Atom<sup>®</sup> プロセッサのフロントサイド・バスに代わり、最適化された新しいメモリー・コントローラーに接続するイントラダイ・インターコネクト (IDI) ファブリックが採用されています。BIU には L2 データ・ブリフエッチャーも装備されます。

新しいコアレベルのマルチプロセッシング (CMP) システム構成では、2 つのプロセッサ・コアが 1 つの BIU に要求を送り、コア間の多重化は BIU によって処理されます。この基本 CMP モジュールを複製してクアドコア構成を作成したり、1 コアのみにしてシングルコア構成を作成できます。

### 16.1.2.1 整数パイプライン

ロードのパイプライン・ステージがほかの整数パイプラインとインライン化されなくなったため、ロードを伴わない操作の実行を高速化し、分岐予測のペナルティが前世代の Intel Atom® プロセッサよりも 3 サイクル少なくなっています。フロントエンドのパイプライン・ステージは前世代の Intel Atom® プロセッサと同じで、フェッチに 3 サイクル、デコードに 3 サイクルかかります。ARR パイプステージは、アウトオブオーダー・アロケーションとレジスターのリネームを行い、必要に応じてマイクロオペレーション (μop) を分割し、各リザベーション・ステーションへ送ります。RSV ステージでは、各リザベーション・ステーションがそれぞれのスケジューリングを行います。実行パイプラインは前世代の Intel Atom® プロセッサによく似ています。マイクロオペレーション (μop) のすべての部分の操作が完了すると、ROB がインオーダーで最終処理を行います。

### 16.1.2.2 浮動小数点パイプライン

INT パイプラインよりも FP パイプラインのほうが長く、命令に応じて 1 ~ 5 の実行ステージがあります。ほかのインテル® マイクロアーキテクチャーと同様に、Silvermont<sup>†</sup> マイクロアーキテクチャーでもハイパフォーマンスを達成するため、FP アシスト (特定の浮動小数点操作を実行パイプラインでネイティブに処理できず、マイクロコードで実行しなければならない場合) の数を最小限に抑える必要があります。そのため、可能な場合は例外をマスクし、DAZ (デノーマルをゼロとして扱う) フラグと FTZ (ゼロフラッシュ) フラグを設定して実行します。

前述のように、各 FPC RSV において命令はインオーダーでスケジュールされますが、RSV 間でアウトオブオーダーになってもかまいません。

## 16.2 Silvermont<sup>†</sup> マイクロアーキテクチャーにおけるコーディングの推奨事項

第 3 章で説明されている一般的なコーディングの推奨事項は、Goldmont<sup>†</sup> と Silvermont<sup>†</sup> マイクロアーキテクチャーにも適用できます。この章の残りでは、一般的な推奨事項の補足と Goldmont<sup>†</sup> と Silvermont<sup>†</sup> マイクロアーキテクチャー固有の手法について説明します。

### 16.2.1 フロントエンドの最適化

#### 16.2.1.1 命令デコーダー

一部の IA 命令は、複雑なタスクを実行するため複数のマイクロオペレーション (μop) にデコードされるマイクロコード・シーケンサー ROM (MSROM) のルックアップが必要になります。MSROM ルックアップが必要な命令については、14.3 のレイテンシー/スループットの表を参照してください。

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、前の世代よりも MSROM のルックアップが大幅に改善されましたが、Goldmont<sup>†</sup> マイクロアーキテクチャーでは、MSROM を必要とする命令の数は、Silvermont<sup>†</sup> マイクロアーキテクチャーに比べ非常に少なくなっています。マイクロコード・フローは、できるだけ回避することが推奨されます。

表 16-6 に、MSROM からデコードされる命令を置き換えることができる非 MSROM 命令のシーケンスを示します。

表 16-6 MSR0M 命令の代替

Instruction from MSR0M	Recommendation for Silvermont	Recommendation for Goldmont
CALL m16/m32/m64	Load + CALL reg	Load + CALL reg
PUSH m16/m32/m64	Load + PUSH reg	Use as is (non MSR0M)
LEAVE	No recommended replacement	Use as is (non MSR0M)
FLD/FST/FSTP m80fp	No recommended replacement	Use as is (non MSR0M)
FCOM+FNSTSW	FCOMI	FCOMI
(I)MUL r/m16 (Result DX:AX)	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32
(I)MUL r/m32 (Result EDX:EAX)	Use (I)MUL r32, r/m32 if extended precision not required, or (I)MUL r64, r/m64	Use as is (non MSR0M)
(I)MUL r/m64 (Result RDX:RAX)	Use (I)MUL r64, r/m64 if extended precision not required	Use as is (non MSR0M)
PEXTRB/D/Q	No recommended replacement	Use as is (non MSR0M)
PMULLD	No recommended replacement	Use as is (non MSR0M)

**チューニングの推奨 1:** perfmon カウンター MS\_DECODED.MS\_ENTRY を使用して、MSR0M が必要な命令の数を特定します (すべてのアシストとフォルトが含まれる)。

**アセンブリ/コンパイラ・コーディング規則 1 (影響 M、一般性 M):** 命令長をできるだけ短くすることで、プリデコード・ビットを効率良く再利用できます。

プリデコード・ビットが正しくないと、デコードのスループットが 3 サイクルごとに 1 命令に減少するため、命令キャッシュのエイリアシングとスラッシングを避けます。

**チューニングの推奨 2:** perfmon カウンター DECODE\_RESTRICTION.PREDECODE\_WRONG を使用して、プリデコード・ビットが正しくないことによるデコードの制限によって命令デコードのスループットが低下した回数を調査します。

### 14.2.1.2 フロントエンドの IPC が高い場合の考慮事項

一般に、サイクルあたりの命令数 (IPC) が高く (>1 に) なるまで、フロントエンドがパフォーマンスを制限することはありません。

デコーダーでサイクルあたり 2 命令を処理するには、次のデコードの規則に従う必要があります。

- Y MSR0M 命令はできるだけ回避します。典型的な例は CALL near の間接メモリー形式です。メモリーバージョンの PUSH と CALL の代わりに、レジスターヘロードし、レジスターバージョンの PUSH と CALL を実行します。
- Y サイクルごとにデコードできる命令バイト長は、マイクロアーキテクチャーによって異なります。
  - Silvermont<sup>†</sup> マイクロアーキテクチャーでは、一緒にデコードされる命令ペアの長さの合計は 16 バイト未満に、最初の命令の長さは 8 バイト以下にします。例えば、命令が 8 バイトを超えるとデコーダー 0 では、サイクルあたり 1 命令しかデコードできません。
  - Goldmont<sup>†</sup> マイクロアーキテクチャーでは、アライメントに依存してサイクルあたり 最大 20 バイトです (例えば、3 つの連続した命令の最初の命令が 4 バイト境界でアライメントされ、3 つの命令シーケンスがデコードの制限を満たす場合)。命令長が 8 バイトを超える場合、デコーダー 0 やサイクルごとに 1 命令に限定されません。
- Y 複数のプリフィクスを持つ命令は、デコーダーのスループットを制限します。プリフィクスとエスケープの合計バイト数が制限に当てはまります。命令のプリフィクス + エスケープは、以下のマイクロアーキテクチャーの制限を超えないようにします。

- Silvermont<sup>†</sup> マイクロアーキテクチャー: 3 バイトを超えるとペナルティーが発生します。
- Goldmont<sup>†</sup> マイクロアーキテクチャー: 4 バイトを超えるとペナルティーが発生します。したがって、上位 8 つのレジスターをアクセスするインテル® SSE4 や AES 命令には、ペナルティーが科せられません。
- Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーでは、デコーダー 0 のみがプリフィクス/エスケープバイトの制限を超えた命令をデコードできます。

Y 各サイクルでデコード可能な分岐の最大数は、Silvermont<sup>†</sup> マイクロアーキテクチャーでは 1、Goldmont<sup>†</sup> マイクロアーキテクチャーでは 2 です。条件分岐を避けることでリステアを防ぎます。

前世代と異なり、Silvermont<sup>†</sup> マイクロアーキテクチャーでは、同じサイクルで 2 つの x87 命令をデコードしても 2 サイクルのペナルティーは発生しません。分岐デコーダーの制限も緩和されています。前世代の Intel Atom® プロセッサでは、デコーダー 0 で条件分岐または間接分岐の次の命令のデコードに 2 サイクルのペナルティーが発生しました。

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、デコーダー 0 で条件分岐または間接分岐の次の命令をペナルティーなしでデコードできます。ただし、(デコーダー 1 にある) 次の命令も分岐である場合、その分岐命令で 3 サイクルのペナルティーが発生します。

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、不成立 (not taken) として予測された分岐をデコーダー 0 またはデコーダー 1 でデコードでき、さらに 3 サイクルのリステア・ペナルティーなしで他の分岐をデコーダー 2 でデコードできます。しかし、不成立として予測された分岐がデコーダー 0 と 1 に 2 つある場合、デコーダー 1 にある 2 番目の分岐は 3 サイクルのペナルティーを被ります。

すべての世代の Intel Atom® プロセッサにおいて、分岐ターゲットが成立すると予測された条件分岐や無条件分岐である場合、1 サイクルのバブルを挟んでデコードされます。

**アセンブリ/コンパイラ・コーディング規則 2 (影響 MH, 一般性 H):** サイクルあたり 2 命令のスルーブットを達成するため、次の命令の使用はできるだけ控えます: (i) MSROM を使用する命令、(ii) プリフィクス + エスケープが制限を超える命令、(iii) 長さが 8 バイトを超える命令、(iv) 連続する分岐命令。

例えば、通常 Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーでは、3 バイトのプリフィクスとエスケープを持つ下位の 8 つのレジスターをアクセスする命令をデコードできます。次に例を示します。

```
PCLMULQDQ 66 0F 3A 44 C7 01 pclmulqdq xmm0, xmm7, 0x1
```

XMM レジスターの上位いずれか (XMM8-15) が参照される場合は、追加の REX プリフィクスも必要になります。その結果、Goldmont<sup>†</sup> マイクロアーキテクチャーでは通常どおりにデコードされますが、Silvermont<sup>†</sup> マイクロアーキテクチャーではデコードのペナルティーが科せられます。次に例を示します。

```
PCLMULQDQ 66 41 0F 3A 44 C0 01 pclmulqdq xmm0, xmm8, 0x1
```

(66 と 0F 3A の間に REX バイト 41 が追加されていることが分かります)。

この 4 つ目のプリフィクスにより、デコードで 3 サイクルのペナルティーが生じます。さらに、このプリフィクスは命令をデコーダー 0 でデコードすることを強制します。命令がデコーダー 1 で開始された場合、デコーダー 0 へ切り替えるのに 3 サイクルかかり、ペナルティーはさらに大きくなります (デコーダーのペナルティーは合計 6 サイクルになります)。そのため、ハイパフォーマンスなアセンブリを記述するには、これらを考慮することを推奨します。これらのケースが頻繁に発生しなければ、成立分岐ターゲットや MS エントリーポイントによってあらかじめデコーダー 0 へアライメントしたほうが良いでしょう。NOP 命令は、パイプラインのほかのリソースを消費するため、NOP の挿入は最終手段として行うべきです。MS エントリーポイントも、デコーダー 1 で開始した場合 3 サイクルのペナルティーが発生するため、同様のアライメントが必要です。プリフィクス/エスケープ長とリステアの制限に関連するペナルティーは、Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーの両方に適用されます。

表 16-7 は、Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーのデコーダーの能力の違いを示しています。

表 16-7 デコーダーの能力の比較

	Goldmont Microarchitecture	Silvermont Microarchitecture
Width	3	2
Max Throughput	20 bytes per cycle (1st instr. aligned to 4B boundary and decoder 1 and 2 restrictions)	16 bytes per cycle (1st instr. <= 8 bytes)
Prefix/Escape Limit	4 bytes	3 bytes
Branch	2	1

### 16.2.1.3 4GB 境界を超える分岐

フロントエンドにおけるもう 1 つの重要なパフォーマンスの考慮事項は分岐予測です。64 ビット・アプリケーションでは、分岐ターゲットが 4GB 以上離れている場合、分岐予測のパフォーマンスに悪影響を与えます。これは、アプリケーションが共有ライブラリーと分離されている場合に発生する可能性があります。新しい glibc のバージョン (2.23 以降) では、この問題を避けるため共有ライブラリーを初めの 2GB に配置できます。環境変数 LD\_PREFER\_MAP\_32BIT\_EXEC に 1 を設定します。プログラマーは、コードの局所性を改善するため静的にビルドすることもできます。LTO によるビルドでは、パフォーマンスさらに向上させなければなりません。

### 16.2.1.4 ループアンロールおよびループストリーム検出器

Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーは、バックエンドにデコード済みのマイクロオペレーション (μop) を提供するループストリーム検出器 (LSD) を備えています。これは、パフォーマンスと消費電力において利点をもたらします。LSD を利用することで、プリフィクス + エスケープのバイト数や命令の長さなどのフロントエンドの制限が排除されます。

ループのオーバーヘッドを減らし、独立したループ反復の作業量を増やす 1 つの方法として、ソフトウェアによるループアンロールが利用できます。ただし、ループアンロールは利点をもたらす一方、パフォーマンスを低下させる恐れもあるため、慎重に使用しなければなりません。パフォーマンスの低下は、コードサイズが大きくなったり、BTB およびレジスターの負が増えることで生じます。また、ループアンロールにより、ループサイズが LSD の上限を超える可能性があるため、ループが LSD に収まるようにループサイズを、Goldmont<sup>†</sup> マイクロアーキテクチャーの 3 ワイドのデコーダーでは 27 命令未満に、Silvermont<sup>†</sup> マイクロアーキテクチャーでは 28 命令未満に抑える対策が必要です。ループサイズが LSD サイズ以下となるように注意する必要があります。

**ユーザー/ソース・コーディング規則 1 (影響 M、一般性 M):** 反復数の多いショートループでループアンロールを利用する場合は、反復あたりの命令数を 28 未満に抑えます。

**チューニングの推奨 3:** perfmon カウンター BACLEAR.SANY を使用して、ループアンロールにより負荷が大きくなりすぎているかを確認します。また、perfmon カウンター ICACHE.MISSES で、ループアンロールにより命令フットプリントに大きな悪影響が生じていないかを確認できます。

### 16.2.1.5 コードとデータの混在

Intel Atom<sup>®</sup> プロセッサは、コードとデータが異なるページにある場合に最適に動作します。ソフトウェアは、フォルス SMC 条件の発生を避けるため同じページ内でコードとデータを共有しないようにしなければなりません。この推奨事項はすべてのページサイズに適用されます。



## 16.2.2 実行コアの最適化

### 16.2.2.1 スケジューリング

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、整数命令でアウトオブオーダー実行が導入されているため、前世代と比べると命令の実行順序が変動する可能性があります。FP 命令には専用のリザーベーション・ステーションが 2 つありますが、互いにインオーダーで実行されます。メモリー命令もインオーダーで発行されますが、修復キュー (Rehab Queue) が追加されているため、アウトオブオーダーで完了することができ、メモリーシステムの遅延によって実行が妨げられることはありません。

Goldmont<sup>†</sup> マイクロアーキテクチャーは、IEC、FPC、そして ME パイプライン全体での完全なアウトオブオーダー実行を特徴としており、これは 3 ポートの IEC、128 ビットの FPC データパス、専用のロードアドレスおよびストアアドレス・パイプラインなど広範囲な強化により達成されます。

**チューニングの推奨** 4: perfmon カウンター `μop_NOT_DELIVERED.ANY` (Silvermont<sup>†</sup> マイクロアーキテクチャーでは `NO_ALLOC_CYCLE.ANY`) を使用すると、バックエンドのパフォーマンス・ボトルネックが分かります。このカウンター値には、メモリーシステムの遅延や実行の遅延などが含まれます。

### 16.2.2.2 アドレス生成

前世代の Intel Atom<sup>®</sup> マイクロアーキテクチャーのアドレス生成の制限は、Silvermont<sup>†</sup> マイクロアーキテクチャーでは解決されています。そのため、Goldmont<sup>†</sup> と Silvermont<sup>†</sup> マイクロアーキテクチャーでは、LEA 命令と ADD 命令のどちらを使ってアドレスを生成してもその効果は同じです。

経験則上、SCALE を使用するが、有効なインデックスやディスプレースメントを持つ LEA を非破壊デスティネーション (特にスタックオフセット) に使用します。そうでない場合は ADD を使用すると良いでしょう。

### 16.2.2.3 FP 乗算-加算-ストアの実行

Goldmont<sup>†</sup> マイクロアーキテクチャーは、統合された FPC リザーベーション・ステーションにより、Silvermont<sup>†</sup> マイクロアーキテクチャーで FPC `μop` のインオーダー・スケジューリングのポート内の依存性によるパフォーマンスの問題を排除します。次の段落と例 16-1 はこの問題を示します。

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、異なるポートで実行する FP 算術命令は互いにアウトオブオーダーで実行できます。そのため、アンロールされたループで乗算結果を加算命令に供給し、その結果をストアする場合、ループの最後にストア命令をまとめることでパフォーマンスが向上します。この方法では、乗算命令と加算命令の実行をオーバーラップさせることができます。例 16-1 について考えてみます。

例 16-1 乗算-ストアポートの競合によってアンロールされたループはインオーダーで実行

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
mulps, xmm1, xmm1	E	E	E	E	E													
	X	X	X	X	X													
	1	2	3	4	5													
addps xmm1, xmm1						E	E	E										
						X	X	X										
						1	2	3										
movaps mem, xmm1									E									
									X									
									1									
mulps, xmm2, xmm2										E	E	E	E	E				
										X	X	X	X	X				
										1	2	3	4	5				
addps xmm2, xmm2															E	E	E	
															X	X	X	
															1	2	3	
movaps mem, xmm2																		E
																		X
																		1

データの依存性により、加算命令は、対応する乗算命令が実行されるまで実行を開始できません。乗算命令とストア命令は、同じポートを使用するため、プログラム順に実行しなければなりません。つまり、2 つ目の乗算命令は 1 つ目の乗算命令および加算命令と依存性がないにもかかわらず、実行を開始できません。次のように、ループの最後にストア命令をグループ化することで、2 つ目の乗算命令を 1 つ目の乗算命令と並列に実行できます (乗算命令をオーバーラップさせると 1 サイクルのバブルが発生する)。

例 16-2 スタア命令をグループ化することでバブルを排除し IPC を向上

Instruction	1	2	3	4	5	6	7	8	9	10	11
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5						
addps xmm1, xmm1						EX1	EX2	EX3			
mulps, xmm2, xmm2		bubble	EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2								EX1	EX2	EX3	
movaps mem, xmm1									EX1		
movaps mem, xmm2											EX1

### 16.2.2.4 整数乗算の実行

Goldmont<sup>†</sup> と Silvermont<sup>†</sup> マイクロアーキテクチャーには専用の整数乗算器が備わっており、一般的に使用される形式の整数乗算のフローを加速します。表 16-8 に MSR0M を使用しない各種 MUL/IMUL 命令形式のレイテンシーとマイクロオペレーション (μop) の数を示します。マイクロコードを利用する乗算形式は回避すべきです。

表 16-8 整数乗算命令のレイテンシー

Integer Multiply Operations	Output	Goldmont Latency	Silvermont Latency
imul/mul r/m8	16	4 <sup>u</sup>	5 <sup>u</sup>
imul/mul r/m16	32	4 <sup>u</sup>	5 <sup>u</sup>
imul/mul r/32	64	3	4 <sup>u</sup>
imul/mul r/m64	128	5	7 <sup>u</sup>
imul/mul r16, r/m16; r16, r/m16, imm	16	4 <sup>u</sup>	4 <sup>u</sup>
imul/mul r32, r/m32; r32, r/m32, imm	32	3	3
imul/mul r64, r/m64; r64, r/m64, imm8	64	5	5

u: ucode flow from MSROM

### 16.2.2.5 ゼロイディオム

XOR/PXOR/XORPS/XORPD 命令は、ソースレジスターとデスティネーションに同じレジスターを指定して (例: XOR eax, eax)、レジスター値をゼロに設定するのによく使用されます。

同等の命令として MOV eax, 0x0 命令がありますが、MOV エンコードのほうが XOR よりもコードバイトが大きくなるため、コンパイラーにとっては MOV よりもこれらの命令のほうが好まれます。

Goldmont<sup>†</sup> と Silvermont<sup>†</sup> マイクロアーキテクチャーには、これらのケースを認識し、アーキテクチャーのレジスターファイルでどちらのソースも有効としてマークする特別なハードウェア・サポートが備わっています。どのような値であってもそれ自身と XOR することでゼロに設定できるため、これにより XOR を高速に実行できます。

このロジックは、PXOR、XORPS、XORPD でもサポートされます。

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、REX.W を使用する 64 ビット汎用オペランドのゼロイディオムに遅延が生じます。ゼロイディオムは、XMM8 - XMM15 または、REX.W なしの上位 8 つの汎用レジスターでサポートされます。そのため、r8 をゼロに設定するには、XOR r8, r8 ではなく XOR r8d, r8d と指定します。

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、64 ビット・オペランドのゼロイディオムをサポートします。

### 16.2.2.6 慣用的な NOP

NOP 命令は、パディングやアライメントの目的で使用されることがあります。Goldmont<sup>†</sup> マイクロアーキテクチャーは、NOP をリザーベーション・ステーションへ割り当てることなく完了できるハードウェアのサポートを備えています。これは、実行リソースと帯域幅を節約します。しかし、リタイアメントのリソースはまだ必要です。

### 16.2.2.7 ムーブの排除 (Move Elimination) と ESP の折りたたみ (Folding)

ムーブの排除 (3.5.1.12 節のゼロレイテンシーの MOV を参照) は、Goldmont<sup>†</sup> マイクロアーキテクチャーでサポートされます。ムーブの排除が有効である場合、それらの命令は高いスループットに加え、0 サイクルのレイテンシーで実行できます。特に、32 ビットと 64 ビットのオペランドサイズを持つ MOV と、XMM を使用する MOVAPS/MOVPD/MOVDQA/MOVDQU/MOVUPS/MOVUPD 命令は、ムーブの排除が有効である場合 0.33 サイクルのスループットを持ちます。MOVZX と MOVZX は、ムーブの排除をサポートしていません。

PUSH/POP/CALL/RET を使用するスタック操作は、Goldmont<sup>†</sup> マイクロアーキテクチャーでは Silvermont<sup>†</sup> マイクロアーキテクチャーよりも効率的です。Goldmont<sup>†</sup> マイクロアーキテクチャーでは、スタック・ポインター・アドレスの計算にアロケーションと実行リソースを消費しません。さらに、PUSH/POP のスループットは、サイクルあたり 1 から 3 クロックに増加しています。

### 16.2.2.8 スタック操作命令

間接メモリー形式の CALL m16/m32/m64 は、MSROM からの  $\mu\text{op}$  フローにデコードされます。レジスターで指定されるターゲットを持つ間接 CALL は、遅延を回避できます。そのため、ターゲットアドレスのレジスターへのロードに続いて、レジスターオペランドを介して間接 CALL を行うことが推奨されます。

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、PUSH m16/m32/m64 のデコードに MSROM を必要としません。これは、LEAVE 命令にも当てはまります。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、PUSH m16/m32/m64 と LEAVE は デコードに MSROM を必要とします。

### 16.2.2.9 フラグの利用

多くの命令には、フラグレジスターに格納される暗黙のデータがあります。これらのデータは、条件移動 (CMOVS)、分岐、さまざまな論理/算術演算 (RCL など) といった幅広い命令で利用されます。分岐条件としてよく使用される命令に比較命令 (CMP) があります。CMP 命令に依存する分岐は次のサイクルで実行できます。ADD 命令や SUB 命令に依存する分岐でも同様のことが言えます。

INC 命令と DEC 命令は一部のフラグのみ設定するため、フラグをマージする追加のマイクロオペレーション ( $\mu\text{op}$ ) が必要になります。そのため、INC 命令や DEC 命令に依存する分岐には 1 サイクルのペナルティーが伴います。このペナルティーは、INC 命令や DEC 命令に直接依存する分岐にのみ適用されます。

**アセンブリー/コンパイラー・コーディング規則 3 (影響 M、一般性 M):** 分岐条件には、INC/DEC 命令ではなく可能な限り CMP/ADD/SUB 命令を使用します。

### 16.2.2.10 SIMD 浮動小数点と x87 命令

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、SIMD FP 実行ユニットのサブセットのみが、128 ビット幅のデータパスで実装されています。Goldmont<sup>†</sup> マイクロアーキテクチャーでは、完全な SIMD FP ユニットが 128 ビット・データパスで実装されています。Goldmont<sup>†</sup> マイクロアーキテクチャーを Silvermont<sup>†</sup> と比較すると、一般にパックド SIMD 命令のレイテンシーは 1 サイクル少なく、スループットは倍で実行されます。

特に、MULPD 命令のレイテンシーは 7 サイクルから 4 サイクルに短縮され、スループットは 4 サイクルごとから 1 サイクルごとで 4 倍を達成します。

Goldmont<sup>†</sup> マイクロアーキテクチャーではまた、x87 拡張精度のロードとストア、FLD m80fp と FST/FSTP m80fp のレイテンシーとスループットが改善されています。詳細は表 16-14 を参照してください。

### 16.2.2.11 SIMD 整数命令

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、比較的小さな SIMD 整数命令のサブセットは、サイクルあたり 2 命令のスループットで実行されます。Goldmont<sup>†</sup> マイクロアーキテクチャーの場合、より多くの SIMD 整数命令をサイクルあたり 2 命令のレートで実行を完了できます。

Goldmont<sup>†</sup> マイクロアーキテクチャーにおけるレイテンシーと/または、スループットの改善には、1 つのポートのみで実行されるその他の SIMD 整数命令も含まれます。例えば、PMULLD は Silvermont<sup>†</sup> マイクロアーキテクチャーでは、11 サイクルのレイテンシーと 11 サイクルごとに 1 つのスループットを提供していました。Goldmont<sup>†</sup> マイクロアーキテクチャーでは、5 サイクルのレイテンシーと 2 サイクルごとに 1 つのスループットに改善されています。

一般に、SIMD 整数乗算器のハードウェアは、Silvermont<sup>†</sup> マイクロアーキテクチャーよりもかなり高速化 (4 サイクルのレイテンシー) され、高いスループット (1 サイクル) です。さらに、PADDQ/PSUBQ 命令は、Silvermont<sup>†</sup> マイクロアーキテクチャーでは 4 サイクルのレイテンシーと 4 サイクルごとのスループットであったのに対し、

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、2 サイクルのレイテンシーと 2 サイクルごとのスループットを提供します。また、PSHUFB 命令の Goldmont<sup>†</sup> マイクロアーキテクチャーでのレイテンシーとスループットは 1 サイクルですが、Silvermont<sup>†</sup> では 5 サイクルのレイテンシーと 5 サイクルごとのスループットです。詳細は表 16-14 を参照してください。

### 16.2.2.12 ベクトル化の注意事項

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、高いスループットを実装する SIMD 実行ユニットの可用性と、MSROM から長い  $\mu\text{op}$  フローへのデコードが要求される SIMD 命令によって、有益なベクトル化の機会が制限される可能性があります。

Goldmont<sup>†</sup> マイクロアーキテクチャーは、直接的なプログラミングと同様にコンパイラーが、各種 SIMD 命令のレイテンシーとスループットの改善によるベクトル化の利益を得ることを可能にします。

**アセンブリー/コンパイラー・コーディング規則 4 (影響 M、一般性 M):** コードのベクトル化には MSROM 命令の使用を回避します。

### 16.2.2.13 その他の SIMD 命令

Silvermont<sup>†</sup> マイクロアーキテクチャーは、ブロック暗号化/復号向けの AES や AES-GCM などの各種暗号化アルゴリズムのパフォーマンスを加速するインテル® AES-NI と PCLMULQDQ 命令をサポートします。

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、実行ハードウェアの実行レイテンシー、デコードのスループットが改善されました。例えば、PCLMULQDQ 命令は、Silvermont<sup>†</sup> マイクロアーキテクチャーでは 10 サイクルのレイテンシーと 10 サイクルごとのスループットであったのに対し、Goldmont<sup>†</sup> マイクロアーキテクチャーでは、6 サイクルのレイテンシーと 4 サイクルごとのスループットを提供します。

さらに、Goldmont<sup>†</sup> マイクロアーキテクチャーは、SHA1 と SHA256 などの安全なハッシュ・アルゴリズムのパフォーマンスを加速する SHANI 命令をサポートします。安全なハッシュ・アルゴリズムと SHANI の詳細については次のサイトをご覧ください。

<https://software.intel.com/en-us/articles/intel-sha-extensions> (英語)

インテル® SHA 拡張を使用した実装の例と参考資料は、以下のサイトでご覧いただけます:

<https://software.intel.com/en-us/articles/intel-sha-extensions-implementations> (英語)

### 16.2.2.14 命令の選択

表 16-9 に、Silvermont<sup>†</sup> マイクロアーキテクチャーの浮動小数点操作と SIMD 整数操作のレイテンシーを示します。「スループット」は、利用可能な実行ユニットで実行が完了できる命令ごとのサイクル数を表しています (例えば、4 は 4 サイクルごとに同じ命令を開始できることを示し、0.33 は 3 つの同じ命令が各サイクルで実行を完了できることを示します)。

表 16-9 浮動小数点と SIMD 整数のレイテンシー

	Goldmont		Silvermont	
	Latency	Throughput	Latency	Throughput
<b>SIMD integer ALU</b>				
128-bit ALU/logical/move	1	0.5	1	0.5
64-bit ALU/logical/move	1	0.5	1	0.5
<b>SIMD integer shift</b>				
128-bit	1	0.5	1	1
64-bit	1	0.5	1	1
<b>SIMD shuffle</b>				
128-bit	1	0.5	1	1
64-bit	1	0.5	1	1
<b>SIMD integer multiplier</b>				
128-bit	4	1	5	2
64-bit	4	1	4	1
<b>FP Adder</b>				
x87 (fadd)	3	1	3	1
scalar (addsd, addss)	3	1	3	1
packed (addpd, addps)	3	1	4	2
<b>FP Multiplier</b>				
x87 (fmul)	5	2	5	2
scalar single-precision (mulss)	4	1	4	1
scalar double-precision (mulsd)	4	1	5	2
packed single-precision (mulps)	4	1	5	2
packed double-precision (mulpd)	4	1	7	4
<b>Converts</b>				
CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTPD2DQ, CVTPD2PI, CVTPS2DQ	4	1	5	2
CVTPI2PS, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI	4	1	4	1
<b>FP Divider</b>				
x87 fdiv (extended-precision)	39	39	39	39
x87 fdiv (double-precision)	34	34	34	34
x87 fdiv (single-precision)	19	19	19	19
scalar single-precision (divss)	19	18	19	17
scalar double-precision (divsd)	34	33	34	32
packed single-precision (divps)	36	35	39	39
packed double-precision (divpd)	66	65	69	69

インテル® SSE のスカラー単精度命令は、ほとんどの FP 命令よりも 1 サイクル高速であることを覚えておいてください。この表を調べることで、インテル® SSE のパックド倍精度命令はスカラーバージョンと比べて長いレイテンシーと低いスループットであることが分かります。

**アセンブリ/コンパイラ・コーディング規則 5 (影響 M、一般性 M):** x87 浮動小数点命令よりもインテル® SSE 浮動小数点命令を利用したほうが良いでしょう。

**アセンブリ/コンパイラ・コーディング規則 6 (影響 MH、一般性 M):** (可能な限り) 例外をマスクして、DAZ フラグと FTZ フラグを設定して実行します。

**チューニングの推奨 5:** perfmon カウンター MACHINE\_CLEARS.FP\_ASSIST を使用して、浮動小数点例外がプログラムのパフォーマンスに影響しているかどうかを確認できます。

### 16.2.2.15 整数除算

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、整数除算には比較的長く低速なマイクロコード・フローを必要とします。レイテンシーは、入力値とデータサイズによって異なります。Goldmont<sup>†</sup> マイクロアーキテクチャーでは、MSROM を使用しない DIV/IDIV の短精度形式向けのハードウェア強化が行われています。高い精度を必要とする DIV/IDIV 形式は MSROM を使用しますが、これもハードウェアの強化により加速されます。表 16-10 と表 16-11 は、除算命令のレイテンシー範囲と MSROM を必要とする命令 ('u' が記されている) を示しています。

表 16-10 符号なし整数除算操作のレイテンシー

	Dividend	Divisor	Quotient	Remainder	Silvermont <sup>u</sup>	Goldmont
DIV r8	AX	r8	AL	AH	25	11-12
DIV r16	DX:AX	r16	AX	DX	26-30	12-17 <sup>u</sup>
DIV r32	EDX:EAX	r32	EAX	EDX	26-38	12-25 <sup>u</sup>
DIV r64	RDX:RAX	r64	RAX	RDX	38-123	12-41 <sup>u</sup>

表 16-11 符号付き整数除算操作のレイテンシー

	Dividend	Divisor	Quotient	Remainder	Silvermont <sup>u</sup>	Goldmont
IDIV r8	AX	r8	AL	AH	34	11-12
IDIV r16	DX:AX	r16	AX	DX	35-40	12-17 <sup>u</sup>
IDIV r32	EDX:EAX	r32	EAX	EDX	35-47	12-25 <sup>u</sup>
IDIV r64	RDX:RAX	r64	RAX	RDX	49-135	12-41 <sup>u</sup>

**ユーザー/ソース・コーディング規則 2 (影響 M、一般性 L):** 除算は真に必要な場合のみ利用し、最も効率良く実行できるように正しいデータサイズと符号を使用します。

**チューニングの推奨 6:** perfmon カウンター CYCLES\_DIV\_BUSY.ANY を使用して、除算がプログラムのボトルネックになっているかどうかを確認できます。

アライメントされている配列からアライメントされていないパックド単精度のグループを取得する場合、MOVUPS よりも PALIGNR が推奨されます。例えば、load A[x+y+3:x+y] について考えてみます。ここで x と y はループ変数です。この場合、(x+y で MOVUPS を使用するよりも) x+y を計算して 4 の倍数に切り下げ、MOVAPS と PALIGNR で要素を取得するほうが適切です。この方法は時間がかかるように見えますが、整数操作は FP 操作と並列に実行できます。また、約 6 サイクルのコストを伴う MOVUPS によるライン分割を回避することもできます。

**ユーザー/ソース・コーディング規則 3 (影響 M、一般性 M):** パックド単精度要素の取得には PALIGNR を使用します。

## 16.2.2.16 整数シフト

レジスター (例えば CL) にシフトカウントを設定する整数シフト命令を使用する場合、パイプラインで実行される先行する命令によってカウントレジスターが生成されていると、スケジューリングに 1 サイクルのバブルが発生します。そのため、ソフトカウントを生成する命令は可能な限り手前で実行すべきです。

また、倍精度シフト命令 (SHLD/SHRD) が 64 ビット入力データを処理する場合、長い MSROM フローが必要となります。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、32 ビットのデスティネーション・レジスターと即値のシフトカウントを持つ SHRD も MSROM からデコードされます。SHLD は MSROM を必要としません。Goldmont<sup>†</sup> マイクロアーキテクチャーでは、32 ビットのデスティネーション・レジスターと即値のシフトカウントを持つ SHLD/SHRD は、MSROM からデコードする必要はありません。32 ビットのデスティネーション・メモリー・オペランドや CL シフトカウントを持つ SHLD/SHRD は、Silvermont<sup>†</sup> と Goldmont<sup>†</sup> の両方で MSROM からのデコードを必要とします。

## 16.2.2.17 ポーズ命令

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、PAUSE 命令のレイテンシーは Silvermont<sup>†</sup> マイクロアーキテクチャーと同じですが、スレッド同期プリミティブによって高い節電能力を達成します。

## 16.2.3 メモリアクセスの最適化

### 16.2.3.1 PALIGNR でアライメントされていないメモリアクセスを軽減

単精度 FP や dword データ配列を使用する場合、配列が 16 バイトにアライメントされていないと 4 つの連続する要素のロードごとにメモリアクセスが発生する可能性があります。例えば、2 つのインデックス  $i, j$  を使用する配列  $A[i+j]$  が入れ子になったループにある場合、内部ループで 1 つインクリメントされる実効インデックス  $i+j$  を使用する 16 バイトのメモリーロードは、4 回のうち 3 回はアライメントされていないアクセスとなります。

これらのアライメントされていないメモリアクセスは回避できます。配列のベースアドレスが 16 バイトにアライメントされていると仮定すると、オリジナルの  $i+j$  の残り  $4*$  から得られる imm8 定数 (PALIGNR によって XMM にすでにロードされている 2 つの連続した 16 バイト・チャンクに続く) の 4 倍数である実効インデックスでロードされるべきです。

**アセンブリ/コンパイラー・コーディング規則 7 (影響 M、一般性 M):** パックド単精度 FP または dword 要素の取得には PALIGNR を使用します。

### 16.2.3.2 メモリー実行の問題を最小化する

Goldmont<sup>†</sup> マイクロアーキテクチャーでは、MEC は完全にアウトオブオーダー実行であり、ロードはアドレス解決されていないストアを先行することができます。これはメモリーのディスアンビゲーション機能によって提供され、若いロードが古いストアと衝突した際にパイプラインのフラッシュを必要とするメモリー順序の問題を軽減します。ハードウェア・メモリー・ディスアンビゲーションがメモリー順序の問題を解決できない状況では、プログラマーはパフォーマンス・カウンター・イベントを使用して、メモリー実行の問題の原因を評価してその場所を特定できます。

Silvermont<sup>†</sup> マイクロアーキテクチャーでは、RehabQ には MEC で対処しなければならないいくつかの実行問題があります。この問題には、ロードブロック、ロード/ストア分割、ロック、TLB ミス、不明なアドレス、および過度のストアなどが含まれます。Silvermont<sup>†</sup> マイクロアーキテクチャーの perfmon カウンター REHABQ は、Silvermont<sup>†</sup> マイクロアーキテクチャー固有の問題を評価するのに使用できます。

**チューニングの推奨 7:** perfmon カウンター MACHINE\_CLEAR.DISAMBIGUATION を使用して、Goldmont<sup>†</sup> マイクロアーキテクチャーと前の世代におけるディスアンビゲーションが失敗するアプリケーション・パフォーマンスへの影響を評価できます。



### 16.2.3.3 ストア・フォワーディング

Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーでは、前世代と比べてストア・フォワーディングが大幅に改善されています。次の条件を満たす場合、先行するストア操作命令から後続のロード命令にデータを転送できます。

- Y ストア操作とロード操作の開始アドレスが同じである。
- Y ロード操作の幅がストア操作の幅以下である。
- Y ストア操作またはロード操作でキャッシュラインの分割が発生しない。

表 16-12 と表 16-13 は、先行するストアのフォワードが成功した場合と、フォワードできない場合の状況を示します。

表 16-12 ストア・フォワーディング条件 (1 バイト・ストアおよび 2 バイト・ストア)

Store Size	Load Size	Load Alignment															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
	2	F	N														

表 16-13 ストア・フォワーディング条件 (4-16 バイト・ストア)

Store Size	Load Size	Load Alignment															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	N	F	N												
	2	F	N	F	N												
	4	F	N	N	N												
8	1	F	N	N	N	N	N	N	N								
	2	F	N	N	N	N	N	N	N								
	4	F	N	N	N	N	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	2	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	4	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	8	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

これらの条件のいずれかが満たされない場合、ロードはブロックされ、再発行のために RehabQ に追加されます。

以下のガイドラインに従って、ストア・フォワーディングの問題を排除/回避できます (推奨順に示します)。

- Y メモリーの代わりにレジスターを使用する。
- Y できるだけ早くストア操作を実行する (ストアはロードよりも後のパイプライン・ステージで実行されるため、ロードよりもかなり先行して実行する必要がある)。

正しくストア・フォワードされるコストは、マイクロアーキテクチャーによって異なります。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、ストア・フォワーディングに 3 サイクル追加されます (つまり、ストアが n サイクルで実行されると、ロードは n + 3 サイクルで実行されます)。Goldmont<sup>†</sup> マイクロアーキテクチャーでのコストは 4 サイクルです。

### 16.2.3.4 PrefetchW 命令

Silvermont<sup>†</sup> と Goldmont<sup>†</sup> マイクロアーキテクチャーは、PrefetchW 命令 (0f 0d /1) をサポートしています。この命令は、RFO (read-for-ownership) 要求で指定したラインをキャッシュにプリフェッチするようにハードウェアにヒントを示します。この命令を使用すると、後続のストアはラインがプリフェッチされていない場合や、別の命令でプリフェッチされた場合よりも、そのラインへの操作を速く完了できます。すべてのプリフェッチ命令は、正しく使用しないとパフォーマンスの低下につながる可能性があるため、PrefetchW を含め、プリフェッチ命令を使用する場合は実際にパフォーマンスが向上するように慎重に利用すべきです。命令オペコード 0f0d /0 は引き続き NOP として処理され、指定されたラインはプリフェッチされません。

### 16.2.3.5 キャッシュラインの分割とアライメント

キャッシュラインの分割は、ロード命令とストア命令の帯域幅を減少させます。そのため、できるだけ回避すべきです。

**チューニングの推奨** 8: perfmon カウンター REHABQ.ST\_SPLIT と REHABQ.LD\_SPLIT を使用すると、複数のキャッシュラインにまたがる操作とその数が分かります。

アライメントされたアクセスが推奨されますが、Silvermont<sup>†</sup> マイクロアーキテクチャーには、アライメントされていないアクセスに対するハードウェア・サポートが備わっています。そのため、前世代の Intel Atom<sup>®</sup> プロセッサとは対照的に、MOVUPS/MOVUPD/MOVDQU 命令はすべて単一のマイクロオペレーション (μop) 命令となっています。

### 16.2.3.6 セグメントベース

簡略化のため、Silvermont<sup>†</sup> マイクロアーキテクチャーの AGU は、セグメントベースが 0 であると想定しています。ほとんどの場合は問題ありませんが、ゼロ以外のセグメントベース (NZB) を使用しなければならないこともあります。NZB を使用する場合、可能な限りセグメントベースをキャッシュライン (0x40) 境界にアライメントします。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、NZB アドレスの生成には 1 サイクルのペナルティーが伴います。Goldmont<sup>†</sup> マイクロアーキテクチャーでは、サイクルごとに 1 つの NZB アドレス生成が可能です。

### 16.2.3.7 コピーと文字列のコピー

通常、memcpy/memset ルーチンを含むライブラリーがコンパイラーによって提供されます。これらのライブラリーは優れたパフォーマンスをもたらすし、コードサイズとアライメントの問題にも対応しています。

memcpy/memset 操作は、最適なバイト/ダブルワード単位のアライメントされた操作に分割した REP MOVSB/STOSB 命令で対応できます。これは、ほとんどの場合に汎用メモリーコピー/セットのソリューションとして利用できます。REP MOVSB/STOSB 命令には固有のオーバーヘッドがあります。REP STOSB は複数のキャッシュラインにまたがる長い文字列に対応できますが、REP MOVSB はできません。これは、ソースとデスティネーション間のアライメントの一致が複雑であるためです。

特定のメモリーコピー/セットにおいて SIMD 命令を使用するマクロコード・シーケンスは、アライメント、バッファ長、バッファ内のキャッシュの有無に応じて、ある程度のパフォーマンスの向上をもたらします (約 12 サイクル)。ただし、複数のキャッシュラインにまたがる大きなメモリーのコピーは例外です。よく考慮されたマクロコードはキャッシュラインの分割を回避し、REP MOVSB のパフォーマンスを大幅に向上させます。

Silvermont<sup>†</sup> マイクロアーキテクチャー・ベースのプロセッサは、REP MOVSB と STOSB の拡張操作をサポートしています。MOVSB と STOSB を使用する REP 文字列操作は、メモリーのコピー/セット操作などのよくある状況において、最小コードサイズで柔軟かつハイパフォーマンスな REP 文字列操作を提供します。拡張 MOVSB/STOSB 操作をサポートするプロセッサは、次のように CPUID 機能フラグで検出できます。

CPUID:(EAX=7H, ECX=0H):EBX.[ビット 9] = 1

汎用性のある実装 (将来の実装を含む) で動作する単純なデフォルトの文字列コピー/セットルーチンが必要な場合は、REP MOVSB と STOSB の拡張をサポートする実装でも REP MOVSB または REP STOSB の使用を検討すべきです。これらの命令は、特定の実装では専用のコピー/セットルーチンよりも遅くなることがあり、専用のコピー/セットルーチンは将来のプロセッサで同じように動作せず、また将来の拡張を利用できない恐れがあります。REP MOVSB と REP STOSB は、将来のプロセッサでもある程度のパフォーマンスが期待できます。

## 16.3 命令レイテンシーとスループット

この節では、最新のマイクロアーキテクチャーによる Intel Atom® プロセッサ世代のスループットとレイテンシーの一覧を示します。MSROM によるデコーダーの支援が必要な命令には、「MSROM」列に「Y」マークが記されています (よりデコード効率の高い代替手段があれば、使用は最小限に抑えるべきです)。それぞれの命令のスループットとレイテンシー値は、CPUID DisplayFamily\_DisplayModel による対応するマイクロアーキテクチャーごとにグループ化されています。同じ DisplayFamily にタイミング特性が同じ多数の DisplayModels がある場合、DisplayFamily は一度だけ記載されることがあります。

この節でカバーされるマイクロアーキテクチャーと対応する DisplayFamily\_DisplayModel のシグネチャーを以下に示します。

Y Goldmont<sup>†</sup> マイクロアーキテクチャー: 06\_5CH, 06\_5FH.  
 Y Silvermont<sup>†</sup> または Airmont<sup>†</sup> マイクロアーキテクチャー: 06\_37H 06\_4AH 06\_4CH 06\_4DH 06\_5AH 06\_5DH

表 16-14 Intel Atom® プロセッサの最新のマイクロアーキテクチャーにおける  
命令レイテンシーとスループット

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH
ADC/SBB r32, imm8	1	2	2	2	N	N
ADC/SBB r32, r32	1	2	2	2	N	N
ADC/SBB r64, r64	1	2	2	2	N	N
ADD/AND/CMP/OR/SUB/XOR/TEST r32, r32	0.33	0.5	1	1	N	N
ADD/AND/CMP/OR/SUB/XOR/TEST r64, r64	0.33	0.5	1	1	N	N
ADDPD/ADDSUBPD/MAXPD/MINPD/SUBPD xmm, xmm	1	2	3	4	N	N
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS	1	1	3	3	N	N
MAXPS/MAXSD/MAXSS/MINPS/MINSD/MINSS xmm, xmm	1	1	3	3	N	N
ANDNP/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS	0.5	0.5	1	1	N	N
AESDEC/AESDECLAST/AEENC/AEENCLAST	2	5	6	8	N	Y
AESIMC/AESKEYGEN	2	5	5	8	N	Y
BLENDDP/BLENDPS xmm, xmm, imm8	0.5	1	1	1	N	N
BLENDVPD/BLENDVPS xmm, xmm	4	4	4	4	Y	Y
BSF/BSR r32, r32	8	10	10	10	Y	Y
BSWAP r32	1	1	1	1	N	N
BT/BTC/BTR/BTS r32, r32	1	1	1	1	N	N
CBW	4	4	4	4	Y	Y
CDQ/CLC/CMC	1	1	1	1	N	N
CMOVB r32, r32	1	1	2	2	N	N
CMPPD xmm, xmm, imm	1	2	3	4	N	N
CMPSD/CMPPS/CMPSS xmm, xmm, imm	1	1	3	3	N	N
CMPXCHG r32, r32	5	6	5	6	Y	Y
CMPXCHG r64, r64	5	6	5	6	Y	Y
(U)COMISD/(U)COMISS xmm, xmm;	1	1	4	4	N	N
CPUID	58	60	58	60	Y	Y
CRC32 r32, r32	1	1	3	3	N	N
CRC32 r64, r64	1	1	3	3	N	N
CVTDQ2PD/CVTDQ2PS/CVTPD2DQ/CVTPD2PS xmm, xmm	1	2	4	5	N	N
CVT(T)PD2PI/CVT(T)PI2PD	1	2	4	5	N	N
CVT(T)PS2DQ/CVTPS2PD xmm, xmm;	1	2	4	5	N	N
CVT(T)SD2SS/CVTSS2SD xmm, xmm	1	1	4	4	N	N

表 16-14 Intel Atom® プロセッサの最新のマイクロアーキテクチャーにおける  
命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH	06_37H, 4AH,4C H,4DH,5 AH,5DH
CVTSD/SS xmm, r32	1	1	7	6	N	N
CVTSD2SI/SS2SI r32, xmm	1	1	4	4	N	N
DEC/INC r32	1	1	1	1	N	N
DIV r8	11-12	25	11-12	25	N	Y
DIV r16	12-17	26-30	12-17	26-30	Y	Y
DIV r32	12-25	26-38	12-25	26-38	Y	Y
DIV r64	12-41	38-123	12-41	38-123	Y	Y
DIVPD <sup>1</sup>	12, 65	27-69	13, 66	27-69	N	Y
DIVPS <sup>1</sup>	12,35	27-39	13, 36	27-39	N	Y
DIVSD <sup>1</sup>	12,33	11-32	13,34	13-34	N	N
DIVSS <sup>1</sup>	12,18	11-17	13,19	13-19	N	N
DPPD xmm, xmm, imm	5	8	8	12	Y	Y
DPPS xmm, xmm, imm	11	12	14	15	Y	Y
EMMS	23	10	23	10	Y	Y
EXTRACTPS	1	4	4	5	N	Y
F2XM1	87	88	87	88	Y	Y
FABS/FCHS	0.5	1	1	1	N	N
FCOM	1	1	4	4	N	N
FADD/FSUB	1	1	3	3	N	N
FCOS	154	168	154	168	Y	Y
FDECSTP/FINCSTP	0.5	0.5	1	1	N	N
FDIV	39	39	39	39	N	N
FLDZ	280	277	280	277	Y	Y
FMUL	2	2	5	5	N	N
FPATAN/FYL2X/FYL2XP1	303	296	303	296	Y	Y
FPTAN/FSINCOS	287	281	287	281	Y	Y
FRNDINT	41	25	41	25	Y	Y
FSCALE	32	74	32	74	Y	Y
FSIN	140	150	140	150	Y	Y
FSQRT	40	40	40	40	N	N
HADDPD/HSUBPD xmm, xmm	5	5	5	6	Y	Y
HADDPS/HSUBPS xmm, xmm	6	6	6	6	Y	Y
IDIV r8	11-12	34	11-12	34	N	Y
IDIV r16	12-17	35-40	12-17	35-40	Y	Y
IDIV r32	12-25	35-47	12-25	35-47	Y	Y

表 16-14 Intel Atom® プロセッサの最新のマイクロアーキテクチャーにおける  
命令レイテンシーとスルーブット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5A H,5DH	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH	06_37H, 4AH,4C H,4DH,5 AH,5DH
IDIV r64	12-41	49-135	12-41	49-135	Y	Y
IMUL r32, r32 (single dest)	1	1	3	3	N	N
IMUL r32 (dual dest)	2	5	3 (4, EDX)	4	N	Y
IMUL r64, r64 (single dest)	2	2	5	5	N	N
IMUL r64 (dual dest)	2	4	5 (6,RDX)	5 (7,RDX)	N	Y
INSERTPS	0.5	1	1	1	N	N
MASKMOVDQU	4	5	4	5	Y	Y
MOVAPD/MOVAPS/MOVDQA/MOVDQU/MOVUPD/ MOVUPS xmm, xmm;	0.33 <sup>2</sup> /0. 5	0.5	0/1	1	N	N
MOVD r32, xmm; MOVQ r64, xmm	1	1	4	4	N	N
MOVD xmm, r32; MOVQ xmm, r64	1	1	4	3	N	N
MOVDDUP/MOVHLP/MSOVLHPS/MOVSHDUP/MO VSLDUP	0.5	1	1	1	N	N
MOVDQ2Q/MOVQ/MOVQ2DQ	0.5	0.5	1	1	N	N
MOVSD/MOVSS xmm, xmm;	0.5	0.5	1	1	N	N
MPSADBW	4	5	5	7	Y	Y
MULPD	1	4	4	7	N	N
MULPS; MULSD	1	2	4	5	N	N
MULSS	1	1	4	4	N	N
NEG/NOT r32	0.33	0.5	1	1	N	N
PACKSSDW/wB xmm, xmm; PACKUSWB xmm, xmm	0.5	1	1	1	N	N
PABSB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PADDB/D/W xmm, xmm; PSUBB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PADDQ/PSUBQ/PCMPEQQ xmm, xmm	1	4	2	4	N	Y
PADDSB/w; PADDUSB/w; PSUBSB/w; PSUBUSB/w	0.5	0.5	1	1	N	N
PALIGNR xmm, xmm	0.5	1	1	1	N	N
PAND/PANDN/POR/PXOR xmm, xmm	0.5	0.5	1	1	N	N
PAVGB/W xmm, xmm	0.5	0.5	1	1	N	N
PBLENDW xmm, xmm, imm	0.5	0.5	1	1	N	N
PBLENDVB xmm, xmm	4	4	4	4	Y	Y
PCLMULQDQ xmm, xmm, imm	4	10	6	10	Y	Y
PCMPEQB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PCMPESTR1 xmm, xmm, imm	13	21	19(C)/ 26(F) <sup>3</sup>	21(C)/ 28(F)	Y	Y

表 16-14 Intel Atom® プロセッサの最新のマイクロアーキテクチャーにおける  
命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH	06_37H, 4AH,4C H,4DH,5 AH,5DH
PCMPESTRM xmm, xmm, imm	14	17	15(X)/ 25(F) <sup>1</sup>	17(X)/ 24(F)	Y	Y
PCMPGTB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PCMPGTQ/PHMINPOSUW xmm, xmm	2	2	5	5	N	N
PCMPISTRI xmm, xmm, imm	8	17	14(C)/ 21(F) <sup>1</sup>	17(C)/ 24(F)	Y	Y
PCMPISTRM xmm, xmm, imm	7	13	10(X)/ 20(F) <sup>1</sup>	13(X)/ 20(F)	Y	Y
PEXTRB/WD r32, xmm, imm	1	4	4	5	N	Y
PINSRB/WD xmm, r32, imm	1	1	4	3	N	N
PHADD/PHSUBD xmm, xmm	4	6	4	6	Y	Y
PHADDW/PHADDSw xmm, xmm	6	9	6	9	Y	Y
PHSUBW/PHSUBSw xmm, xmm	6	9	6	9	Y	Y
PMADDUBSw/PMADDwD/PMULHRSw/PSADBw xmm, xmm	1	2	4	5	N	N
PMASB/W/D xmm, xmm; PMASUB/W/D xmm, xmm	0.5	0.5	1	1	N	N
PMINSB/W/D xmm, xmm; PMINSUB/W/D xmm, xmm	0.5	0.5	1	1	N	N
PMOVMskB r32, xmm	1	1	4	4	N	N
PMOVSXBW/BD/BQ/WD/WQ/DQ xmm, xmm	0.5	1	1	1	N	N
PMOVZXBW/BD/BQ/WD/WQ/DQ xmm, xmm	0.5	1	1	1	N	N
PMULDQ/PMULUDQ xmm, xmm	1	2	4	5	N	N
PMULHUW/PMULHW/PMULLW xmm, xmm	1	2	4	5	N	N
PMULLD xmm, xmm	2	11	5	11	N	Y
POPCNT r32, r32	1	1	3	3	N	N
POPCNT r64, r64	1	1	3	3	N	N
PSHUFb xmm, xmm	1	5	1	5	N	Y
PSHUFD xmm, mem, imm	0.5	1	1	1	N	N
PSHUFW; PSHUFLW; PSHUFW	0.5	1	1	1	N	N
PSIGNB/D/W xmm, xmm	0.5	1	1	1	N	N
PSLLDQ/PSRLDQ xmm, imm; SHUFPS/SHUFPS	0.5	1	1	1	N	N
PSLLD/Q/W xmm, xmm	1	2	2	2	N	N
PSRAD/W xmm, imm;	0.5	1	1	1	N	N
PSRAD/W xmm, xmm;	1	2	2	2	N	N
PSRLD/Q/W xmm, imm;	0.5	1	1	1	N	N
PSRLD/Q/W xmm, xmm	1	2	2	2	N	N
PTEST xmm, xmm	1	1	4	4	N	N

表 16-14 Intel Atom® プロセッサの最新のマイクロアーキテクチャーにおける  
命令レイテンシーとスループット (続き)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5A H,5DH	06_5CH, 5FH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH	06_37H, 4AH,4C H,4DH,5 AH,5DH
PUNPCKHBW/DQ/wD; PUNPCKLBW/DQ/wD	0.5	1	1	1	N	N
PUNPCKHQDQ; PUNPCKLQDQ	0.5	1	1	1	N	N
RCPPS/RSQRTPS	6	8	9	9	Y	Y
RCPSS/RSQRTSS	1	1	4	4	N	N
RDTSC	20	30	20	30	Y	Y
ROUNDPD/PS	1	2	4	5	N	N
ROUNDSD/SS	1	1	4	4	N	N
ROL; ROR; SAL; SAR; SHL; SHR (count in CL)	1	1	1 (2 for CL source)	1 (2 for CL source)	N	N
ROL; ROR; SAL; SAR; SHL; SHR (count in imm8)	1	1	1	1	N	N
SAHF	1	1	1	1	N	N
SHLD r32, r32, imm	2	2	2	2	N	N
SHRD r32, r32, imm	2	4	2	4	N	Y
SHLD/SHRD r64, r64, imm	12	10	12	10	Y	Y
SHLD/SHRD r64, r64, CL	14	10	14	10	Y	Y
SHLD/SHRD r32, r32, CL	4	4	4	4	Y	Y
SHUFPD/SHUFPS xmm, xmm, imm	0.5	1	1	1	N	N
SQRTPD	67	70	68	71	N	Y
SQRTPS	37	40	38	41	N	Y
SQRTSD	34	35	35	35	N	Y
SQRTSS	19	20	20	20	N	Y
TEST r32, r32	0.33	0.5	1	1	N	N
UNPCKHPD; UNPCKHPS; UNPCKLPD; UNPCKLPS	0.5	1	1	1	N	N
XADD r32, r32	2	5	4	5	Y	Y
XCHG r32, r32	2	5	4	5	Y	Y
XCHG r64, r64	2	5	4	5	Y	Y
SHA1MSG1/SHA1MSG2/SHA1NEXTE	1	NA	3	NA	N	NA
SHA1RNDS4 xmm, xmm, imm	2	NA	5	NA	N	NA
SHA256MSG1/SHA256MSG2	1	NA	3	NA	N	NA
SHA256RNDS2	4	NA	7	NA	N	NA

注意:

1. DIVPD/DIVPS/DIVSD/DIVSS では、最初に最速値、次に一般的なケースの値を示しています。最速値は QNAN などの特殊な入力値の場合に適用されます。一般的なケースは、通常の数値の場合に適用されます。
2. ムーブの排除が適用される場合のスループットは 0.33 サイクルですが、それ以外は 0.5 サイクルです。
3. ECX/EFLAGS/XMM0 のレイテンシー値は依存性によるものです: (C/F/X)



## Knights Landing<sup>†</sup> マイクロアーキテクチャーとソフトウェアの最適化

インテル® Xeon Phi™ プロセッサ 7200/5200/3200 製品ファミリーは、Knights Landing<sup>†</sup> マイクロアーキテクチャーをベースにしています。この章では、Knights Landing<sup>†</sup> マイクロアーキテクチャーを対象としたソフトウェアのコーディング手法について説明します。Knights Landing<sup>†</sup> マイクロアーキテクチャーベースのプロセッサは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3C』の第 2 章の表 2-1 に記載される CPUID の DisplayFamily\_DisplayModel シグネチャーを使用して検出することができます。

この章は、Knights Landing<sup>†</sup> マイクロアーキテクチャーの概要から説明を始めます。概要に続いて、Knights Landing<sup>†</sup> マイクロアーキテクチャー上で実行するソフトウェアのパフォーマンスに影響するいくつかの重要なトピックをカバーします: インテル® AVX-512 命令、メモリー・サブシステム、マイクロアーキテクチャー固有の手法、コンパイラ・オプションとディレクティブ、数値シーケンス、MCDRAM キャッシュ、およびスカラーとベクトルコーディング。

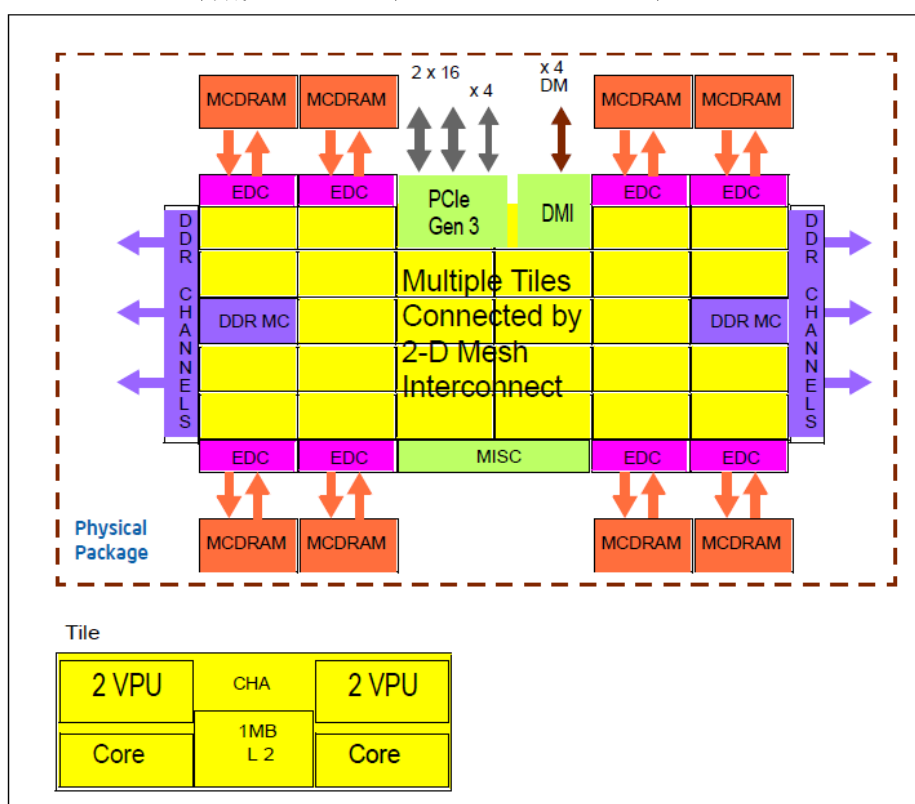


図 17-1 Knights Landing<sup>†</sup> マイクロアーキテクチャーのタイルメッシュトポロジー

### 17.1 Knights Landing<sup>†</sup> マイクロアーキテクチャー

Knights Landing<sup>†</sup> マイクロアーキテクチャーは、高度に並列化されたハイパフォーマンス・アプリケーション向けのプロセッサおよびコプロセッサ製品ファミリーとして設計されました。Knights Landing<sup>†</sup> マイクロアーキテクチャーベースのインテル® Xeon Phi™ プロセッサは以下を含みます。

- Y 多数のタイル。
- Y 2 次元 (2D) メッシュ・インターコネクトで接続されたタイル。
- Y IA 互換プロセッサ・コアとキャッシュ階層を含むすべてのタイルにデータを提供する高度なメモリー・サブシステム。

図 17-1 は、2 次元メッシュ・ネットワークで接続された「タイル」ユニット (プロセッサ・コアのペア) の集合、PCIe\* と DMI インターフェイスを介した I/O、高帯域幅の最適化された MCDRAM をサポートするメモリー・サブシステム、および容量で最適化された DDR メモリー・チャンネルを示しています。

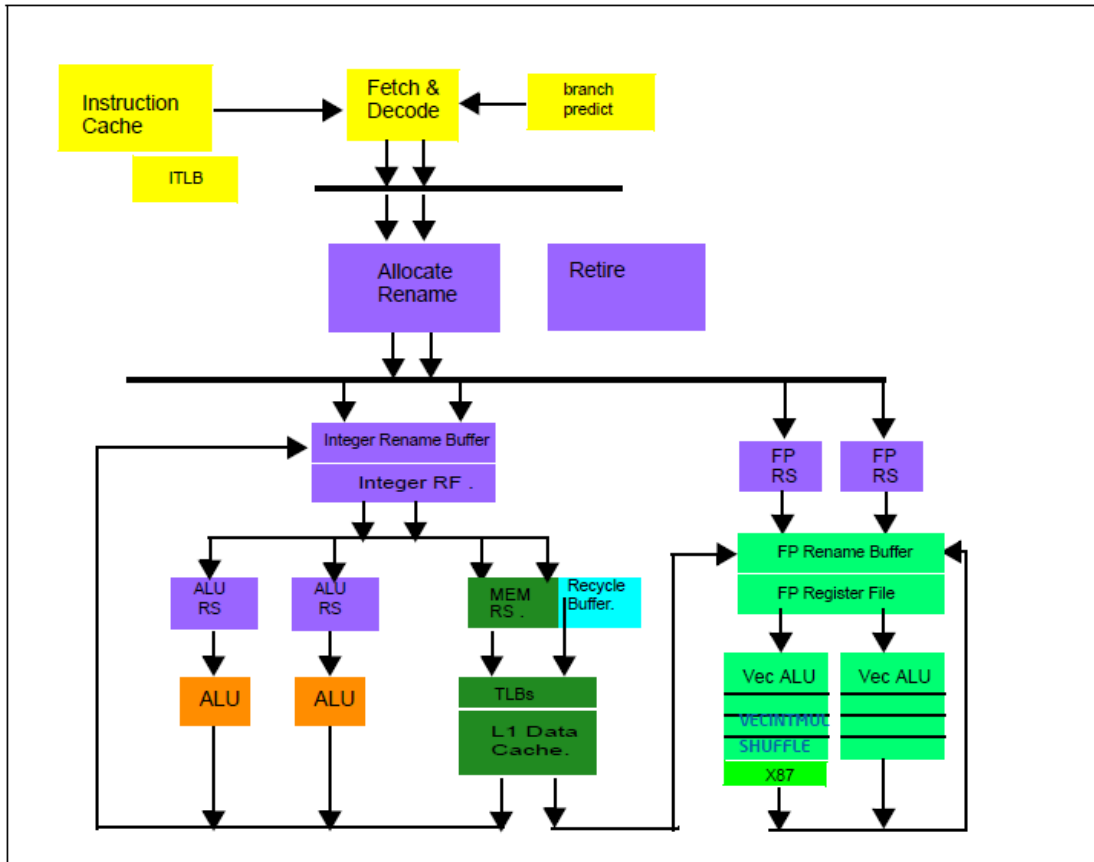


図 17-2 Knights Landing<sup>†</sup> マイクロアーキテクチャーのプロセッサ・コア・パイプラインの機能

図 17-1 はまた、各タイルの構成を示します。

- Y コアあたり 4 つの論理プロセッサのインテル® ハイパースレディング・テクノロジーをサポートする 2 つのアウトオブオーダー IA プロセッサ・コア。
- Y タイル内の 2 つのプロセッサ・コアで共有される 1M バイトの L2 キャッシュ。
- Y 2D メッシュ・インターコネクトへ各タイルを接続するキャッシング・ホーミング・エージェント (CHA)。
- Y 各プロセッサ・コアはまた、512 ビット、256 ビット、128 ビットのベクトルと、スカラー SIMD 命令を処理するベクトル処理ユニット (VPU) を提供します。

図 17-2 は、プロセッサ・コア内部のタイルのパイプライン (VPU パイプラインを含む) のマイクロアーキテクチャー構造を示します。

Knights Landing<sup>†</sup> マイクロアーキテクチャーのプロセッサ・コアは、次の機能を持ちます。

- Y 6 ワイド実行パイプライン (2 VPU, 2 メモリー, 2 整数) のアウトオブオーダー (OOO) 実行エンジン。具体的には、アウトオブオーダー・エンジンは次の機能によってサポートされます。
  - フロントエンドは、サイクルあたり 2 つの命令をマイクロ op (μop) にデコードできます。
  - 2 ワイドのアロケート/リネームステージ。
  - アウトオブオーダー・エンジンは、整数、メモリー、および VPU パイプラインにフィードするリザーベーション・ステーション (72 エントリー) で分配されます。
- Y VPU は、インテル® AVX-512F、インテル® AVX-512CD、インテル® AVX-512ER、インテル® AVX-512PF、インテル® AVX、および 128 ビット SIMD/FP 命令を実行できます。

- Y VPU は、サイクルごとに 2 つの 512 ビット FMA 操作を行うことができ、x87 とインテル® MMX® 命令のスループットはサイクルごとに 1 つに制限されます。
- Y 各プロセッサ・コアは、インテル® ハイパースレディング・テクノロジーにより 4 つの論理プロセッサをサポートします。
- Y 2 つのプロセッサ・コアが、1M バイトの L2 キャッシュを共有してタイルを構成します。

### 17.1.1 フロントエンド

フロントエンドは、サイクルごとに命令の 16 バイトをフェッチできます。デコーダーは、サイクルごとに 24 バイト以下の 2 つの命令をデコードできます。デコーダーは、命令ごとに 1 つの  $\mu\text{op}$  を供給できます。命令が複数の  $\mu\text{op}$  にデコードされる場合 (VSVATTER\* など)、マイクロコード・シーケンサー (MS) がデコーダーの命令のアライメントと MS フローの長さによって 3 ~ 7 サイクルのパフォーマンス・バブルで後続の  $\mu\text{op}$  を供給します。デコーダーは、分岐する分岐命令 (taken branch) に遭遇すると、若干の遅延が生じます。3 つ以上のプリフィックスを持つ命令は、複数サイクルバブルの原因になります。

フロントエンドは、アロケーション、リネーミング、およびリタイアメント・クラスターを介して OOO 実行エンジンに接続されています。 $\mu\text{op}$  のスケジューリングは、整数、メモリー、および VPU パイプラインにまたがって分散されたりザベーション・ステーションによって扱われます。

### 17.1.2 アウトオブオーダー・エンジン

リオーダーバッファ (ROB) は、72  $\mu\text{op}$  の深さを持ちます。16 個のストアバッファ (アドレスとデータの両方向け) を備えています。 $\mu\text{op}$  の分散スケジューリングは以下を含みます (図 17-2 参照)。

- Y 2 つの整数リザベーション・ステーション (ディスパッチ・ポートごとに 1 つ) は、それぞれ 12 エントリーです。
- Y 1 つの MEC リザベーション・ステーションは 12 エントリーであり、サイクルごとに最大 2  $\mu\text{op}$  をディスパッチします。
- Y 2 つの VPU リザベーション・ステーション (ディスパッチ・ポートごとに 1 つ) は、それぞれ 20 エントリーです。

リザベーション・ステーション、ROB、およびストア・データ・バッファは、論理プロセッサごとにハードウェアでパーティション化されています (プロセッサ・コアが処理する 1、2、または 4 つのアクティブな論理プロセッサ数に依存します)。リソースのハードウェア・パーティションは、論理プロセッサがウェイクアップおよびスリープするたびに变化します。ストア・アドレス・バッファは、論理プロセッサごとに 2 つのエントリーが予約され、残りのエントリーは論理プロセッサ間で共有されます。

整数リザベーション・ステーションは、それぞれサイクルごとに 2  $\mu\text{op}$  をディスパッチでき、アウトオブオーダーで動作します。メモリー実行リザベーション・ステーションは、スケジューラーからインオーダーで 2  $\mu\text{op}$  ディスパッチされますが、任意の順番で完了できます。データキャッシュは、サイクルごとに 2 つの 64B キャッシュラインの読み込みと 1 キャッシュラインの書き込みが可能です。VPU リザベーション・ステーションは、サイクルごとにそれぞれ 2  $\mu\text{op}$  ディスパッチでき、アウトオブオーダーで完了します。

Knights Landing<sup>†</sup> マイクロアーキテクチャーの OOO エンジンは、レイテンシーよりもスループットを優先した実行に最適化されています。整数レジスターへのロード (RAX など) は 4 サイクルで、VPU レジスターへのロード (XMM0、YMM1、ZMM2 または MM0) は 5 サイクルです。整数レジスターへのロードは、サイクルごとに 1 つに制限されますが、その他のメモリー操作 (ストアアドレス、ベクトルロード、プリフェッチ) は、サイクルごとに 2 つ ディスパッチできます。リタイアメント後のストアのコミットは、サイクルごとに 1 つの割合で行われます。データキャッシュと命令キャッシュのサイズは、それぞれ 32KB です。

最も一般的に使用される整数算術命令 (add、sub、cmp、test など) は、1 サイクルのレイテンシーでサイクルごとに 2 つのスループットです。整数パイプラインには 1 つの整数乗算器しかないため、オペランドサイズにより 3 もしくは 5 サイクルのレイテンシーがあります。整数除算器のレイテンシーは、オペランドサイズと入力値に依存します。スループットは、1 命令あたり 20 サイクルより高速ではないと予想されます。ストアフォワードの制限に遭遇し

た場合、ロードフォワードへのストアには 2 サイクルのコストが掛かり、サイクルごとに 1 つフォワードできます (表 17-1 を参照)。

表 17-1 Knights Landing<sup>†</sup> マイクロアーキテクチャーのベクトル・パイプラインの特徴

Integer Instruction/operations	Latency (cycle)	Throughput ( cycles per instruction)
Simple Integer	1	0.5
Integer Multiply	3 or 5	1
Integer Divide	Varies	> 20
Store to Load Forward	2	1
Integer Loads	4	1

多くの VPU 数値操作は、どちらかの VPU ポートへ 2 または 6 サイクルのレイテンシーでディスパッチできます (表 17-2 を参照)。次の命令は、単一のポートへのみディスパッチできます。

- ÿ すべての x87 数値操作。
- ÿ FP 除算または平方根。
- ÿ インテル® AVX-512ER。
- ÿ ベクトル permute/shuffle 操作。
- ÿ ベクトルから整数への移動。
- ÿ インテル® AVX-512CD 競合命令。
- ÿ インテル® AES-NI。
- ÿ ストア・セマンティクスがあるベクトル命令のストアデータ操作。

上記の操作は、2 つの VPU ディスパッチ・パイプの一方に制限されます。ベクトル・ストア・データとベクトルから整数への移動は、1 つのディスパッチ・パイプで実行されます。残りの単一パイプ命令は、その他のディスパッチ・パイプで実行されます。

表 17-2 Knights Landing<sup>†</sup> マイクロアーキテクチャーのベクトル・パイプラインの特徴

Vector Instructions	Latency (cycle)	Throughput (cycles per instruction)
Simple Integer	2	0.5
Most Vector Math (including FMA)	6	0.5
Mask Instructions (operating on opmask)	2	0.5
AVX-512ER (64-bit element)	7	2
AVX-512ER (32-bit element)	8	3
Vector Loads	5	0.5
Store to Load Forward	2	0.5
Gather (8 elements)	15	5
Gather (16 elements)	19	10
Register Move (GPR -> XMM/YMM/ZMM)	2	1
Register Move (XMM/YMM/ZMM -> GPR)	4	1
DIVSS/SQRTSS <sup>1</sup>	25	~20
DIVSD/SQRTSD <sup>1</sup>	40	~33
DIVP*/SQRTP* <sup>1</sup>	38	~10
Shuffle/Permute (1 source operand) <sup>1</sup>	2	1
Shuffle/Permute (2 source operands) <sup>1</sup>	3	2
Convert (from/to same width) <sup>1</sup>	2	1
Convert (from/to different width) <sup>1</sup>	6	5
Common x87/MMX Instructions <sup>1</sup>	6	1

**注意:**

- これらの命令を実行する物理ユニットは、VPU 内のユニットの物理レイアウトの影響でスケジュールの遅延が生じることがあります。

さらに、Knights Landing<sup>†</sup> マイクロアーキテクチャーのいくつかの命令は、フロントエンドでは単一の  $\mu\text{op}$  としてデコードされますが、実行には 2 つのオペランドを展開する必要があります。これらの複雑な  $\mu\text{op}$  は、サイクルごとに 1 つのスループットでアロケーションされます。これらの命令の例を以下に示します。

- ÿ POP: 整数ロードデータ + ESP 更新
- ÿ PUSH: 整数ストアデータ + ESP 更新
- ÿ INC: レジスターへの加算 + パーシャルフラグの更新
- ÿ Gather: 2 つの VPU  $\mu\text{op}$
- ÿ RET: JMP + ESP 更新
- ÿ 3 ソースの CALL、DEC、LEA

表 17-3 に Knights Landing<sup>†</sup> マイクロアーキテクチャーのキャッシュリソースの特性リストを示します。

表 17-3 キャッシュリソースの特徴

	Sets	Ways	Latency	Capacity/Comments
uTLB	8	8	1	64 4KB pages (fractured) <sup>1</sup>
DTLB (4KB page)	32	8	4	256 4KB pages
DTLB (2M/4M page)	16	8	4	128 2MB/4MB pages
DTLB (1GB page)	1	16	4	16 1GB pages
ITLB	1	48	4	48 4KB pages (fractured)
PDE	8	4	1	Page descriptors
L1 Data Cache	64	8	4 or 5	32 KB
Instruction Cache	64	8	4	32 KB
Shared L2 Cache	1024	16	13+L1 latency	1 MB

注意:

1.  $\mu$ TLB と ITLB は、4KB メモリー領域のトランスレーション結果のみを保持できます。関連するページが 4KB 以上 (2MB や 1GB) の場合、バッファはアクセスされるページの部分的なトランスレーション結果を保持します。この部分的なトランスレーションをフラクチャー・ページ (fractured page) と呼びます。

### 17.1.3 アンタイトル

Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、多くのタイトルがメッシュ・インターコネクトを介して物理パッケージへ接続されています (図 17-1 を参照)。メッシュと関連するオンパッケージのコンポーネントは“アンタイトル”と呼ばれます。各メッシュストップは、タイトルと特定のキャッシュラインを保持する L2 キャッシュの識別子であるタグ・ディレクトリーに接続されます。物理パッケージ内に共有 L3 キャッシュはありません。タイトルでミスしたメモリーアクセスは、メッシュを介して他のタイトルに複製されているキャッシュを特定するため、タグ・ディレクトリーを調べる必要があります。キャッシュ・コヒーレンスには MESIF プロトコルが使用されます。他のタイトルにキャッシュラインが存在しない場合、メモリーへ要求が送られます。

MCDRAM は、オンパッケージの高帯域幅メモリー・サブシステムであり、読み込みトラフィックのピーク帯域幅を提供しますが、書き込みトラフィックは (読み込みと比較して) 低い帯域幅です。MCDRAM は、オフパッケージのメモリー・サブシステム (DDR メモリーなど) より高い総帯域幅を提供します。DDR メモリーの帯域幅は、単独の書き込みや読み込みでは飽和する可能性があります。MCDRAM が達成可能な帯域幅は、読み込みと書き込みトラフィックの混在に依存しますが、DDR と同じことを行った場合およそ 4x - 6x です。

Knights Landing<sup>†</sup> マイクロアーキテクチャー でサポートされる MCDRAM の容量は、8GB または 16GB のいずれかで製品仕様に依存します。MCDRAM のピーク帯域幅は、搭載されているサイズによって異なります。MCDRAM は、DDR に比べて高い帯域幅を持っていますが、容量は少なくなります。Knights Landing<sup>†</sup> マイクロアーキテクチャーの DDR の最大容量は 384GB です。

プラットフォームの物理メモリーは MCDRAM と DDR の両方を包括します。そして、それらは複数の異なるオペレーション・モードでパーティション化できます。良く使用されるモードを次に示します。

- Y キャッシュモード: ダイレクト・マップ・キャッシュの MCDRAM と DDR は、ソフトウェアによってアドレス指定可能なシステムメモリーとして使用されます。
- Y フラットモード: MCDRAM と DDR は、個別にアクセス可能なシステムメモリーにマップされます。
- Y ハイブリッド・モード: MCDRAM はパーティション化され、その一部はダイレクト・マップ・キャッシュとして、そして残りの MCDRAM は直接アドレス指定可能です。DDR はアドレス指定可能なシステムメモリーにマップされます。

タイトル、タグ・ディレクトリー、そしてメッシュの構成は、キャッシュ・コヒーレント・トラフィックのため、次のクラスター操作モードをサポートします。

- Y 全体全 (All-to-All): コア、タグ・ディレクトリーそしてメモリー・コントローラーのキャッシュライン要求は、メッシュのどこでも行うことができます。
- Y 4 分割 (Quadrant): タグ・ディレクトリーとメモリー・コントローラーは同じメッシュの 4 分割をモニターし、メッシュ内のどのコアも要求することができます。
- Y サブ NUMA クラスタ (SNC): SNC モードでは、BIOS は各 4 分割を NUMA ノードに見えるように扱います。これには、ソフトウェアが NUMA ドメインを認識し、最適なキャッシュミス・レイテンシーを実現するため、メッシュの同じ 4 分割において要求コア、タグ・ディレクトリー、およびメモリー・コントローラーを検出する必要があります。

アプリケーションのワーキングセットのクリティカルな領域が MCDRAM の容量にうまく収まる場合、それらを MCDRAM に配置してフラットまたはハイブリッド・モードを使用することで大きな利益が得られる可能性があります。一般にキャッシュモードは、Knights Landing<sup>†</sup> 向けにまだ最適化されておらず、ワーキングセットが MCDRAM にキャッシュできるようなコードに最も適しています。

一般に全体全モードのキャッシュミス・レイテンシーは、4 分割モードよりも長くなります。SNC モードは最良のレイテンシーを達成できます。4 分割モードはデフォルトのメッシュ構成です。SNC クラスタリングは、異なる NUMA ノードを認識するためソフトウェアから何らかのサポートを必要とします。DDR が均等に装着されていない場合 (DIMM の損失や空きなど)、メッシュは全体全クラスタリング・モードを使用する必要があります。

複数のタイルが同じキャッシュラインを読み込むと、各タイルがキャッシュラインのコピーを保持する可能性があります。同じタイル内の両方のコアがキャッシュラインを読み込むと、タイルの L2 には単一のキャッシュラインのみが保持されます。

MCDRAM がキャッシュとして構成されている場合、単一の位置でコアからアクセスされる命令とデータを保持できます。複数のタイルが同じキャッシュラインを要求すると、MCDRAM のキャッシュラインは 1 つだけ使用されます。

L1 データキャッシュは、L2 キャッシュよりも高い帯域幅と低いレイテンシーを持ちます。L2 からのキャッシュライン・アクセスは、メモリーと比べると高い帯域幅と低いレイテンシーを持ちます。

MCDRAM と DDR メモリーのレイテンシーとスループットのプロファイルは異なります。これは、キャッシュ vs フラット、またはその他のメモリーモードを選択する際に重要となります。ほとんどのメモリー構成では、DDR の容量は MCDRAM の容量よりも大幅に大きくなります。同様に、MCDRAM の容量はすべての L2 キャッシュの容量よりも大きくなります。

ワーキングセットが L2 キャッシュに収まらない場合、MCDRAM に格納する必要があります。大量のまたはアクセス頻度が低いデータ構造は、DDR に格納します。MCDRAM が“キャッシュ”または“ハイブリッド”モードに設定されている場合、Knights Landing<sup>†</sup> マイクロアーキテクチャーのハードウェアはこれを動的に行います。メモリーが“フラット”メモリーモードの場合は、データ構造は 1 つのメモリーに結合されるか、他 (MCDRAM や DDR) に割り当てられます。プログラマーは、MCDRAM へのメモリーアクセス数が最大となるよう努めるべきです。あるアルゴリズムがデータ構造を MCDRAM に配置し、それらが頻繁にアクセスされるようであれば、ワーキングセットはキャッシュに収まりきらないと考えられます。

キャッシュモードでは、最初に MCDRAM をアクセスします。キャッシュラインが MCDRAM に存在しなければ、DDR へのアクセスが開始されます。そのため、“キャッシュ”メモリーモードにおける DDR の平均アクセス・レイテンシーは、“フラット”メモリーモードよりも高くなります。

## 17.2 Knights Landing<sup>†</sup> マイクロアーキテクチャー向けのインテル® AVX-512 コーディングの推奨事項

インテル® AVX-512 ファミリーは、いくつかの命令セット拡張を包括しています。インテル® AVX-512 ファミリーの命令の詳細 (EVEX プリフィクス・エンコード、opmask サポートなど) と概要については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 1』をご覧ください。Knights Landing<sup>†</sup> マイク

IA-32 アーキテクチャーをベースとするインテル® Xeon Phi™ プロセッサ 7200/5200/3100 製品ファミリーは、インテル® AVX-512 基本命令 (インテル® AVX-512F)、インテル® AVX-512 指数および逆数 (インテル® AVX-512ER)、インテル® AVX-512 競合検出 (インテル® AVX-512CD)、およびインテル® AVX-512 プリフェッチ拡張をサポートしています。Knights Landing<sup>†</sup> マイクロアーキテクチャーをベースのプロセッサでは、インテル® AVX とインテル® AVX2 命令もサポートされます。以前の世代のインテル® Xeon Phi™ コプロセッサ 7100/5100/3100 製品ファミリーでは、インテル® AVX-512、インテル® AVX2 またはインテル® AVX 命令はサポートされていません。

### 17.2.1 Gather および Scatter 命令の利用

インテル® AVX-512F の Gather 命令は、インテル® AVX2 の Gather 命令を拡張し、512 ビット操作を行い (32 ビット・データを 16 要素または 64 ビット・データを 8 要素)、書き込みマスクとして opmask レジスターを使用して、要素を条件付きでデスティネーション ZMM レジスターに更新します。

インテル® AVX-512F の Scatter 命令は、ZMM レジスターの要素を選択的にインデックス・ベクトルで表現されるメモリー位置へストアします。デスティネーションへの条件付きストアは、opmask レジスターを使用して選択されます。インテル® AVX やインテル® AVX2 では、Scatter 命令はサポートされていません。

次のようなコードについて考えてみます。

```
for (uint32 i = 0; i < 16; i ++ ) {
    b[i] = a[indirect[i]];
    // ベクトル計算のシーケンス
}
```

例 17-1 インテル® AVX-512F とインテル® AVX2 における Gather の比較

AVX-512F	AVX2
vmovdqu zmm0, [rsp+0x1000]; load indirect[]	vmovdqu ymm0, [rsp+0x1000]; load half of index vector
kxnor k1, k0, k0; prepare mask	vmovdqu ymm3, [rsp+0x1020]; 2nd half of indirect[]
vpgatherdd zmm2{k1}, [rax+zmm0*4]	vpcmpqdd ymm4, ymm4, ymm4; prepare mask
; compute sequence using vector register	vmovdqa ymm1, ymm4
	vpgatherdd ymm2, [rax+ymm0*4], ymm1
	vpgatherdd ymm5, [rax+ymm3*4], ymm4
	; compute sequence using vector register

VGATHER と VSCATTER を使用する場合、多くのケースですべてのマスクを 1 に設定する必要があります。これを効率良く行う命令は、自身をマスクレジスターとする KXNOR です。VSCATTER と VGATHER は、動作の最後にマスクをクリアするため、VGATHER から KXNOR ヘルプ伝搬依存を生成します。そのため、KXNOR のソースとデスティネーションに同じマスクを使用しないようにすることが賢明です。k0 マスクがデスティネーションとして使用されることはまれであるため、“KXNORW k1, k0, k0” が “KXNOR k1, k1, k1” よりも高速である可能性があります。

インテル® AVX-512F の Gather と Scatter は、前の世代のインテル® Xeon Phi™ コプロセッサ (例 17-2 では「KNC<sup>+</sup>」と表現されています) とは異なります。

例 17-2 インテル® AVX-512F と KNC<sup>+</sup> における Gather の比較

AVX-512F	KNC-Equivalent Sequence
vmovdqu zmm0, [rsp+0x1000]; load indirect[]	vmovdqu zmm0, [rsp+0x1000]; load indirect[]
kxnor k1, k0, k0; prepare mask	kxnor k1, k1; prepare mask
vpgatherdd zmm2{k1}, [rax+zmm0*4]	g_loop: ; verify gathered elements are complete
; compute sequence using vector register	vpgatherdd zmm2{k1}, [rax+zmm0*4]
	jkznd k1, g_loop; gather latency exposure
	; compute sequence using vector register



## 17.2.2 拡張された逆数命令の利用

インテル® AVX-512ER 命令は、高精度の指数、逆数、および逆平方根の近似を可能にします。インテル® AVX-512ER における近似数学命令は、RCPSS の 11 ビットや VRCP14SS の 14 ビットに対し、28 ビットの精度を提供します。インテル® AVX-512ER は、ニュートンラプソンなどの反復法の実行時間を短縮します。図 15 に示すサンプルコードは、VRCP28SS 命令と除算を使用し単一の 32 ビット単精度浮動小数点を計算するため、ニュートンラプソン法を使用しています。両方の値はスタックから読み込まれます。いくつかの算術計算では、丸めモードのオーバーライドに注意してください。

例 17-3 32 ビット浮動小数点除算に VRCP28SS を使用する

```
vgetmantss  xmm18, xmm18, [rsp+0x10], 0
vgetmantss  xmm20, xmm20, [rsp+0x8], 0
vrpc28ss    xmm19, xmm18, xmm18
vgetexpss   xmm16, xmm16, [rsp+0x8]
vgetexpss   xmm17, xmm17, [rsp+0x10]
vsubss     xmm22, xmm16, xmm17
vmulss     xmm21{rne-sae}, xmm19, xmm20
vfmadd231ss xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss xmm21, xmm19, xmm20
vscalefss  xmm0, xmm21, xmm22
```

## 17.2.3 インテル® AVX-512CD 命令の利用

インテル® AVX-512CD 命令は、特定の種類のアクセスパターンにおいて効率良いベクトル化を可能にします。良く利用されるパターンの 1 つに、“ヒストグラムの更新”として特徴付けられるものがあります。これは、メモリー位置が読み込まれ、分散セットに変換後、更新された部分がメモリーに書き戻されます。このアクセスパターンの C コードの例の一部を示します。

```
for (i=0; i < 512; i++) histo[key[i]] += 1;
```

上記の C コードをベクトル化する強引な方法では、配列 key[] をインデックスとしてギャザーを行い、vpadd でレジスター内の分布を更新して、更新された分布をメモリーへストアするためスカッターを試みることがあります。この方法では、レイテンシーによるパフォーマンスの問題に加え、ZMM レジスターにのギャザーされた 16 要素内の重複の衝突を考慮して適切に処理しないと、正当性は保証されません。

インテル® AVX-512CD 命令は、このようなケースを検出してループを正しくベクトル化することを可能にします。例 17-4 に、前述のヒストグラム更新をベクトル化するコンパイラーが生成したメインループの逆アセンブリー・コードを示します。

例 17-4 インテル® AVX-512CD を使用したベクトル化されたヒストグラムの更新

```
Top:
vmovups    zmm4, [rsp+rdx*4+0x40]
vpxord     zmm1, zmm1, zmm1
kmovw     k2, k1
vpconflict zmm2, zmm4
vpgatherdd zmm1{k2}, [rax+zmm4*4]
vptestmd  k0, zmm2, [rip+0x185c]
```

```

kmovw    ecx, k0
vpadd    zmm3, zmm1, zmm0
test     ecx, ecx
jz       <No_conflicts>
vmovups  zmm1, [rip+0x1884]
vptestmd k0, zmm2, [rip+0x18ba]
vplzcntd zmm5, zmm2
xor      bl, bl
kmovw    ecx, k0
vpsubd   zmm1, zmm1, zmm5
vpsubd   zmm1, zmm1, zmm5
Resolve_conflicts:
vpbroadcastd zmm5, ecx
kmovw    k2, ecx
vpermd   zmm3{k2}, zmm1, zmm3
vpadd    zmm3{k2}, zmm3, zmm0
vptestmd k0{k2}, zmm5, zmm2
kmovw    esi, k0
and      ecx, esi
jz       <No_conflicts>
add      bl, 0x1
cmp      bl, 0x10
jb       <Resolve_conflicts>
No_conflicts:
kmovw    k2, k1
vpscatterdd [rax+zmm4*4]{k2}, zmm3
add      edx, 0x10
cmp      edx, 0x400
jb       <Top>

```

### 17.2.4 インテル® ハイパースレッディング・テクノロジーの利用

Knights Landing<sup>†</sup> マイクロアーキテクチャーは、それぞれのプロセッサ・コアで 4 つの論理プロセッサをサポートします。高度にスレッド化されたソフトウェアでは、次のことを考慮する必要があります。

- Y 論理プロセッサごとにコアのリソースを最大化することでスレッドのパフォーマンスを最大化します。
- Y プロセッサ・コアで複数の論理プロセッサの実行を可能にすることで、コアごとのスループットを最大化します。

コアあたりのスレッド数を 2 または 4 にすると、ほとんどのアプリケーションは最も高いパフォーマンスを発揮しますが、スレッドごとのパフォーマンスは低下します。アプリケーションが任意のスレッド数でパフォーマンスを完璧にスケールすることができるのであれば、コアあたり 4 スレッドで最も高い命令スループットを持つ可能性があります。コアあたりのスレッド数を増やすと、メモリー容量や並列処理の制約によりスケールングの効率が制限される可能性があります。

Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、いくつかのコアリソース (ROB やスケジューラーなど) は、4 つの論理プロセッサごとにパーティション化されます。そのため、3 スレッドの構成では、スレッドが利用できる内部リソースがコアごとに 1、2、または 4 スレッドの構成と比べ最も少なくなります。プロセッサ・コアに 3 スレッドを配置しても、2 または 4 スレッドほど良い結果は得られないでしょう。

### 17.2.5 フロントエンドに関する考察

フロントエンドの制限がパフォーマンスを損ねないことを保証するため、ソフトウェアは次のことを考慮すべきです。

- Y MSROM 命令は可能な限り回避します。典型的な例は CALL near の間接メモリー形式です。メモリーバージョンの PUSH と CALL の代わりに、レジスターヘロードし、レジスターバージョンの PUSH と CALL を実行します。表 17-4 にいくつかの例を示します。
- Y サイクルごとにデコードできる命令バイトの合計長は、最大 16 バイトでそれぞれの命令の命令長は 8 バイト以下です。例えば、命令が 8 バイトを超えるとデコーダー 0 では、サイクルあたり 1 命令しかデコードできません。32 ビット・ディスプレイメントを使用するメモリーアドレスを持つベクトル命令は、デコーダーでパフォーマンスが制限されます。
- Y 複数のプリフィクスを持つ命令は、デコーダーのスルーポットを制限します。プリフィクスとエスケープの合計バイト数が制限に当てはまります。Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、命令のプリフィクス/エスケープが 3 つを超えると、3 サイクルのペナルティーが発生します。デコーダー 0 のみがプリフィクス/エスケープ・バイトの制限を超えた命令をデコードできます。
- Y 各サイクルでデコード可能な分岐の最大数は 1 です。

### 17.2.5.1 命令デコーダー

一部の IA 命令は、複数のマイクロオペレーション (μop) フローにデコードするため、マイクロコード・シーケンス ROM (MSROM) のルックアップが必要になります。MSROM を必要としない代替命令シーケンスを選択すると、パフォーマンスが向上します。

表 17-4 に、MSROM からデコードされる命令を置き換えることができる非 MSROM 命令のシーケンスを示します。

表 17-4 MSROM の代替命令

Instruction from MSROM	Recommendation for Knights Landing
CALL m16/m32/m64	Load + CALL reg
PUSH m16/m32/m64	Store + RSP update
(I)MUL r/m16 (Result DX:AX)	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32
(I)MUL r/m32 (Result EDX:EAX)	Use (I)MUL r32, r/m32 if extended precision not required, or (I)MUL r64, r/m64
(I)MUL r/m64 (Result RDX:RAX)	Use (I)MUL r64, r/m64 if extended precision not required

### 17.2.5.2 4GB 境界を超える間接分岐

フロントエンドの観点から考慮すべき他の重要なパフォーマンス事項は、間接分岐 (間接分岐や call または ret) に対する分岐予測です。64 ビット・アプリケーションでは、分岐命令があるアドレス空間と異なる 4GB チャンクに分岐先があると (つまり、分岐命令と分岐先の仮想アドレスの上位 32 ビットが異なる場合)、間接分岐予測が失敗します。これは、アプリケーションが共有ライブラリーと分離されている場合に発生する可能性があります。レイテンシーに敏感なライブラリー呼び出しが頻繁に行われる場合、プログラマーは、コードの局所性を改善するため静的にビルドすることもできます。他の選択肢は glibc 2.23 以降を使用して、LD\_PREFER\_MAP\_32BIT\_EXEC 環境変数を設定することで、動的リンカーにすべての共有ライブラリーをアドレス空間の下位に配置することです。

## 17.2.6 整数実行に関する考察

### 17.2.6.1 フラグの利用

多くの命令には、フラグレジスターに格納される暗黙のデータがあります。これらのデータは、条件移動 (CMOVS)、分岐、さまざまな論理/算術演算 (RCL など) といった幅広い命令で利用されます。分岐条件としてよく使用される命令に比較命令 (CMP) があります。CMP 命令に依存する分岐は次のサイクルで実行できます。ADD 命令や SUB 命令に依存する分岐でも同様のことが言えます。

INC 命令と DEC 命令は一部のフラグのみ設定するため、フラグをマージする追加のマイクロオペレーション (μop) が必要になります。そのため、INC や DEC 命令は、パーシャルフラグ・ストールを避けるため、“ADD reg, 1” や “SUB reg, 1” に置き換えるべきです。

Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、8 ビットまたは 16 ビットレジスターを操作する命令はハードウェアで最適化されていません。一般に、整数命令は 32 ビットや 64 ビット汎用レジスターオペランドを操作する方が、8 ビットや 16 ビットレジスターを操作するよりも高速です。

## 17.2.6.2 整数除算

整数除算は、いくつかの数式では良く使用される操作です。しかし、ハードウェア整数除算命令の使用は、しばしばパフォーマンス上適切ではないことがあります。整数値が比較的小さいことが判明している場合 (16 ビット以下)、代替となる除算をエミュレートする高速なソフトウェア・シーケンスが知られています。除数が 2 の累乗である場合、DIV 命令の代わりに SHR (除数) と/もしくは AND (余り) 命令を使用します。定数による除算は、MUL と定数に置き換えることができます。収束しない入力値の場合、事前計算されたルックアップ・テーブルを使用することでパフォーマンスが向上する可能性があります。10.2.4 節「128 ビット整数除算の 128 ビット乗算への置換」と 11.5 節「ASCII 形式への数値データの変換」に、いくつかの手法の例が示されています。

コンパイラーは前述の手法を使用するか、内部ループから冗長な除算をホイストして積極的に除算命令を最小化するべきです。

## 17.2.7 FP およびベクトル実行の最適化

### 17.2.7.1 命令選択の注意事項

一般に、512 ビット命令の使用は、256 ビット命令よりも高いスループットを達成するには好都合です。これは、同様に 256 ビットと 128 ビットのベクトル命令にも当てはまります。128 ビットのインテル® SSE 命令は、等価な x87 命令を使用するよりも高いスループットを達成します。ベクトル命令拡張では x87 命令の機能 (超越関数) が提供されないことがありますが、これはベクトル命令を使用したライブラリー実装に置き換えることができます。

Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、COMIS\* と UCOMIS\* 命令 (レガシー、VEX または EVEX エンコード) は EFLAGS を更新するため低速です。これらの命令は、インテル® AVX-512F バージョンの VCMPS\* と KORTEST のより適切なシーケンスに置き換えるべきです。

#### 例 17-5 VCOMIS\* を VCMPS/KORTEST に置き換える

```
vcmpss k1, xmm1, xmm2, imm8; specify imm8 according to desired primitive
kortest k1, k1
```

VCOMPRESS\* などの一部の命令は、レジスターに書き込む場合は単一 μop ですが、メモリーに書き込む場合は MS フローになります。可能な限り、高速なレジスターへのストアを行う VCOMPRESS を使用してください。同様の最適化は、何らかの操作を行ってストアを実行するすべてのベクトル命令に適用されます。

Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、インテル® SSE 命令とインテル® AVX 命令の混在はパフォーマンスの損失を避けるため特別な考慮が必要です。可能な限り、インテル® SSE コードを等価な 128 ビットのインテル® AVX に置き換えます。

次の条件でパフォーマンス上のペナルティーが生じます。

- 128 ビット以上のベクトル長でエンコードされているインテル® AVX 命令が、実行中のインテル® SSE 命令がリタイアする前に割り当てられる場合。
- VZERoupper 命令のスループットは低速です。そのため、インテル® SSE コードが実行された後に

インテル® AVX へ移行することは推奨されません。VZEROALL のスルーブットもまた低速です。ZEROUPPER または VZEROALL 命令のどちらを使用してもパフォーマンスの損失につながります。

MASKMOVDQU と VMASKMOV のような、条件付きパックドロード/ストア命令は、要素の選択にベクトルレジスターを使用します。インテル® AVX-512F 命令は、要素の選択に opmask レジスターを使用した代替手段を提供しますが、要素選択にベクトルレジスターを使用するよりも適しています。

いくつかのベクトル数学命令は実装に VPU で複数の  $\mu\text{op}$  を必要とします。これによる、個々の命令のレイテンシーが、2 または 6 の標準的な計算レイテンシーを上回ります。一般に、出力/入力要素幅を変更する命令 (VCVTSD2SI など) がこのカテゴリーに分類されます。バイトとワード単位で操作を行うインテル® AVX2 命令は、32 ビットや 64 ビット単位で操作する同様の命令と比較するとパフォーマンスが低下します。

VPU のいくつかの実行ユニットは、依存関係のある  $\mu\text{op}$  フローのシーケンスがそれらの実行ユニットを使用する場合スケジューリングに遅延が生じます。これから分離ユニットは表 17-2 の脚注に示されています。この問題が生じると 2 サイクルバブルのコストが加算されます。VPU の分離ユニットと他のユニット間を頻繁に移行するコードは、これらのバブルによるパフォーマンスの問題が生じます。

opmask レジスターを使用する大部分のインテル® AVX-512 命令は、デスティネーションを条件付きで更新します。一般に、すべて 1 の opmask を使用する方が、ゼロ以外の値の opmask を使用するよりも高速です。ゼロ化非更新要素が選択された場合、ゼロ以外の opmask 値を使用すると、すべて 1 の opmask の命令と同等のスピードです。ゼロ以外の opmask 値とデスティネーションの非更新要素がマージされると、低速になります。

インテル® AVX の水平加算/減算命令には、等価なインテル® AVX-512 命令がありません。ソフトウェア・シーケンスを使用した水平リダクションは最良の実装です (例 17-6)。

アルゴリズムがリダクションを必要とする状況で、水平加算を行うことなくリダクションを実装する方法があります。

例 17-7 は、DGEMM 行列乗算ルーチンの内部ループのコードの一部を示し、 $C = A * B$  の密行列計算を行っています。

例 17-7 には 16 個の部分和があります。FMA 命令のシーケンスは、2 つの VPU の能力を利用して、サイクルごとに 2 FMA のスルーブットで、6 サイクルのレイテンシーを達成します。例 17-7 の FMA コードは、メモリーオペランドに圧縮を使用しないアドレス形式を示しています。コード・ジェネレーターが、FMA 命令長が 8 バイト未満になるように圧縮された disp8 アドレス形式を使用して最適なコードを確実に生成することが重要です。内部ループの最後で部分和は集計され、結果はメモリー上の行列 C にストアされます。

例 17-6 水平リダクションのソフトウェア・シーケンスを使用

<pre>vextractf64x4 ymm1, zmm6, 1; reduction of 16 elements vaddps  ymm1, ymm6, ymm1 vpermpd ymm4, ymm1, 0xff vpermpd ymm5, ymm1, 0xaa vpermpd ymm3, ymm1, 0x44 vaddps  xmm1, xmm1, xmm4 vaddps  xmm3, xmm5, xmm3 vaddps  xmm3, xmm1, xmm3 vpsrlq  xmm1, xmm3, 32 vaddss  xmm3, xmm1, xmm3</pre>	<pre>vextractf64x4 ymm1, zmm6, 1; reduction of 8 elements vaddps  ymm1, ymm6, ymm1 valignq ymm4, ymm1, 0x3 valignq ymm5, ymm1, 0x2 valignq ymm3, ymm1, 0x1 vaddsd  ymm1, ymm1, ymm4 vaddsd  ymm3, ymm5, ymm3 vaddsd  ymm3, ymm1, ymm3</pre>
---	---

例 17-7 Knights Landing<sup>†</sup> マイクロアーキテクチャー向けに最適化された DGEMM の内部ループ

```

:: matrix - matrix dense multiplication
prefetcht0 [rdi+0x400] ;; get A matrix element into L1$
vmovapd zmm30, [rdi]
prefetcht0 [rsi+0x400] ;; get B matrix element into L1$
vfmadd231pd zmm1, zmm30, [rsi+r12](b) ;; broadcast B elements
vfmadd231pd zmm2, zmm30, [rsi+r12+0x08](b) ;; displacement shown in un-compressed form
vfmadd231pd zmm3, zmm30, [rsi+r12+0x10](b)
vfmadd231pd zmm4, zmm30, [rsi+r12+0x18](b)
vfmadd231pd zmm5, zmm30, [rsi+r12+0x20](b)
vfmadd231pd zmm6, zmm30, [rsi+r12+0x28](b)
vfmadd231pd zmm7, zmm30, [rsi+r12+0x30](b)
vfmadd231pd zmm8, zmm30, [rsi+r12+0x38](b)

prefetcht0 [rsi+0x440] ;; pull line into the L1$
vfmadd231pd zmm9, zmm30, [rsi+r12+0x40](b)
vfmadd231pd zmm10, zmm30, [rsi+r12+0x48](b)
vfmadd231pd zmm11, zmm30, [rsi+r12+0x50](b)
vfmadd231pd zmm12, zmm30, [rsi+r12+0x58](b)
vfmadd231pd zmm13, zmm30, [rsi+r12+0x60](b)
vfmadd231pd zmm14, zmm30, [rsi+r12+0x68](b)
vfmadd231pd zmm15, zmm30, [rsi+r12+0x70](b)
vfmadd231pd zmm16, zmm30, [rsi+r12+0x78](b)
    
```

### 17.2.7.2 前世代からの組込み関数の移植

ほとんどの組込み関数はネイティブ・ハードウェアの固有の命令に対応しています。512 ビット組込み関数の中には、インテル® AVX-512F と互換性のない KNC<sup>†</sup> 命令セット間の違いを隠匿する構文を提供するものがあります。

しかし、KNC<sup>†</sup> で実行するために最適化された組込み関数のコードは、マイクロアーキテクチャーが異なることから (アライメントされていないメモリアクセス、並べ替えのコストの違い、KNC<sup>†</sup> の制限などにより) Knights Landing<sup>†</sup> マイクロアーキテクチャー上では最適化された状態で動作しません。

KNC<sup>†</sup> 向けの組込み関数を使用してコーディングするよりも、高レベル言語 (C/Fortran) でアルゴリズムを記述して、インテル® AVX-512F をサポートするインテル® コンパイラーでコンパイルする方が最適なコードを生成できます。

### 17.2.7.3 ベクトル化のトレードオフを推定する

高レベル言語で記述されたループのベクトル化でインテル® AVX-512 の使用により利点が得られるかどうかは、コンパイラーおよびアセンブリーによる手動コーディングにおける最適化の重要な部分です。最もシンプルなループ構造を見積もるには、ループカウントだけをベースにします。例えば、ループカウント 4 以下ではスカラーコードを上回るパフォーマンスを得るのは難しいかもしれません。インテル® AVX-512 において、ベクトル化を考慮すべき最小のループカウントは 16 です。つまり 16 以上では利点があります。

複雑なループ構造のベクトル化のトレードオフを見積もるには、多くの考察が求められます。この節の残りのセクションでは、ループ本体の構成を調査する分析アプローチにより、表 17-5 に示す基本的な操作のコスト見積もりを使用してベクトル化とスカラーコードを比較してトレードオフを明らかにします。

表 17-5 Knights Landing<sup>†</sup> マイクロアーキテクチャー向けのベクトル化を見積もる  
サイクル・コスト・ビルディング・ブロック

Operation	Cost (cycles)	Example Code Construct
Simple scalar math	1	A*B+C, or A+B, or A*B
Load (split cacheline)	1 (2)	A[i] /* load reference to an array element */
Store (split cacheline)	1(2)	A[i] = 2;
Gather (Scatter) 8 elements	15 (20)	A[key[i]]
Gather (Scatter) 16elements	20 (25)	A[key[i]];
Horizontal reduction	30	sum += A[i]
Division or Square root	15	A/B

最初に、コスト見積もり方法を説明するため、簡単なループを考えてみます。

```
for (i=0; i<N; i++) { sum += a[i]*K + b[i]; }
```

ループ本体の基本操作は次のように構成されます。

- Y 反復ごとに 2 つのロード (a[i], b[i])、
- Y 反復ごとに 1 つの FMA、
- Y スカラーバージョン: ループ反復ごとの累積、ベクトルバージョン: ループの最後で水平リダクション。

スカラーコードの N 回の総コストは 4N です。一方、メインループとリマインダー・ループ (N が 8 の倍数でない場合) の両方がベクトル化されると想定した場合、64 ビット・データ要素をインテル® AVX-512 を使用してベクトル化したコードの総コストは  $3 * \text{Ceiling}(N/8) + 30$  です。つまり、ベクトル化から利益を得るには少なくとも 9 回以上のトリップカウントが必要です。

GATHER 命令の利点を活用する不規則なアクセスパターンからのデータをフェッチする別の例を考えてみます。

```
for (i=0; i<N; i++) {c[i] = a[indir[i]] * K + b[i]; }
```

ループ本体の基本操作は次のように構成されます。

- Y 反復ごとに 2 つのロード (indir[i], b[i])、
- Y 反復ごとに 1 つの FMA、
- Y 反復ごとに 1 つのストア、
- Y スカラーバージョン: 3 回の反復ごとにロード、ベクトルバージョン: 8 反復ごとに 1 つの GATHER、

スカラーコードの N 回の総コストは 5N です。一方、ベクトル化されたコードの総コストは、 $19 * \text{Ceiling}(N/8)$  です。N < 4 の場合スカラーコードのほうが高速です。

前の例よりも不規則なアクセスパターンからのデータをフェッチする例を考えてみます。

```
for (i=0; i<N; i++) {c[i] = a[ind[i]]*K + b[ind[i]]; }
```

- Y 反復ごとに 1 つのロード (ind[i])、
- Y 反復ごとに 1 つの FMA、
- Y 反復ごとに 1 つのストア、
- Y スカラーバージョン: ループ反復ごとにさらに 2 つのロード、ベクトルバージョン: 8 反復ごとに 2 つの GATHER、

スカラーコード向けの N 回の総コストはまだ 5N です。一方、ベクトル化されたコードの総コストは、 $(15 \cdot 2 + 3) \cdot \text{Ceiling}(N/8) = 33 \cdot \text{Ceiling}(N/8)$  です。わずかなベクトル化の利益を得るのにかなり大きなトリップカウントを必要とします。

次の例では、1 つの不規則なアクセスパターンと水平リダクションからのデータをフェッチする状況を考えてみます。

```
for (i=0; i<N; i++) {sum += a[ind[i]]*K + b[i]; }
```

スカラーのコストはまだ 5N です。ベクトル化のコストは、 $19 \cdot \text{Ceiling}(N/8) + 30$  になりました。N ≤ 13 の場合スカラーコードのほうが高速です。

除算を伴うスカッターを例に考えてみます。

```
for (i=0; i<N; i++) {c[ind[i]] = a[i] / b[i]; }
```

スカラーコストは  $(15+4) \cdot N$  です。ベクトル化のコストは、 $(15+20+3) \cdot \text{Ceiling}(N/8)$  です。N > 2 の場合ベクトル化の利点があります。

スカッターの後にギャザーが続く場合を考えてみます。

```
for (i=0; i<N; i++) {b[ind[i]] = a[ind[i]]; }
```

スカラーコードのコストは  $3 \cdot N$  で、ベクトルコードのコストは  $(15+20+1) \cdot \text{Ceiling}(N/8)$  です。ベクトル化の利点はありません。

miniMD として知られるさらに複雑なループ本体のコードを考えてみましょう。

```
for (int k = 0; k < numneigh; k++) {
    int j = neighs[k];
    double rsq = (xtmp - x[3*j])^2 +
        (ytmp - x[3*j+1])^2 +
        (ztmp - x[3*j+2])^2;
    if (rsq < cutforcesq) {
        double sr2 = 1.0/rsq;
        double sr6 = sr2*sr2*sr2;
        double force = sr6*(sr6-0.5)*sr2;
        res1 += delx*force;
        res2 += dely*force;
        res3 += delz*force;
    }
}
```



if 節について考える前に、1 つのロード、3 つのギャザー (x[] のストライドロード)、3 つの減算、そして 3 つの乗算があります。if 節の内部では、1 つの除算、8 つの数学計算、そして 3 つの水平リダクションが行われています。スカラーのコストは、 $10 * \text{numneigh} + 23 * \text{numneigh} * \text{percent\_rsq\_less\_than\_cutforcesq}$  です。そしてベクトルのコストは、 $(52 + 23) * \text{ceiling}(\text{numneigh} / 8) + 3 * 30$  です。 $(\text{numneigh} < 6)$  が成立するか、コンパイラーが if 節はほとんど実行されないという高い確証があるなら、スカラーコードを使用することは理にかなっています。

多くのコンパイラーは、ベクトル化されたループを生成し、余剰操作を処理するためリマインダー・ループを使用します。言い換えると、ベクトル化されたループは  $\text{floor}(N/8)$  回実行され、リマインダー・ループが  $(N \bmod 8)$  回実行されることとなります。この場合、一次ループをベクトル化するかどうか判断するため、 $\text{ceiling}$  の代わりに  $\text{floor}$  を使用するように変更します。リマインダー・ループがあり、ループの最大カウントが明白である場合、ベクトル幅は 1 つ少なくなります。N が不明である場合、ベクトル幅の半分を N に設定するのがもっとも容易です (倍精度の ZMM ベクトルでは 4)。

より洗練された分析が可能です。例えば、表 17-5 に示す 1 サイクルの簡単なビルディング・ブロックの数学操作は、依存性チェーンや長いレイテンシーの操作によってブロックされない一般的な命令シーケンスです。コスト表の内容を拡大することでより複雑な状況をカバーできます。

## 17.2.8 メモリー最適化

### 17.2.8.1 データ・アライメント

キャッシュライン境界をまたぐアドレスのデータアクセスには、わずかなパフォーマンスの利点があります。メモリーをストリームするアクセスパターンでは、64 バイト・アクセスごとに確実にキャッシュライン境界にアライメントすることで、キャッシュライン分割を避けることができます。メモリーの 32 バイトを YMM ヘロードする場合、opmask 値を使用してメモリーの 64 バイトにアクセスし上位 32 バイトをマスクしてはなりません。

4K バイト境界にまたがるメモリー参照は、パフォーマンス上の高いコストを被ります。512 ビット命令を使用したメモリーをストリームするアクセスパターンのスルーputは、4K バイト境界を超えても高いレートをもたらします。64 バイトにアライメントすると、ページ分割のペナルティーも避けることができます。

ページ境界をまたがった次のコード空間までの距離を推測することが可能な場合、現在の読み込みストリームの数回前の反復で PREFETCH1 (L2 へ) を挿入することができます。これにより、ページ変換を事前に開始することで、L2 ハードウェアのプリフェッチャーが次のページのフェッチを開始できるようになります。

ギャザーとスキッターのいくつかのアクセスパターンは、常に連続したアドレスのペアを持ちます。典型的な例として、実部と虚部が連続して配置される複素数があります。また、w、x、y、および z の要素が連続するのも一般的な例として上げられます。値が 32 ビットであれば、連続する 64 ビット要素の半分をギャザーまたはスキッターの方が高速です。値が 64 ビットである場合、ロードしてギャザー操作を行う代わりに 128 ビットの値を挿入する方が高速です。

### 17.2.8.2 ハードウェア・プリフェッチャー

タイルには 2 タイプの HW プリフェッチャーがあります。命令ポインター・プリフェッチャー (IPP) はプロセッサ・コア内にあり、データキャッシュ内のすべてのアクセスとアクセスを要求した命令を分析します。このプリフェッチャーは、キャッシュ可能なページでのストライド・アクセス・パターンを検出すると、L1 キャッシュへハードウェア・プリフェッチャーを挿入しようと試みます。IPP は 4K ページ境界を超えることはできません。IPP はテーブルへのインデックスを作成するため、命令アドレスと論理プロセッサを使用します。これは、コンパイラーがメモリーをアクセスする命令が異なるテーブルのエントリーとなるように、大きなループ (> 256 バイト) に NOP を挿入する可能性があるためです。

L2 ハードウェア・プリフェッチャーは、ストリーミング・アクセス・パターンを識別して、48 のアクセスパターンを追跡します。ストリーミング・アクセス・パターンは、昇順または降順で連続したキャッシュラインを参照します。L2 で検

出されるストライドは常に +/- 1 キャッシュラインです。要求を起こした論理プロセッサにかかわりなく、48 個の検出器が割り当てられます。それぞれの検出器は、4KB 領域内で行われたアクセスを監視します。ストリームが検出されると、ストリームの後の要素向けのハードウェア・プリフェッチは L2 に送られ、アクセスがミスするとメモリーを参照します。ハードウェア・プリフェッチャーは、4 KB アドレス境界をまたがるストリームにアクセスしません。同じ 4KB 領域内で複数のアクセスパターンが行われると、検出器は混乱してストリームの検出に失敗します。

### 17.2.8.3 ソフトウェア・プリフェッチ

Knights Landing<sup>†</sup> マイクロアーキテクチャーは、アウトオブオーダー実行をサポートします。一般に、これは KNC<sup>†</sup> のインオーダー・マイクロアーキテクチャーよりもキャッシュミスをうまく隠匿できます。従って、プログラマーはソフトウェア・プリフェッチを積極的に使用するのを避けるべきです。

17.2.8.2 節で示した 2 つのハードウェア・プリフェッチャーは、ほとんどのストリーミングと短いストライド・アクセスパターンを検出します。アクセスパターンがストリーミングであるなら、4K バイト・ページ境界を超えるソフトウェア・プリフェッチには利点があります。アクセスパターンが不明でストリーミングしない場合、ソフトウェア・プリフェッチが有効である可能性があります。アクセスパターンが比較的大きなストライド (> 256 バイト) で、IPP が 4 KB 境界をまたがってフェッチしない場合は特に有効です。ソフトウェア・プリフェッチは、PMH (ページ・ミス・ハンドラー) が TLB を埋めるためウォークスルーを行い、早期にメモリー参照を開始できます。

一般的に、L2 キャッシュへのソフトウェア・プリフェッチは、L1 キャッシュへのソフトウェア・プリフェッチよりも多くの利点があります。L1 へのソフトウェア・プリフェッチは、キャッシュラインを完全にフィルするまでハードウェアの重要なリソース (フィルバッファ) を消費します。L2 へのソフトウェア・プリフェッチは、そのようなリソースを消費しないためパフォーマンスへの悪影響は少ないと推測されます。L1 ソフトウェア・プリフェッチを使用する場合、ハードウェア・リソースの占有時間を最小化するため、L1 ソフトウェア・プリフェッチが L2 キャッシュにヒットするように考慮することを強く推奨します。

無効なアドレスからのソフトウェア・プリフェッチ命令は、リタイアメント・スロットを消費するためパフォーマンスに悪影響を与えます。無効なアドレスからのプリフェッチや、ユーザコードから OS 特権レベルへの移行のパフォーマンス上のペナルティーは非常に大きくなります。パフォーマンス監視イベント NUKE.ALL は、コードに影響を与える指標を示します。

### 17.2.8.4 メモリー実行クラスター

MEC が `pop` をアウトオブオーダー実行するには制限があります。メモリー `pop` は、スケジューラーからインオーダーでディスパッチされますが、任意の順番で完了できます。メモリー命令の順番を再配置することで MEC の能力をうまく利用できれば、パフォーマンスが向上する可能性があります。

例 17-8 に、2 つの配列 `a[]` と `b[]` に 2 つのリードストリームでアクセスするメモリー命令シーケンスの順番の影響を示します。図 17-8 の左のリストは、最適なシーケンスであり、`b[]` からの 2 番目のベクトルロードはサイクル `N+5` でディスパッチされ L1 キャッシュにヒットすると想定されます。右のリストは、自然なメモリー命令の順番であり、2 番目のベクトルロードはサイクル `N+8` でディスパッチされます。

右のリストは左のリストよりも多くのレジスターを使用しています。ポインターロードが L1 をミスすると、リスト中のコメントに示すように左のリストの利点が大きくなります。

例 17-8 MEC 向けメモリー命令の順番

<code>movq r15, [rsp+0x40]; cycle N (load &amp;a[0])</code>	<code>movq r15, [rsp+0x40]; cycle N (load &amp;a[0])</code>
<code>movq r14, [rsp+0x48]; cycle N+1 (load &amp;b[0])</code>	<code>vmovups zmm1, [r15+rax*8]; executes in cycle N+4</code>
<code>vmovups zmm1, [r15+rax*8]; executes in cycle N+4</code>	<code>movq r15, [rsp+0x48]; cycle N+4 (load &amp;b[0])</code>
<code>vmovups zmm2, [r14+rax*8]; cycle N+5</code>	<code>vmovups zmm2, [r15+rax*8]; cycle N+8</code>

マシン上で多くのロードが実行される場合、追加で整数レジスターの予約を要求することなく、ポインターのロードと逆参照間にいくつかのメモリー参照が存在するように、ポインターロードの巻き上げが可能であるかもしれません。

### 17.2.8.5 ストア・フォワーディング

Knights Landing<sup>†</sup> マイクロアーキテクチャーにおける整数実行と MEC 向けのストア・フォワーディングの制限は、Silvermont<sup>†</sup> マイクロアーキテクチャーと同じです。ここでは、VPU とフォワーディングの制限について説明します。

ベクトル、x87、およびインテル® MMX® 命令のロードとストアは、ストアとロードが同じメモリーアドレスを持ち、ロードがストア幅よりも小さい場合にフォワードできます (ZMM0、YMM1、XMM2、MM3 および ST4)。VPU ストアは整数ロードへフォワードできません。また、整数ストアを VPU ロードへフォワードすることもできません。どちらの場合も、ロードはストアのリタイアメント後メモリーから値を取得するまで待機します。

opmask を使用するベクトルストアはフォワードされません。アルゴリズムがそのような機能を必要とする場合、レジスター内の値をマージして、その後条件 opmask を使用せずにストアすることで利点が得られる可能性があります。ロードには、マージされた値がフォワードされます。

### 17.2.8.6 ウェイとセットの競合

メモリー階層は、アクセスされるアドレスに基づいて要求の転送を決定します。L1 データキャッシュは、使用するキャッシュのセットを特定するため、アドレスの 11:6 ビットを使用します。フォワードロジックは、アクセスのサイズからフォワードの可能性やロードとストアの衝突を識別するため、アドレスの 11:0 ビットを使用します。多くの衝突がある場合、パフォーマンスが低下する可能性があります。

多くの動的メモリー割り当てルーチン (OS とコンパイラーに依存) は、同じ底部 12 ビットで大きなメモリー領域を開始します。アクセスパターンが同一の形状 (要素のサイズや次元) と類似したインデックスを持つ多くの配列にアクセスする場合、セットの競合によりパフォーマンスが大幅に低下する可能性があります。メモリーアクセスのビット [11 .. 6] が異なっていると、セットの競合を避けることができます。例えば、以下を考えてみます。

```
a = malloc(sizeof(double) * 10000);
b = malloc(sizeof(double) * 10000);
for (i=0; i < 10000; i++) {
    a[i] = b[i] + 0.5 * b[i-1]);
}
```

ほとんどの OS では、a[] と b[] の有効アドレスは同じ下位 12 ビットを持つ可能性があります。つまり、(a & 0xfff) == (b & 0xfff) になります。ループ内で以下が発生する可能性があります。

- Y 反復 N の a[i] と b[i] が衝突
- Y 反復 N-1 の a[i] と 反復 N の b[i-1] が衝突

動的配列をオフセットするには、いくつかの方法があります。次に例を示します。

- Y キャッシュラインの量に応じて、malloc から得られたベースポインターをオフセットします。
- Y カスタム化された malloc() ルーチンを使用します。
- Y 異なるアライメント (2 の累乗: 64, 128, 256, 512 など) を得るため、それぞれの動的な割り当てでアライメント・ディレクティブと posix\_memalign() ルーチンを使用します。

Leslie3D として知られる HPC ワークロードは、アライメントの問題の影響を受けます。

### 17.2.8.7 ストリーミング・ストアと通常のストア

メモリーに書き込まれるデータがロードによってしばらくアクセスされないことが予想される場合、ストリーミング・ストアまたは通常のストア（ライトバック）のどちらを選択するか決定する良い例と言えます。Knights Landing<sup>†</sup> マイクロアーキテクチャーでは、“フラット”メモリーモードが選択されている場合、ストリーミング・ストアが望ましいでしょう。17.1.3 節を参照してください。

MCDRAM がキャッシュモードに設定されている場合、MCDRAM キャッシュに収まるようにデータが書き込まれていると、通常のストアがうまく動作します。実際には、両方のオプションを試すことでアプリケーションに適したパフォーマンスがもたらされるでしょう。

### 17.2.8.8 コンパイラー・オプションとディレクティブ

Fortran 90 構文を使用する場合、適切であれば Fortran プログラマーは CONTIGUOUS 属性を使用すべきです。さもないと、コンパイラーは受け取る配列が連続していないと仮定し、ベクトルロードとストア命令を VGATHER と VSCATTER 命令に置き換える可能性があります。これはパフォーマンスに悪影響を与えます。

インテル® コンパイラーを使用する開発者は、さまざまなプラグマやディレクティブを使用してコードに注釈を加えることができます。有効な注釈として、LOOP\_COUNT、SIMD、および UNROLL が挙げられます。これらのプラグマやディレクティブのドキュメントを参照して、適切な場所で使用してください。コンパイラーは、ベクトル化のコストを評価するためより多くの情報が得られる場合に優れたコードを生成できます。

インテル® コンパイラーを使用する場合、コンパイラー・オプション “-xMIC-AVX512” で、Knights Landing<sup>†</sup> マイクロアーキテクチャーをターゲットとします。

### 17.2.8.9 ダイレクト割り当て MCDRAM キャッシュ

MCDRAM がキャッシュモードに設定されている場合、MCDRAM キャッシュはメモリー帯域幅を増加させる便利な方法です。メモリー側のキャッシュとして、自動的に使用されたデータをキャッシュし、DDR メモリーよりもはるかに高い帯域幅を提供します。

MCDRAM キャッシュは、ダイレクト・マップ・キャッシュです。これは、複数のメモリー位置がキャッシュ内の単一の場所にマップされることを意味します。そのため、メモリー帯域幅に影響を受けるプログラム向けの簡単な最適化は、MCDRAM をキャッシュモードとして有効にすることです。数 GB のメモリーを頻繁に使用する一部のアプリケーションでは、最大 4 倍のパフォーマンス向上を達成できました。これは、ソースコードを変更することなく、DDR からキャッシュモードの MCDRAM へ移行するだけで大幅にパフォーマンスが向上できるため、最初に試みるべき最適化です。

キャッシュを有効にすることでパフォーマンスが低下するいくつかの状況があります。1 つは、MCDRAM キャッシュがアクセスされたワーキングセットを保持できない場合です。アプリケーションが 64GB メモリーを再利用することなくストリームすると、MCDRAM キャッシュのチェック（およびミス）のため、DDR メモリーへのアクセスと比較してメモリーアクセスのコストは増加します。

MCDRAM ダイレクト・マップ・キャッシュへのデータのキャッシングでは、リニアアドレスではなく物理アドレスを使用します。リニア/仮想アドレス空間でアドレスが連続していても、OS が割り当ておよび管理する物理アドレスは連続しているとは限りません。これは、MCDRAM の大部分を使用する場合、キャッシュ競合を引き起します。この競合は、利用可能なピークメモリー帯域幅を減少させます。これは、OS のページ割り当てが実行ごとに異なるため、実行するたびに変わります。Knights Landing<sup>†</sup> マイクロアーキテクチャーのパフォーマンス・モニタリング・ハードウェアは、MCDRAM のキャッシュヒット率を計算するイベント UNC\_E\_EDC\_ACCESS を提供しており、この問題を診断する上で役立つことがあります。

MCDRAM キャッシュが有効である場合、タイル内のキャッシュ (L1 または L2 キャッシュ) のすべての変更されたキャッシュラインは、MCDRAM キャッシュにエントリーを持つ必要があります。キャッシュラインが MCDRAM から追い出されている場合、タイル内のキャッシュの変更されたラインのデータはメモリーにライトバックされ、キャッシュ状態は共有へ移行します。頻繁にリードとライトされるラインのペアが、同じ MCDRAM セットにエイリアスされることはまれです。MCDRAM をキャッシュモードで使用すると、通常タイル内のキャッシュにヒットするライトのペアが、余分なメッシュ・トラフィックを発生させる原因となります。これは、そのスレッドのペアが、チップ内のほかのスレッドよりも遅くなる原因となります。リニアから物理アドレスへのマッピングは一定でなく、実行ごとに振る舞いが異なるため診断が困難になります。

例えば、2 つのスレッドがプライベートなスタックをそれぞれリードおよびライトすると、この問題が発生します。概念的には、通常リード/ライトされるすべてのデータの場所にリード/ライトできるべきですが、スタックへのレジスター退避が最も頻繁に発生するケースです。スタックが物理メモリーで 16GB の倍数 (または、総 MCDRAM キャッシュサイズ) にオフセットされている場合、同じ MCDRAM キャッシュセットで衝突する可能性があります。実行時にすべてのスレッドのスタックを物理メモリー領域に連続して割り当ててを強制することで、この問題を回避できます。

Knights Landing<sup>†</sup> マイクロアーキテクチャーには、セット衝突の発生頻度を低減するハードウェア機能があります。特定のノード上で、このシナリオに遭遇する可能性は非常に低いでしょう。この問題を特定する手がかりは、同じチップ上のスレッドのペアは、プログラム全体を通して他のスレッドよりもかなり低速になることです。パッケージ内のどのスレッドコアが衝突を起こすかは、実行ごとに異なり、またまれにしか発生せず、“キャッシュ” メモリーモードが有効である場合のみに限定されます。ユーザーはシステム上でこの問題に遭遇しない可能性もあります。

インテルが提供するアプリケーション・パフォーマンス・ツールを活用すると、インテル® アーキテクチャー (IA) ベースのプロセッサのパフォーマンスを引き出すことができます。この付録では、これらのツールを紹介し、その機能について説明します。これらのツールを使用すると、アセンブリー・コードを記述しなくても最も効率の高いプログラムを開発できます。

次のパフォーマンス・ツールが利用できます。

#### Y コンパイラー

- インテル® C++ コンパイラー: ハイパフォーマンスな最適化された C および C++ クロスコンパイラーはインテル® HD グラフィックスやインテル® メニー・インテグレートッド・コア (インテル® MIC) アーキテクチャーへの計算集約型のコードのオフロード機能とインテル® Cilk™ Plus や OpenMP\* を使用した複数実行ユニットでの実行機能を備えています。
- インテル® Fortran コンパイラー: ハイパフォーマンスな最適化された Fortran コンパイラーです。

#### Y パフォーマンス・ライブラリー: インテル® アーキテクチャー・ベースのプロセッサ向けに最適化されたソフトウェア・ライブラリーのセット。

- インテル® インテグレートッド・パフォーマンス・プリミティブ (インテル® IPP): 画像処理、信号処理、データ圧縮および暗号化向けのハイパフォーマンス・ライブラリー。
- インテル® マス・カーネル・ライブラリー (インテル® MKL): 高度にベクトル化およびスレッド化された線形代数、高速フーリエ変換 (FFT)、ベクトル数学関数、そして統計関数のセット。
- インテル® スレディング・ビルディング・ブロック (インテル® TBB): 高いパフォーマンスとスケーラビリティに優れた並列アプリケーションの開発を支援する、C と C++ 向けのテンプレート・ライブラリー。
- インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL): データの取得からデータマイニング、機械学習まで、すべてのデータ解析フェーズに対応した最適化された解析ビルディング・ブロックを提供する C++ および Java\* API ライブラリーです。ハイパフォーマンスなビッグデータ・アプリケーションには不可欠です。

#### Y パフォーマンス・プロファイラー: パフォーマンス・プロファイラーは、CPU、GPU、スレッド化、ベクトル化および MPI 並列をチューニングするため、ソフトウェアのパフォーマンス・データを取集、解析、そして表示する機能を備えています。収集されたデータにより、システム全体から特定のコード行まで調査できます。

- インテル® VTune™ Amplifier パフォーマンス・プロファイラー
- インテル® Graphics Performance Analyzers (インテル® GPA): グラフィックス・アプリケーション向けのパフォーマンス・アナライザー
- インテル® Advisor: ベクトル化の最適化とスレッド化のプロトタイプ作成
- インテル® Trace Analyzer & Collector (トレース・アナライザー/コレクター): MPI 通信のパフォーマンス・プロファイラーと正当性検証ツール

#### Y デバッガー

- インテル® Inspector: メモリーとスレッド化向けデバッガー
- インテル® Application Debugger
- インテル® JTAG デバッガー
- インテル® System Debugger

#### Y クラスターツール

- インテル® MPI ライブラリー: ハイパフォーマンス MPI ライブラリー
- インテル® MPI Benchmarks: クラスタや MPI 実装のパフォーマンスを検証するための MPI カーネルテストのセット

上記のパフォーマンス・ツールは、次の製品スイートのいずれかに含まれています。

- Y インテル® Parallel Studio XE
  - インテル® Media Server Studio
  - インテル® System Studio

## A.1 コンパイラー

インテル® コンパイラーは、/O1、/O2、/O3、/fast などの汎用的な最適化の設定をサポートしています。いずれを指定した場合でも、それぞれ固有の最適化オプションが有効になります。ほとんどの場合は、/O1 オプションよりも、関数のインライン展開が有効になる /O2 オプションを使用することを推奨します。小規模な関数呼び出しの多いプログラムには、関数インラインが役立ちます。コードサイズが重要なときは、/O1 オプションが望ましい場合もあります。デフォルトでは、/O2 オプションが有効になります。

/Od (Linux\* では -O0) オプションを使用すると、最適化機能がすべて無効になります。/O3 オプションを使用すると、より強力な最適化機能が有効になります。ただし、そのほとんどは、以下で説明するプロセッサ固有の最適化機能と組み合わせた場合のみ有効です。

/fast オプションは、プログラム全体の速度を最大限にします。インテル® 64 プロセッサおよび IA-32 プロセッサ向けには、「/fast オプション」は、「/O3 /Qipo /Qprec-div -/fp:fast=2 /QxHOST (Windows\*)」、「-Q3 -ipo -no-prec-div -static -fp-model fast=2 -xhost (Linux\*)」、および「-O3 -ipo -mdynamic-no-pic -no-prec-div -fo-model fast=2 -xhost (OS X\*)」と同じです。

上記のコマンドライン・オプションは、インテル® コンパイラーのドキュメントで説明されています。

### A.1.1 インテル® 64 プロセッサと IA-32 プロセッサの推奨される最適化設定

表 A-1 は、インテル® プロセッサ向けにコードを生成する推奨コンパイラー・オプションを示しています。表 A-1 は、インテル® 64 プロセッサの互換モードで動作するコードにも適用されますが、64 ビット・モードでの動作には適用されません。最新の情報については次の記事をご覧ください:

<https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-option-s-forse-generation-and-processor-specific-optimizations/> (英語)

表 A-1 推奨されるプロセッサ最適化オプション

条件	推奨	説明
Y インテル® AVX2 を利用する インテル® プロセッサで最高のパフォーマンスを達成。	Y /QxCORE-AVX2 (Linux* と macOS* では -x CORE-AVX2)	Y 単一コードパス。
Y インテル® AVX2 を利用する インテル® プロセッサで最高のパフォーマンスを達成。	Y /QaxCORE-AVX2 (Linux* と macOS* では -ax CORE-AVX2)	Y 複数のコードパスが生成されます。 Y 実行するすべてのシステム上でアプリケーションの検証を行う必要があります。
Y インテル® SSE4.2 を利用する インテル® プロセッサで最高のパフォーマンスを達成。	Y /QxSSE4.2 (Linux* と macOS* では -x SSE4.2)	Y 単一コードパス。
Y インテル® SSE4.2 を利用する インテル® プロセッサで最高のパフォーマンスを達成。	Y /QaxSSE4.2 (Linux* と macOS* では -ax SSE4.2)	Y 複数のコードパスが生成されます。 Y 実行するすべてのシステム上でアプリケーションの検証を行う必要があります。

## A.1.2 ベクトル化とループの最適化

インテル® C++コンパイラーとインテル® Fortran コンパイラーのベクトル化機能は、同一の命令によるシーケンシャルなデータアクセスを検出し、対象とするプロセッサ・プラットフォームに応じてインテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4、インテル® AVX、インテル® AVX2、インテル® AVX-512 を使用するようにコードを変換できます。ベクトライザーは以下の機能をサポートしています。

- Y 複数のデータ型: float/double, char/short/int/long(符号付きと符号なし), \_Complex float/double をサポートします。
- Y ステップごとの診断: /Qopt-report (Linux\* や macOS\* では -qopt-report) オプションによって、ベクトライザーは、行や変数ごとに、ベクトル化されたコード、ベクトル化されなかったコード、および各コードがベクトル化されなかった理由をレポートします。このフィードバックから得られる情報に基づいて、開発者は、依存関係ディレクティブ、restrict キーワード、または OpenMP\* ディレクティブを使用してベクトル化が行われるようにコードを修正または再構築できます。
- Y 動的なデータ・アライメント手法: アライメント手法には、ループピーリングやループアンロールが含まれます。ループピーリングは、アライメント済みロードを生成し、アプリケーションのパフォーマンスを向上させます。ループアンロールは、フル・キャッシュラインのプリフェッチと一致させ、スケジューリングを向上させます。
- Y 移植性のあるコード: 適切なインテル® コンパイラー・オプションを使用して新しいプロセッサの機能を利用でき、開発者はソースコードを書き換える必要がありません。

プロセッサ固有のベクトライザーのオプションには次のものがあります: /Qx<CODE> および /Qax<CODE> (Linux\* と macOS\* では -x<CODE> および -xa<CODE>)。コンパイラーは、ベクトル化を制御するベクトライザーのオプションを多数備えています。それらのオプションを使用するには、/Qx<CODE> または /Qax<CODE> オプションのいずれかが有効である必要があります。デフォルトは無効です。

### A.1.2.1 OpenMP\* による並列化

インテル® C++ コンパイラーおよびインテル® Fortran コンパイラーは、OpenMP\* プラグマやディレクティブ、ランタイム API および環境変数を使用した共有メモリー並列 (スレッド化) と分散メモリー並列 (オフロード) 機能をサポートします。OpenMP\* プラグマ/ディレクティブは、/Qopenmp (Linux\* と macOS\* では -qopenmp) コンパイラー・オプションにより有効化されます。利用可能なプラグマ/ディレクティブについては、インテル® C++ および Fortran コンパイラーとともに提供されるデベロッパー・ガイドおよびリファレンス、または OpenMP\* シンタックス・リファレンス・カードに示されています。OpenMP\* に関する詳細は、<http://www.openmp.org> (英語) のウェブサイトをご覧ください。

### A.1.2.2 自動並列化

依存関係のない単純なループでは、インテル® C++ コンパイラーとインテル® Fortran コンパイラーは、マルチスレッド・コードを自動的に生成できます。この機能は、コンパイラー・スイッチ /Qparallel (Linux\* と macOS\* では -parallel) によって有効になります。

### A.1.3 ライブラリー関数のインライン展開 (/Oi、/Oi-)

デフォルトでは、C、C++、数値演算ライブラリーの標準関数がコンパイラーによってインライン展開されます。通常は、これにより実行速度が高速化されます。しかし、ライブラリー関数をインライン展開すると、予期しない結果をもたらす場合があります。詳細については、インテル® コンパイラー・ドキュメントを参照してください。

### A.1.4 プロシージャー間とプロファイルに基づく最適化

コードのプロファイルとプロシージャー間の依存関係に基づいてコードのパフォーマンスを改善する 2 つの方法を以下に示します。



### A.1.4.1 プロシージャー間の最適化 (IPO)

ソースファイル内のプロシージャー間の最適化を行うには、/Qip (Linux\* と macOS\* では -ip) オプションを使用します。また、複数のソースファイル間のプロシージャー間の最適化を有効にするには、/Qipo (Linux\* と macOS\* では -ipo) オプションを使用します。

### A.1.4.2 プロファイルに基づく最適化 (PGO)

コンパイラーはソースコードからインストルメントを行うコードを埋め込んで、プロファイルを収集可能なプログラムを作成し、リンクします。このインストルメント・コードが実行されるたびに、動的に情報ファイルが作成されます。2 度目のフィードバック・コンパイルでは、生成された動的情報がマージされてサマリーファイルが作成されます。このプロファイル情報の概要を使用して、コンパイラーはプログラム内で最も頻繁に実行されるパスの最適化を図ります。

プロファイルに基づく最適化は、特にインテル® Pentium® 4 プロセッサーとインテル® Xeon® プロセッサー・ファミリーで有効です。これは、コンパイラーによる命令キャッシュの利用率とメモリーページに関連する最適化の判断を大幅に高めます。また、PGO では最適化をガイドする実行時間情報を使用できるため、プロセッサーに適した分岐のヒント生成し、マイクロアーキテクチャーのパイプラインで最も頻繁に実行されるパスを維持するため分岐と基本ブロックを再配置することで分岐予測がかなり改善されます。

PGO を使用する際は、次のガイドラインに従ってください。

- Y インストルメント済みコードを実行してからフィードバック・コンパイルを行うまでの間は、プログラムに加える変更を最小限にします。フィードバック・コンパイルでは、情報が生成した後に変更された関数の動的情報は無視します。

#### 注意

プログラムが変更されている場合、コンパイラーは動的情報が関数に対応していないことを示す警告を示します。

- Y インストルメント済みコードを実行してからフィードバック・コンパイルを行うまでの間にソースファイルに多数の変更を加えた場合、インストルメンテーション・コンパイルを繰り返します。

最適化オプションの詳細については、インテル® コンパイラーのドキュメントを参照してください。

### A.1.5 インテル® Cilk™ Plus

インテル® Cilk™ Plus は、3 つのキーワードだけで簡単に単純なループとタスク並列をアプリケーションに実装できる C/C++ コンパイラー拡張です。また、ベクトル化機能と、高度なループベースのデータ並列処理およびタスク処理を組み合わせることで、優れた機能を提供します。

## A.2 パフォーマンス・ライブラリー

インテル® パフォーマンス・ライブラリーは、本書を通して説明する多くの最適化を実装しています。これには、ループアンロール、命令のペアリングとスケジューリングなどのアーキテクチャー固有のチューニング、および暗黙や明示的なデータ・プリフェッチとキャッシュ・チューニングを含むメモリー管理などが含まれます。

インテル® パフォーマンス・ライブラリーは、インテル® MMX® テクノロジー、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3) を使用した SIMD 命令レベルの並列処理の利点を活用します。これらの手法により計算集約型のアルゴリズムのパフォーマンスを向上させ、高級言語開発環境向けに手動で最適化されたパフォーマンスを提供します。

インテル® パフォーマンス・ライブラリーは、パフォーマンスが重要なアプリケーション向けに、頻繁に実行される関数のアセンブリ・レベルでの時間がかかる作業から開発者を解放します。プロトタイプ化と新しいアプリケーション機能の実装に必要な時間が大幅に短縮され、さらに市場投入までの期間を劇的に短縮できます。インテル® パフォーマンス・ライブラリーを使用して開発されたアプリケーションは、アップグレードされたバージョンのライブラリーをアプリケーションと再リンクするだけで、簡単に将来のインテル® プロセッサ世代の新しいアーキテクチャーの機能を活用できます。

ライブラリー・セットには、インテル® インテグレートッド・パフォーマンス・プリミティブ (インテル® IPP)、インテル® マス・カーネル・ライブラリー (インテル® MKL)、およびインテル® スレッディング・ビルディング・ブロック (インテル® TBB) が含まれます。

## A.2.1 インテル® インテグレートッド・パフォーマンス・プリミティブ (インテル® IPP)

インテル® IPP for Linux\* およびインテル® IPP for Windows\*: インテル® IPP は、ビデオデコード/エンコード、オーディオデコード/エンコード、イメージカラー変換、コンピューター・ビジョン、データ圧縮、ストリング処理、信号処理、音声認識、音声デコード/エンコード、暗号化、さらに数学ルーチンなど広範囲な分野をサポートする、クロスプラットフォーム向けソフトウェア・ライブラリーです。

これらの関数は、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、インテル® アドバンスド・ベクトル・エクステンション (インテル® AVX) 命令、およびインテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512) 命令セットを使用して高度に最適化されています。単一の API で広範囲のプラットフォームに対応できるため、互換性と開発コストの軽減を実現します。

## A.2.2 インテル® マス・カーネル・ライブラリー (インテル® MKL)

インテル® MKL for Linux\*, Windows\*, macOS\*: インテル® MKL は、インテル® プラットフォーム上で高いパフォーマンスが要求される工学、科学、および金融アプリケーション向けの高度に最適化された数学関数で構成されます。ライブラリー関数は、LAPACK や BLAS から成る線形代数、離散フーリエ変換 (DFT)、ベクトル超越関数 (ベクトル数学ライブラリー/VML)、およびベクトル統計関数 (VSL) などを含んでいます。インテル® MKL は、インテル® Xeon Phi™ 製品、インテル® Xeon® プロセッサ、インテル® Core™ プロセッサ、インテル® Core™2 プロセッサ、インテル® Pentium® 4 プロセッサベースのシステムの機能を引き出すように最適化されています。マルチコアおよびメニーコア向けのマルチスレッド化によるパフォーマンスの最適化は特に配慮されています。

## A.2.3 インテル® スレッディング・ビルディング・ブロック (インテル® TBB)

インテル® TBB は広く使用されている C++ テンプレート・ライブラリーであり、安定性を備え、移植性とスケラビリティに優れた並列アプリケーションの作成を支援します。インテル® TBB を利用することで、さまざまな環境でマルチコアおよびメニーコア・プロセッサの能力を最大限に活用し、パフォーマンスを引き出すことができだけでなく、保守も容易な優れたタスクベースの並列アプリケーションを簡単に短期間で開発できます。

インテル® TBB は、Windows\*, Linux\*, そして OS X\* プラットフォーム上で、各種コンパイラーを使用して検証され、商用サポートされています。また、オープンソース・コミュニティにより、FreeBSD\*, IA ベース Solaris\*, Xbox\* 360 および PowerPC\* ベースのシステムでも利用できます。

## A.2.4 利点のまとめ

上記のライブラリーを使用することで、アプリケーション開発者は総じて次の利点が得られます。

- Y **開発期間の短縮** — 低レベルの各種ビルディング・ブロック関数によってアプリケーションを素早く開発でき、開発期間を短縮できます。

- Y **パフォーマンス** — 高度に最適化された C インターフェイスを備えた各種ルーチンによって、C/C++ 開発環境でアセンブリ・レベルのパフォーマンスが得られます。インテル® MKL は Fortran インターフェイスにも対応しています。
- Y **最適化されたプラットフォーム** — プロセッサ固有の最適化により、インテル® プロセッサの世代ごとに最高のパフォーマンスを引き出すことができます。
- Y **互換性** — 単一のアプリケーション・プログラミング・インターフェイス (API) を使用してプロセッサ固有の最適化を行うことにより、開発コストを削減しつつ、最適なパフォーマンスを提供できます。
- Y **スレッド化されたアプリケーションのサポート** — インテル® MKL とインテル® IPP の関数はスレッド環境で安全に使用できるため、スレッド化されているアプリケーションで容易に利用できます。

## A.3 パフォーマンス・プロファイラー

インテルのシリアルおよび並列処理プロファイル・ツールは、アプリケーションを再コンパイルすることなく低オーバーヘッドでパフォーマンスのボトルネックを特定し、高速化のためのスケーリング情報と改善のための意思決定に有用な情報を提供します。プロファイル・ツールは、組み込みシステムからスーパーコンピュータまで、インテル® プロセッサベースの広範囲なシステムの評価を可能にし、アプリケーションのパフォーマンス向上に役立ちます。

### A.3.1 インテル® VTune™ Amplifier

インテル® VTune™ Amplifier<sup>13</sup> は、Windows\* および Linux\* 向けの強力なスレッド化とパフォーマンス最適化ツールです。インテル® VTune™ Amplifier を使用すると、プロセッサ・コアを完全に活用し、確実に新しいプロセッサの能力を最大限に引き出して、最適なパフォーマンスに向けてファインチューニングが可能になります。

この節では、インテル® VTune™ Amplifier の主要な機能について説明します。これらの機能の詳細については、インテル® VTune™ Amplifier の製品ドキュメントとオンラインヘルプを参照してください。

#### A.3.1.1 ハードウェア・イベントベース・サンプリング解析

インテル® VTune™ Amplifier は、イベントベース・サンプリングによるデータ収集に基づいて、インテル® Core™2 プロセッサから最新のプロセッサまで、マイクロアーキテクチャーの解析を可能にします。解析タイプごとに、インテル® VTune™ Amplifier は関連するハードウェア・イベントをモニターし、収集したデータの生のイベントカウント (キャッシュミス、クロックティック、命令リタイアなど) とパフォーマンス・メトリックを表示します。それぞれのメトリックには、しきい値とイベント比率が設定されています。プログラムユニットのメトリックごとのパフォーマンスがしきい値を超えると、インテル® VTune™ Amplifier はパフォーマンスの問題としてその値を (ピンク色で) ハイライトし、推奨される解決方法を提示します。

それぞれのインテル® プロセッサで利用可能なイベントについては、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」をご覧ください。

#### A.3.1.2 アルゴリズム解析

インテル® VTune™ Amplifier は、ユーザーモードのサンプリングとトレース収集に基づくアルゴリズム解析タイプを導入しています。

- Y **ホットスポット解析**は、アプリケーションの実行フローを理解し、実行に長い時間がかかっているコード領域 (ホットスポット) を特定するのに役立ちます。特定のプロセス、スレッドまたはモジュールで多数のサンプルが収集されていると、プロセッサの利用率が高くパフォーマンスのボトルネックである可能性があります。一部のホットスポットは排除できますが、アプリケーション機能の根本にあるホットスポットは排除することはできません。インテル® VTune™ Amplifier は、関数で費やされた時間順にアプリケーションの関数のリストを作成します。

<sup>13</sup> 次のリソースもご覧ください: <https://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/> (英語)

関数のコールスタックも表示するため、時間を費やしている関数がどのように呼び出されているかを確認できます。

- Y **ロックと待機解析**は、プロセッサ使用率が効率的ではない原因を特定します。スレッドが長時間同期オブジェクト (ロック) を待機するのは、最も一般的な問題の 1 つです。コアが十分に利用されず待機が発生すると、パフォーマンスは影響を受けます。ロックとウェイト解析により、アプリケーションに導入するそれぞれの同期オブジェクトの影響を検証し、アプリケーションが同期オブジェクト、またはスリープやブロック化 I/O などの API で待機している時間を理解できます。
- Y **並行性解析**は、プロセッサの利用率が低い場所のホットスポット関数を特定するのに有効です。ホットスポットでコアがアイドル状態になっている場合、それらのコアにワークを与えることでパフォーマンスを向上できる可能性があります。

### A.3.1.3 プラットフォーム解析

レンダリング、ビデオ処理、および計算向けにグラフィックス処理ユニット (GPU) を使用するアプリケーション向けのプラットフォーム全体のメトリックを収集するためインテル® VTune™ Amplifier を使用できます。システムの各種 CPU と GPU コアにおけるコードの実行を理解するため、CPU/GPU 並行性解析を使用し、ターゲットアプリケーションが GPU 依存であるか CPU 依存であるかを特定します。

## A.4 スレッドとメモリーチェッカー

アプリケーションの信頼性、セキュリティ、および精度を向上するため、スレッド化とメモリーエラーのチェックを 1 つの強力なエラー・チェック・ツールとして統合しました。

### A.4.1 インテル® Inspector

インテル® Inspector は、高い並列実行を行うアプリケーション向けのスレッドのバック解析 (データ競合、デッドロック、スレッドと同期 API の使用とスレッド間のメモリーアクセスを検出) と、シリアルおよび並列アプリケーション向けのメモリーチェック解析 (メモリーリークとメモリー破壊、メモリーの割り当てと解放 API の不一致、および一貫性のないメモリー API の利用などを検出) を提供します。

インテル® Inspector は、開発サイクルの早い段階で重大なメモリーとスレッド化の欠陥を発見することで、開発の生産性を高めてアプリケーションの信頼性を向上させます。この機能は、アプリケーションのメモリーとスレッド動作に関する詳細情報を提供し、アプリケーションの信頼性を向上できるように支援します。強力なスレッド検証ツールとデバッガーにより、実行コードパスの潜在的なエラーを簡単に見つけられます。また、エラーを引き起こすシナリオが実行されない場合でも、間欠的なエラーや非決定性的エラーを発見します。また、開発者は特殊なコードのビルドやコンパイルを行うことなく、頻繁にコードをテストできるようになります。

## A.5 ベクトル化のアシスタント

### A.5.1 インテル® Advisor

インテル® Advisor は、ベクトル化のアシスタントとスレッドのプロトタイプ化ツールであり、ソースコード中で最もベクトル化とスレッド化の影響がある領域を特定することで、容易にスレッド化とベクトル化が可能になります。

## A.6 クラスターツール

インテル® Parallel Studio XE Cluster Edition は、IA-32、IA-64、インテル® 64 アーキテクチャーをベースとするクラスター向けの並列アプリケーションの開発、解析、およびパフォーマンス最適化に役立ちます。Cluster Edition には、クラスター向けのコード開発に有用な、インテル® Trace Analyzer & Collector、インテル® MPI ライブラリー、インテル® MPI Benchmarks が含まれています。

## A.6.1 インテル® Trace Analyzer & Collector

インテル® Trace Analyzer and Collector<sup>14</sup> は、MPI 通信におけるパフォーマンス・ボトルネックを素早く発見することで、クラスター上のアプリケーション・パフォーマンスを理解および最適化するのに欠かせない情報を提供します。インテル® アーキテクチャー・ベースのクラスターシステムのサポート、現在の標準化と高い互換性を持つ機能、またトレースファイルの理想化と比較、カウンターデータ表示、パフォーマンス・アシスタント、MPI 正当性チェック・ライブラリーなどを含みます。MPI パフォーマンスを解析し、並列アプリケーションの実行をスピードアップし、ホットスポットとボトルネックを特定して、トレースファイルとグラフィックスを比較してタイムラインに適合した詳細な解析を提供します。

### A.6.1.1 インテル® MPI パフォーマンス・スナップショット

MPI Performance Snapshot (MPS) は、MPI アプリケーション向けのスケラブルで軽量なパフォーマンス・ツールです。通信、アクティビティー、負荷バランスなどの MPI アプリケーションの特性を収集し、分かりやすい形式で表示します。MPS はアプリケーションのハイレベルの概要を提供するため、インテル® MPI ライブラリーと OS およびハードウェア・カウンターからライトウェイト統計を組み合わせます。このツールは、インテル® Trace Analyzer & Collector の一部としてインストールされます。

## A.6.2 インテル® MPI ライブラリー

インテル® MPI ライブラリーは、メッセージ・パッシング・インターフェイス 3.0 (MPI-3.0) 仕様を実装する、マルチファブリックをサポートするメッセージ・パッシング・ライブラリーです。インテル® プラットフォーム向けに標準ライブラリーを提供します。インテル® MPI ライブラリーは、InfiniBand\*、Myrinet\*、およびインテル® True Scale ファブリックを含む複数のハードウェア・ファブリックをサポートしています。Direct Access Programming Library (DAPL) 方式を介した高速インターコネクト向けのユニバーサルなマルチファブリック・レイヤーを提供することにより、すべての構成をカバーします。実行時にユーザーがどのファブリックを選択しても、効率良く実行できる、ファブリックに依存しない MPI コードを開発できます。

インテル® MPI ライブラリーは、必要な場合のみ動的に接続を確立し、メモリー・フットプリントを削減します。また、利用できるトランスポートの中から最も高速なものを自動で選択します。ジョブ開始時のソケットへのフォールバックにより、インターコネクトの選択に失敗した場合でも、実行の失敗を回避できます。これは、特にバッチ・コンピューティングにおいて有効です。インテル® MPI ライブラリーで開発されたアプリケーションの展開は、インテルから無料ランタイム環境キットをダウンロードできるため、ランタイム互換が保証されます。また、マルチコアノード内または SMP ノード内で DAPL をオプションで使用すると、大きなメッセージ帯域幅の利点から、アプリケーション・パフォーマンスが向上します。

## A.6.3 インテル® MPI Benchmarks

インテル® MPI Benchmarks を利用すると、MPI 関数とパターンの性能比較を簡単に行うことができます。このベンチマークでは、ユーザビリティ、アプリケーション・パフォーマンス、相互運用性が強化されています。

## A.7 インテル® ACADEMIC COMMUNITY

インテル® Academic Community で提供されるクラスルーム・トレーニングの詳細については、<https://software.intel.com/en-us/articles/intel-academic-community/> (英語) をご覧ください。開発者向けの一般的な情報については、<https://software.intel.com/en-us/> (英語) を参照してください。

<sup>14</sup> インテル® Trace Analyzer & Collector は、インテル® Parallel Studio XE Cluster Edition にも含まれます。

パフォーマンス監視イベントは、プログラムの命令シーケンスとマイクロアーキテクチャー・サブシステムの相互作用の特性を評価するための機能を提供します。パフォーマンス監視イベントの詳細については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章と第 19 章で説明されています。

この節の前半では、インテル® マイクロアーキテクチャー向けのパフォーマンス・チューニングにおける、パフォーマンス・ボトルネックを解析するトップダウン方式のマイクロアーキテクチャー解析手法 (TMAM) について説明します。B.1 節では、最近のインテル® マイクロアーキテクチャーに適用できる一般化された手法を紹介します。Skylake<sup>†</sup> マイクロアーキテクチャー向けの TMAM の具体例が含まれています。

この節の残りのセクションでは、以前の世代のインテル® マイクロアーキテクチャーで役立つ情報をカバーしています。

## B.1 トップダウン解析法

この節では、アウトオブオーダー・コアのパフォーマンス・ボトルネックを特定するトップダウン・マイクロアーキテクチャー解析方式 (TMAM) について説明します。一般的な階層フレームワークと階層的なテクニックの概念は、多くのアウトオブオーダー・マイクロアーキテクチャーに適用できます。

TMAM は、単一の階層構造で編成されるマイクロアーキテクチャーに依存しないメトリックを使用することでサイクル・アカウンティング (パフォーマンス・ボトルネックのコストを特定する過程で、CPI のブレイクダウンとも呼ばれます) を簡略化します。

図 B-1 は、現代のマイクロアーキテクチャーの主要な機能ブロックに関連したパフォーマンス・ボトルネックを分類する階層的なアプローチを示しています。TMAM を使用したそれぞれのマイクロアーキテクチャーの世代に関連する高い学習曲線は、真にパフォーマンスを制限する原因を明らかにする体系的なドリルダウンによって置き換えられます。これは、マイクロアーキテクチャーの詳細をすべて理解することなく、パフォーマンス解析を行うことを可能にします。

このトップダウン階層フレームワークの利点は、調査するマイクロアーキテクチャーで考えられることをドリルダウンして開発者を導く体系的なアプローチです。あまり意味のない問題を無視し、真に解決すべき問題に解析の労力を充てることを可能にするため、ツリーのノードを重み付けします。

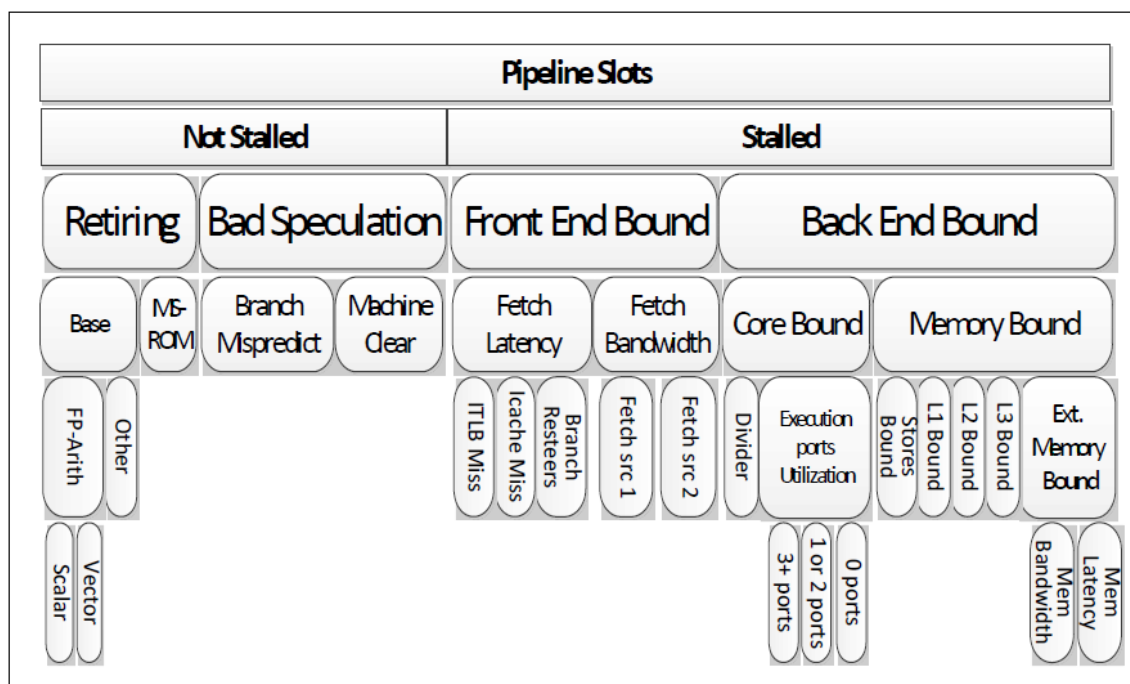


図 B-1 アウトオブオーダー・マイクロアーキテクチャー向けの一般的な TMAM 階層構造

例えば、アプリケーションに命令フェッチの問題があるなら、TMAM はフロントエンド依存としてツリーの最上位レベルでそれを分類します。ユーザーやツールは、フロントエンドのサブツリーのみに注目してドリルダウンできます。ドリルダウンは、ツリーのリーフ（樹木葉）に達するまで繰り返されます。リーフはワークロードの特定のストールを示すか、アプリケーションのパフォーマンスを制限する一般的なマイクロアーキテクチャー上の兆候と問題のサブセットを表します。

TMAM は、Sandy Bridge<sup>+</sup> マイクロアーキテクチャーのパフォーマンス監視機能に関連して開発されました。この方法論は、複数世代にわたるマイクロアーキテクチャー世代をサポートするため洗練され、その後 PMU 機能によって拡張されました。

### B.1.1 トップレベル

トップレベルで、TMAM はパイプライン・スロットを 4 つの状態に分類します。

- Y フロントエンド依存。
- Y バックエンド依存。
- Y 投機の問題。
- Y リタイア。

図 B-1 に見られるように、後者 2 つは非ストールスロットを示し、前者 2 つはストールを示します。

図 B-2 にドリルダウンの簡単な決定木を示します。

- Y スロットが何らかの操作で利用されると、それはリタイア（コミット）されたかどうかにより、リタイアまたは投機の問題として分類されます。
- Y パイプラインのバックエンドが操作を受け入れることができない場合（バックエンド・ストール）、利用されなかったスロットはバックエンド依存として分類されます。または、
  - フロントエンド依存: バックエンドはストールしていないにもかかわらず、操作（μop）が供給されなかったことを示します。

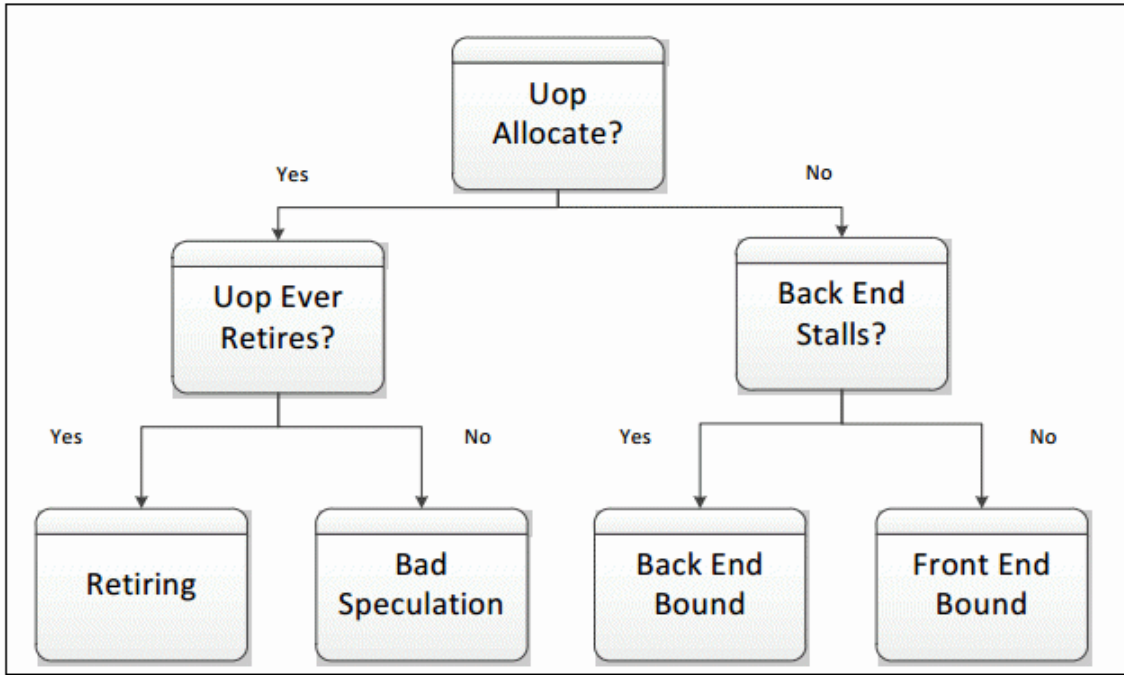


図 B-2 TMAM のトップレベルにおけるドリルダウンのフローチャート

パイプラインの問題となるステージ (アロケーション・ステージ) の単一のエントリー分岐点で、4 つのカテゴリーはスロット全体に付加可能になります。スロット粒度 (副サイクル) での分類は、トップレベルに必要なスーパースカラー・コア向けの正確で堅牢なブレイクダウンを可能にします。

**リタイア**は、“有効な操作” で利用されたスロットを意味します。ここでサイクルあたりの命令数 (IPC) と関連付けてすべてのスロットを表示したいと思うかもしれません。高いリタイア率はスピードアップの余地がないことを意味するものではありません。

**投機の問題**は、誤った投機により浪費されたスロットを示します。次のものが含まれます: (a) 最終的にリタイアしなかった操作のスロット、および (b) 発行されたパイプラインが、先行する誤った投機からの回復によりブロックされたスロット。Branch\_Resteers による 3 番目の問題があることを忘れてはいけません。このカテゴリーは投機のタイプごとに分類できます。例えば、分岐予測ミスとマシニングリアは、制御フローとデータの誤った投機に関連します。

**フロントエンド依存**は、パイプラインのフロントエンドがバックエンドに操作を十分に供給できていないことを示します。フロントエンドは、後続のバックエンドで実行される操作を提供するパイプラインの一部です。このカテゴリーはさらに、フェッチのレイテンシー (命令キャッシュや ITLB ミスなど) とフェッチの帯域幅 (部分的なデコードなど) に分類されます。

**バックエンド依存**は、新しい操作を受け入れるバックエンドのリソース不足により、ストールしたスロットを示します。これは、メモリー・サブシステムによる実行ストールを反映する**メモリー依存**、と実行ユニットへのプレッシャー (計算依存) や命令レベルの並列性 (ILP) 不足を反映する**コア依存**に分類されます。

以降の節では、これらのカテゴリーに関する詳細と階層の次のレベルのノードについて説明します。

### B.1.2 フロントエンド依存

フロントエンドは、分岐予測器が次のフェッチアドレスを予測し、命令キャッシュからコードバイトのストリームをフェッチし、命令をバックエンドで実行されるマイクロオペレーションに解析およびデコードするパイプラインの一部です。**フロントエンド依存**は、プロセッサ・コアのフロントエンドがバックエンドに対し供給不足であることを示します。これは、バックエンドが  $\mu\text{op}$  (マイクロオペレーション) を受け入れる準備ができていないと、フェッチバブルの原因となります。



長くバッファされたパイプラインの初めで生じるフロントエンドの問題に TMAM なしで対処するのは困難です。一時的なこの問題は実際のパフォーマンスに影響しないこともあり、フロントエンドの問題がトップレベルで検出された場合にのみ詳細を調査すべきです。特に IPC が高い場合、フロントエンドの帯域幅はパフォーマンスに影響します。これにより、フェッチ・パイプラインのレイテンシーを隠匿し、デコード済み命令キャッシュ (DSB) と同様にループ・ストリーム検出器 (LSD) などに必要な帯域幅を維持する専用ユニットが追加されました。

TMAM はさらに、フロントエンドにおけるストールのレイテンシーと帯域幅を識別します。

- Y 命令キャッシュミスは、Fetch Latency (フェッチレイテンシー) に分類されます。
- Y 命令デコーダーの非効率性は、Fetch Bandwidth (フェッチ帯域幅) に分類されます。

これらのメトリックはトップダウンのアプローチで定義されることに注意してください。Fetch Latency (フェッチレイテンシー) は、原因にかかわらずフェッチのスタベーション (μop が供給されない) を引き起こす状況をカウントします。これには命令キャッシュと命令 TLB ミスが該当しますが、それだけではありません。Branch Resteers (分岐切り替え) は、パイプラインのフラッシュに続くフェッチの遅延をカウントします。パイプラインのフラッシュは、分岐予測ミスやメモリー・ニュークなどのいくつかのイベントによって発生します。Branch Resteers (分岐切り替え) は、投機の問題と密接に関連しています。

この方法論はさらに、フェッチユニットごとの帯域幅の問題、マイクロオペレーション・キュー (図 2-3 を参照) への μop の挿入などを分類します。命令デコーダーは、一般的に使用される x86 命令をプロセッサが理解できる μop に変換します。これはフェッチ単位で行われます。CPUID などいくつかの x86 命令はマイクロオペレーション・フローを必要とします。これは MSROM から長いマイクロオペレーション・フローが供給され、2 番目のフェッチ単位となります。プロセッサの世代ごとにフェッチユニットの帯域幅が異なる可能性があります。図 2-3 は、Skylake<sup>+</sup> マイクロアーキテクチャーの詳細を示しています。

### B.1.3 フロントエンド依存

**バックエンド依存**は、バックエンドがマイクロオペレーションを受け入れるために必要なリソースが不足しているため、発行パイプラインでマイクロオペレーションが供給されないスロットがあることを示します。このカテゴリーに分類されるパフォーマンスの問題には、データ・キャッシュ・ミスや除算ユニットがオーバーロード状態であることによるストールなどが含まれます。

**バックエンド依存**は、**メモリー依存**と**コア依存**に分類されます。これは、サイクルごとの実行ユニットの占有率に基づいてバックエンドのストールを調査することで達成できます。IPC を最大に維持するため、実行ユニットをビジーに保つ必要があります。例えば、4 スロット幅のマシンでは、3 つ以下のマイクロオペレーションがコードの定常状態で実行される場合、最良である IPC 4 を達成することはできません。これらの部分的な最適サイクルは、ExecutionStalls と呼ばれます。

### B.1.4 メモリー依存

Memory Bound (**メモリー依存**) は、キャッシュとメモリー・サブシステムに関連する実行ストールに相当します。通常、これらのストールは、ロードがすべてのキャッシュをミスした直後のように、実行ユニットが短時間でスタベーションを引き起こす場合に表れます。最近のインテル® Core™ プロセッサ世代の多くは、外部メモリーのレイテンシーを隠匿する 3 階層のキャッシュを備えています。最初のレベルはデータキャッシュ (L1D) です。L2 は 2 番目のレベルの命令とデータを共有する各コアのプライベート・キャッシュです。L3 は、物理パッケージ上のすべてのプロセッサ・コア間で共有されます。

アウトオブオーダーのスケジューラーは、複数の実行ユニットへマイクロオペレーションを供給することができます。マイクロオペレーションがインフライトで実行される間、保留されているメモリー・アクセスに関連しない有効なマイクロオペレーションで実行ユニットをビジーに保つことで、データのメモリー・アクセス・レイテンシーを隠匿することができます。したがって、一般的には、スケジューラーが実行ユニットに対して何も提供する準備ができていない場合、メモリー・アクセスは実際のペナルティーを被ります。さらにマイクロオペレーションが保留されているメモリー・アクセスを待っているか、準備ができていないマイクロオペレーションに依存している可能性もあります。

実行ストールは、特定のキャッシュレベルとそれぞれのキャッシュレベルで扱われるデータ要求に依存する、いくつかのサブカテゴリーに関連しています。ある状況では、ロード要求がキャッシュレベルでミスしない場合、実行ストールはキャッシュ・レベル・レイテンシーよりも長いレイテンシーを被ることがあります。

例えば、L1D では ALU ストール (または浮動小数点加算/乗算、整数乗算などの一般的に使用される実行ユニットの完了を待機) に匹敵する短いレイテンシーがしばしば発生します。しかし、アドレスがオーバーラップした先行するストアからのデータフォワードによりブロックされているロードなどでは、ロードは L1D によって最終的にデータが返されるまで長いレイテンシーを被る可能性があります。そのような場合、インフライトのロードはしばらくの間 L1D を利用することができます。これは L1 依存にタグ付けされます。4K エイリアスによってブロックされるロードは、同様の兆候が見られるもう 1 つのケースです。

さらに、ストア操作に関連する実行ストールは、**ストア依存**のカテゴリーとして扱われます。ストア操作は、メモリーオーダーの要件からバッファリングされ、リタイア後に実行されます。通常、ストア操作は、パフォーマンスにそれほど影響しませんが、完全に無視できるほど小さくはありません。TMAM では、ストア依存を実行ポートの利用率が低く、ストアをバッファリングするたためリソースの消費が高いストア数のサイクルの一部であると定義します。

データ TLB (DTLB) ミスは、各種メモリー依存のサブノードに分類されます。例えば、TLB 変換が L1D で扱われる場合、それは L1 依存にタグ付けされます。

MEM Bandwidth (**メモリー帯域幅**) と MEM Latency (**メモリー・レイテンシー**) を Ext Memory Bound (**外部メモリー依存**) に分類するため、簡単なヒューリスティックが使用されます。このヒューリスティックは、メモリー・コントローラーから返されるデータが保留される要求の占有期間をベースにします。占有期間が高いしきい値を超える (要求最大数の 70%) 場合、メモリー・コントローラーは同時に処理することができ、TMAM はこれをメモリー帯域幅によって潜在的に制限されるとして識別します。残りはメモリー・レイテンシーと分類されます。

## B.1.5 コア依存

**コア依存**は、実行ユニットのプレッシャーやプログラムの命令レベルの並列性 (ILP) の欠如を示します。コア依存のストールは、短期間の命令スタベーションや、特定するのが困難である部分的な実行ポートの利用を表します。例えば、長いレイテンシーの除算操作は、実行ポートに対して特定の `µop` タイプがプレッシャーを与え、サイクルで利用されるポートが減少することからしばらくの間命令スタベーションを引き起こし、実行をシリアル化する可能性があります。

コア依存の問題は、より適切なコードを生成することで軽減できることがあります。例えば、依存関係のある数学計算のシーケンスは、コア依存に分類されます。コンパイラーは、より適切な命令スケジュールによってストールを軽減できる可能性があります。ベクトル化もまたコア依存の問題を緩和できます。

## B.1.6 投機の問題

Bad Speculation (**投機の問題**) は、誤った投機により浪費されたスロットを示します。これには次の 2 つが含まれます。

- Y 最終的にリタイアしなかったマイクロオペレーションの発行に消費されたスロット。
- Y 先行する投機のミスから回復するためブロックされた発行パイプラインのスロット。

例えば、誤って予測された分岐の背後で発行されたマイクロオペレーションは、このカテゴリーに分類されます。誤った予測のペナルティーは、どれくらい迅速に正しいターゲットをフェッチできるかによります。これは、他のフロントエンドのストールをオーバーラップする可能性がある Branch Resteers (**分岐のリステア**) としてカウントされません。

トップレベルでの投機の問題のカテゴリーは、TMAM における重要な課題です。これには、解析で誤った実行パス (ほかのカテゴリーに挙げられている測定の精度に影響する) の影響を受けるワークロードの領域を明らかにします。また、低レベルのノードで、従来のパフォーマンス・カウンター (ほとんどのカウンターイベントが投機的にカウントさ

れる)の使用を可能にします。従って、高い値を示す**投機の問題**を“red flag (レッドフラグ)”として、他のカテゴリーを調査する前に最初に対応すべきです。投機の問題を最小化することは、プロセッサ・リソースの利用を改善するだけでなく、階層を通してレポートされるメトリックの確度を高めます。

TMAM はさらに、**投機の問題**を同じ兆候を示しパイプラインがフラッシュされる**分岐予測ミス**と**マシנקリア**に分類します。分岐予測ミスは、BPU が分岐方向と (または) ターゲットを誤って予測した場合に適用されます。メモリー・オーダー・クリア (例えば、メモリー・ディスアンビゲーション) による誤ったデータ投機は、マシנקリアのサブセットです。これらの問題を解析する次のステップは完全に異なることがあります。メモリー・オーダーによるマシנקリアや自己修正コードなど、以降に予期しない状況を招くコードでは、最初に、プログラムの制御フローが分岐予測に適するようにします。

### B.1.7 リタイア

このカテゴリーは、発行されたマイクロオペレーションがパフォーマンス・ボトルネックを発生させることなく素早くリタイアした、“適切なマイクロオペレーション”によって利用されたスロットを示します。理想的には、すべてのスロットが**リタイア**カテゴリーの属性となることが望まれます。すべてのスロットの 100% リタイアは、マイクロアーキテクチャーのサイクルごとに最大のマイクロオペレーションがリタイアしたことに相当します。例えば、1 つの命令が 1 マイクロオペレーションにデコードされる場合、1 スロットでの 50% のリタイアは、4 ワイドのマシンでは IPC 2 であることを示します。言い換えると、**リタイア**のカテゴリーを最大化することでプログラムの IPC を高めることができます。

高いリタイア値はより高いパフォーマンスの余地がないことを意味するものではありません。浮動小数点 (FP) アシストなどのマイクロコード・シーケンスはパフォーマンスを損ねますが、回避することができます。これらは注意を促すため、MSROM のカテゴリーに分類されます。

ベクトル化されていないコードでの高いリタイア値は、ベクトル化の候補となります。これを実現するには、単一の命令/マイクロオペレーションでより多くの操作を完了することです。その結果パフォーマンスが向上します。TMAM は、さらに**リタイア** -> **基本カテゴリー**を**スカラー**と**ベクトル操作**による **FP 計算**に分類します。詳細については、行列乗算の利用例 2 をご覧ください。

### B.1.8 TMAM と Skylake<sup>+</sup> マイクロアーキテクチャー

Skylake<sup>+</sup> マイクロアーキテクチャーのパフォーマンス監視機能は、以前の世代と比べ大幅に拡張されています。TMAM は、カウンターイベントの多様性とプリサイズ・イベントベースのサンプリング (PEBS) 機能の拡張の利点を得られます。図 B-3 は、Skylake<sup>+</sup> マイクロアーキテクチャーにおける TMAM のサポートを示しています (緑のボックスは PEBS が利用可能です)。

インテル® VTune™ Amplifier は、多くのインテル® マイクロアーキテクチャーで TMAM を適用することを可能にします。<https://www.isus.jp/products/vtune/help-GUID-02271361-CCD4-410C-8338-4B8158157EB6/>にある技術文書で、さらに詳しい情報をご覧ください。

#### B.1.8.1 TMAM の例

12.15.1 節では、レイテンシーに関連する浮動小数点演算の最適化テクニックと、FP MUL、FP ADD、および FMA 命令のスルーputtに関する考慮事項を説明しています。FP\_ADD と FP\_MUL 命令のレイテンシーの問題を直接検出するパフォーマンス・カウンター・イベントはありません。

TMAM は、この問題がパフォーマンスを制限する可能性があることを理解するのに使用できます。

主なボトルネックがバックエンド依存 -> コア依存 -> ポート利用であり、GFLOPS メトリックにおいて顕著な測定があれば、コードはこの問題に遭遇している可能性があります。12.15.1 節に示される最適化を考慮してみても良いでしょう。

12.3.1 節では、Skylake<sup>+</sup> マイクロアーキテクチャーにおいて YMM の上位がダーティーである場合にインテル® SSE コードを実行するパフォーマンスの問題について説明しています。パーシャルレジスターの依存性と、インテル® SSE コード実行時のブレンドコストに関するパフォーマンスの問題を特定するには、ソースコードが直接インテル® AVX 命令を実行せずパフォーマンス上重要なインテル® SSE コードでのインテル® SSE 命令とインテル® AVX 命令の混在比率をモニターするのに TMAM を使用できます。

主なボトルネックが、バックエンド依存 -> コア依存である場合、Vector-MixRate (ベクトルミックス比率) メトリックに顕著な測定が見られれば、YMM レジスター上位の混在操作によるベクトル幅の不一致なベクトル操作が存在する可能性を示します。

VectorMixRate (ベクトルミックス比率) メトリックには、Skylake<sup>+</sup> マイクロアーキテクチャーで利用可能な UOPS\_ISSUED.VECTOR\_WIDTH\_MISMATCH イベントが必要です。このイベントは、ベクトルレジスターの上位ビットを保持するため、発行ステージで挿入されたマイクロオペレーションをカウントします。このイベントは、ベクトルレジスターの上位ビットを維持するため、リソース・アロケーション・テーブル (RAT) からリザベーション・ステーション (RS) へ発行されたブレンド・マイクロオペレーションの数をカウントします。

さらに、メトリックは分母として、最近のインテル® マイクロアーキテクチャーでは一般的な UOPS\_ISSUED.ANY を使用します。UOPS\_ISSUED.ANY イベントは、RAT が RS へ発行したマイクロオペレーションの総数をカウントします。

VectorMixRate メトリックは、発行されたすべてのマイクロオペレーションのうち、挿入されたブレンド・マイクロオペレーションの比率を示します。通常、VectorMixRate が 5% を上回ると調査の必要があります。

$$\text{VectorMixRate}[\%] (\text{ベクトルミックス比率}) = 100 * \text{UOPS\_ISSUED.VECTOR\_WIDTH\_MISMATCH} / \text{UOPS\_ISSUED.ANY}$$

実際のペナルティーは、挿入されたブレンド操作が追加するデスティネーション・レジスターとのデータ依存関係から生じるため、一定ではありません。

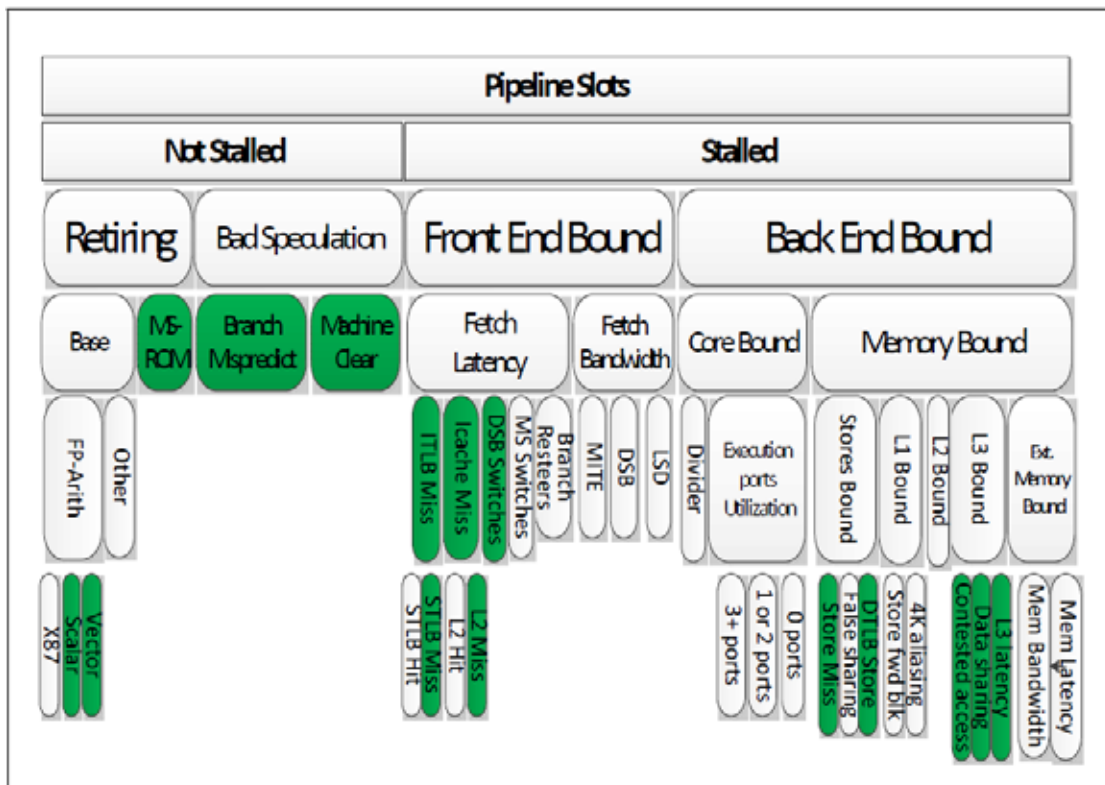


図 B-3 Skylake<sup>+</sup> マイクロアーキテクチャーでサポートされる TMAM 階層

## B.2 パフォーマンス監視とマイクロアーキテクチャー

この節では、Silvermont<sup>†</sup>、Airmont<sup>†</sup> および Goldmont<sup>†</sup> マイクロアーキテクチャーに関連するパフォーマンス監視ハードウェアと用語について説明します。ここで説明する特性は、表 B-1 に示すようにマイクロアーキテクチャー固有である可能性があります。

表 B-1 パフォーマンス監視の分類

名前	説明	マイクロアーキテクチャー
L2Q, XQ	<p>メモリー参照が L1 データキャッシュをミスした場合、L2 キュー (L2Q) へ要求が送られます。要求が L2 キャッシュでもミスすると要求は XQ へ送られ、ダイ上のインターフェイス (IDI) リンクを介してメモリーへ発行されるのを待機します。L2 キャッシュは、プロセッサ・コアのペアで共有されるため、単一の L2Q もこれら 2 つのコアで共有されることに注意してください。同様に、L2Q と IDI リンクの間位置するプロセッサ・コアのペアには単一の XQ があります。</p> <p>XQ に到達する新しい要求よりも IDI リンクからの要求が少ない場合、XQ は一杯になります。イベント L2_reject_XQ は、XQ が一杯であるため要求を L2 から XQ に送ることができないことを示し、その結果、メモリー・サブシステムがオーバーサブスクライブであることを示します。</p>	Silvermont <sup>†</sup> 、 Airmont <sup>†</sup> 、 Goldmont <sup>†</sup>
Core Reject	<p>core_reject イベントは、コアからの要求が L2Q によって受け入れられなかったことを示します。要求が L2Q によって拒否されるにはいくつかの理由があります。L2Q が一杯であると以降の要求は拒否されます。つまり、一方のコアの要求を拒否して、他のコアの公平性を維持することができます。つまり、あるコアが L2Q/キャッシュ/XQ/IDI リンクへの共有接続要求を占有することが許可されない、L2Q に余地があっても要求は拒否される可能性があります。さらに、コアからの要求がダーティーな L1 キャッシュの追い出しである場合、ハードウェアはこの追い出しが L2Q 内のすべての保留要求と競合しないことを保証する必要があります (保留要求には外部スヌープが含まれます)。イベントが競合すると、ダーティー追い出し (eviction) 要求は L2Q に余地があっても拒否される可能性があります。</p> <p>そのため、L2_reject_XQ イベントは、両方のコアからメモリーへの要求比率がメモリーの応答比率を超えていることを示しますが、Core_reject イベントは微妙です。これは、L2Q への要求比率が XQ からの応答比率を超えていることを示すか、L2Q への要求比率が L2 からの応答比率を超えていることを示すか、またはコアが L2Q からの応答の妥当な共有数を超えた要求を試みていることを示します。また、それはダーティー追い出しと他の保留要求の競合を示すかもしれません。</p> <p>つまり、L2_reject_XQ イベントはメモリーのオーバーサブスクリプション状態を示します。Core_reject イベントは、次のいずれかを示します。(1) メモリーのオーバーサブスクリプション、(2) L2 のオーバーサブスクリプション、(3) 他のコアへの公正性を保証するため一方のコアの要求を拒否、(4) ダーティー追い出しと他の保留要求間の競合。</p>	Silvermont <sup>†</sup> 、 Airmont <sup>†</sup> 、 Goldmont <sup>†</sup>

<p>Divider Busy (除算器ビジー)</p>	<p>すでにディスパッチされている除算 <math>\mu\text{op}</math> を処理中である場合、除算ユニットは新しい除算 <math>\mu\text{op}</math> を受け入れることができません。                  "CYCLES_DIV_BUSY.ANY" イベントは、除算ユニットがビジーであるサイクルをカウントします。これは、別の除算マイクロオペレーションが、除算ユニットへの割り当て (RS から) を待機しているかどうかにかかわらずカウントされます。このイベントでは、RS が空であっても除算が処理中であればサイクルがカウントされます。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup>, Goldmont<sup>†</sup></p>
<p>BACLEAR</p>	<p>命令をデコードして分岐/コール/ジャンプ/リターン命令を識別した直後、分岐アドレス計算クリア (BACLEAR) イベントの発生が可能となります。BACLEAR が発生する原因には、誤った直接分岐のターゲット予測や命令の位置で分岐が予測されていないなどが含まれます。</p> <p>BACLEAR は、フロントエンドに異なる位置からフェッチを再開させます。BACLEAR は、パイプラインの一部を実行することで識別される分岐予測ミスと類似していますが、BR_MISP_RETIRED イベントはカウントされない、または LBR (LBR が予測ミスを報告) での予測ミスと示されます。分岐予測ミスと BACLEAR は、新しいターゲット位置から命令のフェッチを始めるためフロントエンドを再起動し、投機されたワークをフラッシュするという点では同じです。しかし、分岐予測ミスでは、フロントエンドとバックエンドの両方から命令を完全にフラッシュする必要があるのに対し、BACLEAR は、デコード時に発生するため命令バイトをフラッシュしますが、完全にデコードされていない命令はフラッシュしません。BACLEAR 後のリカバリーは、分岐予測ミス後のリカバリーよりも単純で高速です。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup>, Goldmont<sup>†</sup></p>
<p>フロントエンドのボトルネック</p>	<p>フロントエンドは、命令をフェッチし、マイクロオペレーション (<math>\mu\text{op}</math>) ヘデコードし、バックエンドで処理するためマイクロオペレーション・キューへそれらを送出する役割を持ちます。次にバックエンドは、それらのマイクロオペレーションを取得して必要なリソースを割り当てます。すべてのリソースの準備が整うと、マイクロオペレーションが実行されます。フロントエンドのボトルネックは、マシンのフロントエンドがマイクロオペレーションをバックエンドへ供給できない場合 (バックエンドはストールせず) に発生します。バックエンドの準備ができていないためマイクロオペレーションを受け入れることができないサイクルは、フロントエンドのボトルネックとしてカウントされません。バックエンドのボトルネックはアロケーション・ユニットのストールを引き起こし、その結果フロントエンドはバックエンドがマイクロオペレーションを受け入れるようになるまで待機します。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup>, Goldmont<sup>†</sup></p>
<p>NO_ALLOC_CYCLES</p>	<p>フロントエンドの問題は、このイベントクラスの各種サブイベントを使用して分析できます。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup></p>
<p>UOPS_NOT_DELIVERED.ANY</p>	<p>UOPS_NOT_DELIVERED.ANY イベントは、マシンが確実にフロントエンド依存であることを特定するため、フロントエンドの効率を測定するのに使用されます。非効率なフロントエンドの例には次のものがあります: フロントエンドの帯域幅を制限する命令キャッシュミス、ITLB ミス、およびデコーダーの制限。</p>	<p>Goldmont<sup>†</sup></p>

<p>ICache (命令キャッシュ)</p>	<p>命令キャッシュ (ICache) への要求は、チャンクと呼ばれる固定サイズ単位で行われます。キャッシュラインには複数のチャンクがあるため、複数のアクセスが同じキャッシュラインに集中する可能性があります。</p> <p>Goldmont<sup>†</sup> マイクロアーキテクチャーでは、このイベントはキャッシュライン単位でカウントされるため、単一のキャッシュラインへの複数のフェッチは 1 つの ICACHE.ACCESS、1 つの HIT または 1 つの MISS としてカウントされます。特に、直線的なコードがキャッシュラインにまたがるか、分岐ターゲットが新しいラインである場合、イベントはカウントされます。このイベントは、本質的にかなり投機的であり、デコード、実行、およびリタイアの前に命令バイトがフェッチされます。投機は分岐と同様に直線的なコードでも行われることがあります。その結果、リタイアした命令数を調査することでは、命令キャッシュの特性は推定できません。</p> <p>Silvermont<sup>†</sup> マイクロアーキテクチャーでは、ICACHE イベント (HIT、MISS) は異なる粒度でカウントされます。</p>	<p>Goldmont<sup>†</sup></p>
<p>ICache (命令キャッシュ) アクセス</p>	<p>命令キャッシュのフェッチアクセスは、固定サイズのチャンクにアライメントされます。命令キャッシュからの特定チャンクのフェッチ要求は、投機実行により複数回現れる可能性があります。同じチャンクが実行中の状態で複数回要求される可能性もあります。しかし、命令フェッチミスは一度だけカウントされ、処理中のすべてのサイクルをカウントするわけではありません。</p> <p>命令キャッシュミスがラインをフェッチした後、同じキャッシュラインへの別の要求が再び行われ、それはヒットと見なされます。そのため、"ヒット" + "ミス" の値は、アクセス数と等しくなりません。</p> <p>ソフトウェアの観点から命令キャッシュヒットの正確な数を取得するには、命令キャッシュヒットから命令キャッシュミスの値を減算すべきです。</p>	<p>Silvermont<sup>†</sup>、 Airmont<sup>†</sup>、 Goldmont<sup>†</sup></p>
<p>ラスト・レベル・キャッシュ参照、ミス</p>	<p>L3 を持たないプロセッサでは、L2 がラスト・レベル・キャッシュとなります。LLC 参照とミスをカウントするアーキテクチャーのパフォーマンス・イベントは、L2_REQUESTS.ANY と L2_REQUESTS.MISS です。</p>	<p>Silvermont<sup>†</sup>、 Airmont<sup>†</sup>、 Goldmont<sup>†</sup></p>
<p>マシנקリア</p>	<p>マシנקリアを引き起こす可能性がある多くの状態があります (割り込み、トラップ、フォルトの受信を含みます)。それらすべての条件 (MO (メモリーオーダー)) を含みますが限定はされません。SMC (自己またはクロス修正コード)、および FP (浮動小数点アシスト) は、MACHINE_CLEAR.ANY イベントで検出されます。さらに、いくつかの条件 (SMC、MO、FP など) が明確にカウントされます。しかし、SMC、MO、および FP マシנקリアの合計が ANY 数と等しくなるわけではありません。</p>	<p>Silvermont<sup>†</sup>、 Airmont<sup>†</sup>、 Goldmont<sup>†</sup></p>
<p>MACHINE_CLEAR.FP_ASSIST</p>	<p>ほとんどの場合、浮動小数点実行ユニットは適切な出力ビットを生成します。命令に対してマシנקリアが発生する場合など、まれにヘルプを必要とすることがあります。マシנקリアが発生すると、マシンのフロントエンドはどのような浮動小数点操作が必要であるかを特定する命令を供給し、正しい結果を生成するため特別なコードを実行します (例えば、結果が浮動小数点デノーマル値であれば、ハードウェアは IEEE に準拠する適切な丸めを行うようにアシストを要求します)。</p>	<p>Silvermont<sup>†</sup>、 Airmont<sup>†</sup>、 Goldmont<sup>†</sup></p>

MACHINE_CLEAR.SMC	<p>自己修正コード (SMC) とは、マシンが実行する命令ストリームにコードの一部を書き込むことを指します。Silvermont<sup>†</sup> マイクロアーキテクチャーでは、プロセッサが 1K でアライメントされた領域内で SMC を検出します。SMC 条件が検出されるとマシン・クリア・アシストが発生し、パイプラインがフラッシュされます。</p> <p>プロセッサが実行しているコードの 1K 以内のメモリーに書き込みを行うと、SMC 検出機能が起動されマシンクリアが発生します。マシンクリアによってストア・パイプラインが排出されるため、フロントエンドが再起動して正しい命令を (書き込み後) 実行できます。</p>	Silvermont <sup>†</sup> , Airmont <sup>†</sup> , Goldmont <sup>†</sup>
MACHINE_CLEAR.MO	<p>メモリー・オーダー・マシン・クリアは、スヌープ要求が発生してメモリーの順番を維持できるかどうかマシンが不確実である場合に起こります。例えば、2 つのロードがあると仮定し、プログラムの順番で 1 つがアドレス X を指定し、もう 1 つがそれに続く Y を指定します。両方のロードが発行されますが、Y のロードが最初に完了し、このロードと依存関係を持つマイクロオペレーションが処理を続行します。X へのロードはデータを待機しています。同時に別のプロセッサが Y と同じアドレスに書き込みを行い、アドレス Y へのスヌープが発生すると仮定します。</p> <p>ここで問題が生じます。X のロードがまだ完了していなかったため、Y のロードは古いを取得しています。他のプロセッサは、アドレス Y にストアされた最新の値を利用しないことで、異なる順番でロードを認識します。書き込み後のデータを取得するには、アドレス Y からのロードをすべて元に戻す必要があります。</p> <p>注意: 他に保留されている読み込みがなければ、ロード Y を元に戻す必要はありません。ロード X が完了していないことが、この順序付けの問題を引き起こします。</p>	Silvermont <sup>†</sup> , Airmont <sup>†</sup> , Goldmont <sup>†</sup>
MACHINE_CLEAR.DISAMBIGUATION	<p>ディスアンビゲーション・マシン・クリアは、先行するストアと同じアドレスをもつロードがストアを追い越すことで引き起こされます。</p>	Goldmont <sup>†</sup>



<p>ページウォーク</p>	<p>リニアアドレスから物理アドレスへの変換の際にトランスレーション・ルックアサイド・バッファ (TLB) に変換済みのアドレスが見つからない場合、必要であれば専用ハードウェアがページテーブルと他のページ構造から物理アドレスを検索しなければいけません。これをページウォークといいます。ページウォークが完了すると、変換情報は以後の処理のため TLB に格納されます。</p> <p>ページング構造はメモリーに保存されているため、ページウォークには複数のメモリーアクセスが必要となることもあります。ページウォークが命令参照を変換した場合、これらのメモリーアクセスは要求されたデータの一部であると考えられます。ページウォークのサイクル数は可変であり、必要なメモリーアクセスの回数とこれらのメモリーアクセスのキャッシュ局所性に依存します。</p> <p>PAGE_WALKS イベントは、EDGE トリガービットがクリアされてからページウォークにかかった時間をカウントします。ページウォーク時間をページウォークの回数で割ると平均時間が求められます。</p> <p>Goldmont<sup>†</sup> マイクロアーキテクチャーでは、MEM_UOPS_RETIRED.DTLB_MISS と ITLB.MISS イベントを使用してページウォークの回数を測定できます。</p> <p>Silvermont<sup>†</sup> マイクロアーキテクチャーでは、PAGE_WALKS.WALKS によってデータと命令が混在したページウォークの回数がカウントされます。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup>, Goldmont<sup>†</sup></p>
<p>RAT</p>	<p>RAT は、フロントエンドからバックエンドへマイクロオペレーションを転送する割り当てパイプラインです。割り当てパイプラインの最後で、マイクロオペレーションは 6 つのリザベーション・ステーション (RS) のいずれかに書き込まれます。各 RS は、特定の実行 (またはメモリー) クラスタへ送られるマイクロオペレーションを保持します。RS の数には制限があるため、実行クラスタにマイクロオペレーションを送出できない場合、それらは蓄積される可能性があります。RS が一杯になる典型的な理由として、除算のような長い実行レイテンシー、依存関係によりマイクロオペレーションをスケジューリングできない、また多くのメモリー参照が実行中である、などがありますがこれらは限定的ではありません。RS が一杯になると、それ以上のマイクロオペレーションを受け入れることができなくなり、割り当てパイプラインをストールさせます。RS_FULL_STALL.ANY イベントは、RS のいずれかが一杯になったことで割り当てがストールするサイクルが生じるとカウントされます。割り当てパイプラインはいくつかの原因でストールする可能性があります。RS フルが原因でない場合 RS_FULL_STALL.ANY はカウントされません。MEC サブイベントは、MEC RS フルがさらに割り当てを妨げるかどうか判断するのを可能にします。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup>, Goldmont<sup>†</sup></p>

<p>REHABQ</p>	<p>REHABQ は、1 つまたは複数の理由により、完了できないマイクロオペレーションのメモリー参照を保持する内部のキューです。マイクロオペレーションは、再発行され正常に完了するまで REHABQ に残っています。</p> <p>マイクロオペレーションが REHABQ に格納されるボトルネックの原因としては、キャッシュライン分割、ブロックされたストアフォワード、およびデータの準備ができていない、などがありますが限定的ではありません。ロードやストアが REHABQ に送られるには、多くの状況が考えられます。例えば、先行するストアのストアアドレスが不明である場合、そのアドレスが判明するまで後続のストアはすべて REHABQ に送られる必要があります。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup></p>
<p>LOAD_BLOCKS</p>	<p>ロードは多くの原因でブロックされる可能性があります。原因は限定的ではありませんが、UTLB ミス、ブロックされたストアフォワード、4K エイリアシングなどが上げられます。ロードが先行するストアで保存されるデータ (全体もしくは部分的) を必要とする場合、マシンのフォワード処理は次の 2 つの状況に直面します: (i) 先行するストアの完了を待機する (フォワードは制限され、ロードはブロックされる) か、(ii) ストアが完了する前にデータをロードにフォワードできる。制限される状況は次のように説明されます。</p> <p>先行するストアに対しロードをチェックする場合、アドレスビットのすべてがストアアドレスと比較されるわけではありません。アドレスが未定のストアと類似 (LD_BLOCKS.4K_ALIAS) しているため、ロードはブロックされますが技術的にはブロックする必要はありません。ロードが先行するストアからデータを受け取ることができない場合、ロードは未定のストアが完了するまでブロックされます。LD_BLOCKS.STORE_FORWARD は、アドレスの不一致 (以下で説明) によってストアからフォワードされたデータの受信が制限されたロードの回数をカウントします。</p> <p>LD_BLOCKS.DATA_UNKOWN は、その時点でストアデータが利用可能でないため、ストアフォワードによりブロックされたロードをカウントします。ロードブロックは、LD_BLOCKS.DATA_UNKNOWN および LD_BLOCK.STORE_FORWARD としてカウントされません。表 16-12 に、ロードが先行するストアからデータを受けとることが可能な条件を示します。</p> <p>これらのイベントはプリサイスイベントであるため、リタイアしない投機的なロードはカウントしません。</p>	<p>Goldmont<sup>†</sup></p>
<p>μop リタイア</p>	<p>プロセッサは、複雑なマクロ命令を単純なマイクロオペレーションのシーケンスにデコードします。ほとんどの命令は 1 もしくは 2 つのマイクロオペレーションで構成されます。いくつかの命令は長いマイクロオペレーションのシーケンスにデコードされます (浮動小数点超越関数命令、アシスト、文字列リピート命令など)。</p> <p>ある状況では、マイクロオペレーションのシーケンスが単独のマイクロオペレーションに融合されるか、命令全体が融合されます。Goldmont<sup>†</sup> では、MSROM マイクロオペレーションを区分するため UOPS_RETIRED にサブイベントがあります。このサブイベントは他のマイクロアーキテクチャーとは異なります。</p>	<p>Silvermont<sup>†</sup>, Airmont<sup>†</sup>, Goldmont<sup>†</sup></p>

HW_INTERRUPTS	これらのイベントは、ハードウェア割り込みに関する情報を提供します。HW_INTERRUPTS.RECEIVED は、プロセッサが受信したハードウェア割り込みの総数をカウントします。このイベントは ROB が認識する割り込みカウントの総数に直結します。HW_INTERRUPTS.PENDING_AND_MASKED は、EFLAGS.IF が 0 であるため割り込みが上げられず保留されているコアのサイクル数をカウントします。しかし、TPR や ISR によってマスクされている割り込みはカウントされません。これらはプリサイズイベントではありませんが、これらのイベントで非プリサイズ PEBS レコードを収集することは、無反応状態を引き起こす問題を特定するのに有効です。	Goldmont <sup>†</sup>
MEM_UOPS_RETIRED	これらのイベントは、マイクロオペレーションのリタイアが有効である場合、命令がデータの読み込み (ロード)、または書き込み (ストア) を行うマイクロオペレーションをカウントします。投機的なロードとストアはカウントされません。サブイベントでは、操作を完了するために必要な付加サイクルの状態を示すことができます。特に、メモリー・マイクロオペレーションのアドレスがデータ変換ルックアサイド・バッファー (DTLB) をミスすると、要求されたデータはキャッシュラインに分割されているか、メモリー・マイクロオペレーションはロックされたロードです。これはプリサイズイベントであるため、PEBS レコードの EventingRIP フィールドはイベントの引き起こした命令を示します。	Silvermont <sup>†</sup> 、 Airmont <sup>†</sup> 、 Goldmont <sup>†</sup>
MEM_LOAD_UOPS_RETIRED	これらのイベントは、マイクロオペレーションのリタイアが有効である場合、命令がデータを読み込む (ロード) マイクロオペレーションを生成するのをカウントします。投機的なロードはカウントされません。このイベントは、要求されたデータのメモリー階層での各種状態をレポートし、データのレイテンシー・ストールの原因を特定するのに役立ちます。これはプリサイズイベントであるため、PEBS レコードの EventingRIP フィールドはイベントの引き起こした命令を示します。	Goldmont <sup>†</sup>

### B.3 インテル® Xeon® プロセッサ 5500 番台

インテル® Xeon® プロセッサ 5500 番台は、インテル® Core™ i7 プロセッサと同じマイクロアーキテクチャーをベースにしています (2.6 節「インテル® マイクロアーキテクチャー開発コード名 Nehalem」を参照)。また、インテル® Xeon® プロセッサ 5500 番台は、2 ウェイ・プロセッサで構成される Non-Uniform Memory Access (NUMA) をサポートします (図 B-4 を参照)。図 B-4 に、各物理プロセッサの 4 つのプロセッサ・コアと 1 つのアンコア・サブシステムを示します。

L3 アンコア・サブシステムは、統合型メモリー・コントローラー (IMC) とインテル® QuickPath インターコネクト (インテル® QPI) インターフェイスで構成されます。メモリー・サブシステムは、各 IMC にローカルに接続された 3 チャンネルの DDR3 メモリーで構成されます。非ローカルな IMC に接続された物理メモリーへのアクセスは、リモート・メモリー・アクセスと呼ばれます。

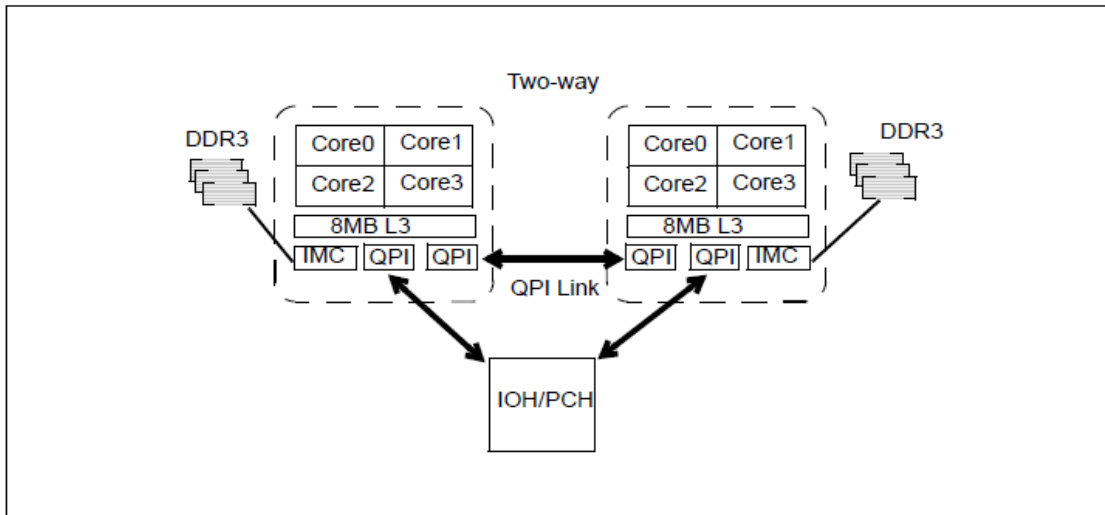


図 B-4 インテル® Xeon® プロセッサ 5500 番台でサポートされるシステムトポロジー

インテル® Xeon® プロセッサ 5500 番台のパフォーマンス監視イベントは、ソフトウェア（コードおよびデータ）とマイクロアーキテクチャー・ユニット間の相互作用を階層的に分析するために使用できます。

- Y コアごとの PMU: 各プロセッサ・コアは、4 つのプログラム可能なカウンターと 3 つの固定カウンターを備えています。プログラム可能なコアごとのカウンターは、フロントエンド/マイクロオペレーション (μop) のフロー問題と、プロセッサ・コア内のストールを調査するように構成できます。また、コアごとの PMU イベントのサブセットは、プリサイス・イベントベース・サンプリング (PEBS) をサポートします。ロード・レイテンシーの測定機能は、インテル® Core™ i7 プロセッサとインテル® Xeon® プロセッサ 5500 番台の新機能です。
- Y アンコア PMU: アンコア PMU は、8 つのプログラム可能なカウンターと 1 つの固定カウンターを備えています。プログラム可能なアンコアのカウンターは、L3 とインテル® QPI の操作およびローカル/リモートデータのメモリアクセスの特性評価を行うように構成できます。

インテル® Xeon® プロセッサ 5500 番台で利用できるさまざまなパフォーマンス・カウンターとプログラム可能な広範なパフォーマンス・イベントにより、ソフトウェアを最適化する開発者はパフォーマンスの問題を分析してパフォーマンスを高めることができます。パフォーマンス・イベントによるパフォーマンスに関する問題の分析は、次の分野にまとめることができます。

- Y サイクル・アカウンティングとマイクロオペレーション (μop) フロー
- Y ストールの区分とコア・メモリアクセス・イベント (非 PEBS)
- Y プリサイス・メモリアクセス・イベント (PEBS)
- Y プリサイス分岐イベント (PEBS, LBR)
- Y コア・メモリアクセス・イベント (非 PEBS)
- Y その他のコアイベント (非 PEBS)
- Y フロントエンドの問題
- Y アンコアイベント

## B.4 インテル® Xeon® プロセッサ-5500 番台のパフォーマンス分析手法

この節で説明する手法は、実行時に測定可能なパフォーマンスのボトルネックを排除または軽減する可能性を特定することに注目します。コンパイル時およびソースコード・レベルの手法は、本書のほかの章で説明します。各項では、パフォーマンス監視イベントから直接測定または計算可能なさまざまなメトリックを調べることで、チューニングの可能性を特定するため具体的な手法について触れていきます。

### B.4.1 サイクル・アカウンティングとマイクロオペレーション (μop) フロー

表 B-2 に、基本的なサイクル・アカウンティング手法の目的、パフォーマンス・メトリック、イベントを示します。

表 B-2 サイクル・アカウンティングおよびマイクロオペレーション (μop) フローの構成

概要	
目的	重大なストールが発生したコード/基本ブロックを特定
方法	「生産的」および「非生産的」領域へバイナリーのサイクルを区分
PMU とパイプラインの焦点	実行のため発行されたマイクロオペレーション (μop)
イベントコード/ Umask	マイクロオペレーション (μop) を実行する場合は、イベントコード B1H、Umask = 3FH 合計サイクル数をカウントする場合は、イベントコード 3CH、Umask = 1、CMask = 2
EvtSelc	サイクルの数をカウントし、アクティブなサイクルとストールサイクルを分離する場合は、 CMask、Invert、Edge フィールドを使用
基本式	“合計サイクル数” = UOPS_EXECUTED.CORE_STALLS_CYCLES + UOPS_EXECUTED.CORE_ACTIVE_CYCLES
メトリック	UOPS_EXECUTED.CORE_STALLS_CYCLES / UOPS_EXECUTED.CORE_STALLS_COUNT
ドリルダウンの範囲	カウント: ワークロード、サンプリング: 基本ブロック
バリエーション	計算主体のマイクロオペレーション (μop) は、ポート 0、1、5 でサイクルをカウント

実行されたマイクロオペレーション(μop)のサイクル・アカウンティングは、パフォーマンス・チューニングにおけるストールサイクルの特定に効果的な手法です。マイクロアーキテクチャー・パイプライン内の、「発行された」、「ディスパッチされた」、「実行された」、「リタイアした」マイクロオペレーション (μop) の意味は厳密に定義されています。これを図 B-5 に示します。

サイクルは、マイクロオペレーション (μop) が実行ユニットにディスパッチされているサイクルと、ディスパッチされていないサイクルに分けられますが、後者は実行ストールであると考えられます。

テスト用コードに対して実行された「合計サイクル数」は、CPU\_CLK\_UNHALTED.THREAD (イベントコード 3CH、Umask = 1) と IA32\_PERFEVTSELc を設定 (CMask = 2、INV = 1) して直接測定できます。

(ポート 2、3、4 で) 実行されたメモリー・アクセス・マイクロオペレーション (μop)をカウントするために使用されたシグナルは、論理プロセッサ単位で測定できないコアイベントのみです。イベントコード B1H と Umask=3FH の組み合わせでは、コア単位でのみカウントされるため、実行ストールの合計数はコア単位でのみ評価できます。HT が無効になっている場合は、マイクロオペレーション (μop) フロー・サイクル・アカウンティングの分析をスレッド単位で容易に実行できます。

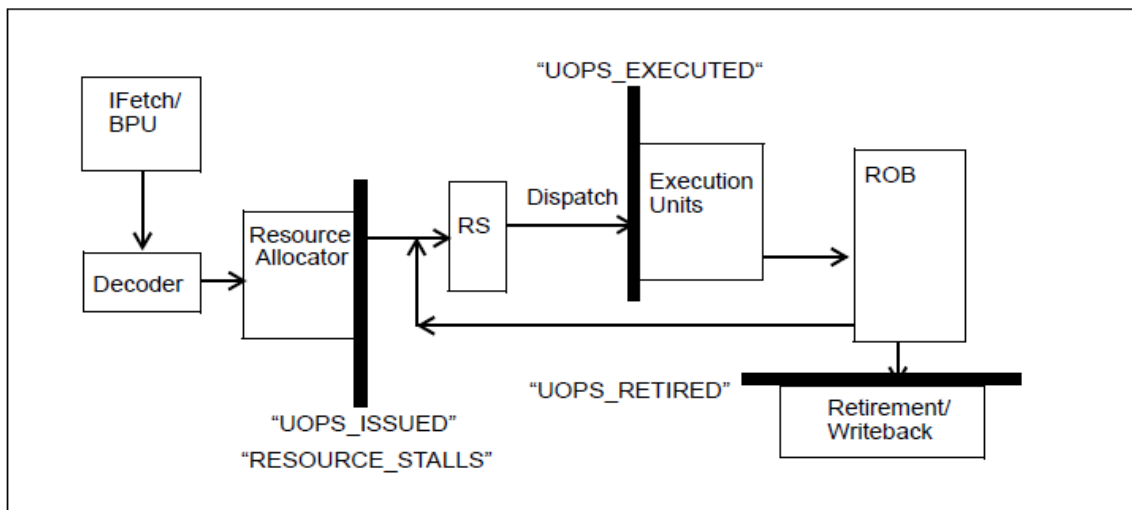


図 B-5 パイプラインにおける PMU 固有のイベントロジック

ポート 0、1、5 で UOPS\_EXECUTED をカウントする PMU シグナルは、HT が有効であってもスレッド単位でカウントできます。これにより、テスト中のワークロードが HT と相互に作用する場合、代替のサイクル・アカウンティング手法が提供されます。

この代替メトリックは、適切な CMask、Inv、Edge 設定を使用して、UOPS\_EXECUTED.PORT015\_STALL\_CYCLES から構成されます。表 B-3 にパフォーマンス・イベントの詳細を示します。

表 B-3 マイクロオペレーション (μop) フロー用イベントの CMask/Inv/Edge/スレッドの粒度

イベント名	Umask	イベント コード	Cmask	Inv	Edge	すべての スレッド
CPU_CLK_UNHALTED.TOTAL_CYCLES	0H	3CH	2	1	0	0
UOPS_EXECUTED.CORE_STALLS_CYCLES	3FH	B1H	1	1	0	1
UOPS_EXECUTED.CORE_STALLS_COUNT	3FH	B1H	1	1	1	1
UOPS_EXECUTED.CORE_ACTIVE_CYCLES	3FH	B1H	1	0	0	1
UOPS_EXECUTED.PORT015_STALLS_CYCLES	40H	B1H	1	1	0	0
UOPS_RETIRED.STALL_CYCLES	1H	C2H	1	1	0	0
UOPS_RETIRED.ACTIVE_CYCLES	1H	C2H	1	0	0	0

### B.4.1.1 サイクルのドリルダウンと分岐予測ミス

実行されたマイクロオペレーション (μop) は、供給されている実行ユニットの観点から見れば生産的であると考えられますが、このようなマイクロオペレーション (μop) がすべてプログラムの処理の進行に貢献するわけではありません。分岐予測ミスは、アウトオブオーダー (OOO) プロセッサ内で非効率な実行を行う可能性があり、通常、これは次の 3 つの要素に分類されます。

- Y 不適切に予測されたパスのマイクロオペレーション (μop) の実行に関連する無駄な処理。
- Y 不適切なマイクロオペレーション (μop) のパイプラインがフラッシュされる際に失われるサイクル。
- Y 不適切なマイクロオペレーション (μop) が実行ユニットに到達するのを待機している間に失われるサイクル。

インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサには、誤って予測されたマイクロオペレーション (μop) のパイプラインのクリア (2 番目の要素) に関連する実行ストールはありません。このようなマイクロオペレーション (μop) は、実行やディスパッチをストールさせることなく、簡単にパイプラインから排除されます。そのため、通常、予測が外れた分岐のペナルティーは低下します。また、命令スタベーション (3 番目の要素) に関連するペナルティーを測定できます。

実行されたマイクロオペレーション (μop) において無駄になる処理は、リタイアしないマイクロオペレーション (μop) です。これは分岐予測ミスに関連するコストの一部であり、パイプラインを通過するマイクロオペレーション (μop) フローを監視することによって検出できます。マイクロオペレーション (μop) フローは、図 B-5 の 3 つのポイント、イベント UOPS\_ISSUED によって RS に入るとき、UOPS\_EXECUTED によって実行ユニットに入るとき、そして UOPS\_RETIRED によってリアイアするときに測定できます。誤って予測されたマイクロオペレーション (μop) に関連する無駄な処理は、アップストリームでの測定値とリタイア時の測定値の差によって判定できます。

UOPS\_EXECUTED は、スレッド単位ではなくコア単位で測定されるため、コアあたりの無駄な処理は次のように評価されます。

$$\text{Wasted Work (無駄な処理)} = \text{UOPS\_EXECUTED.PORT234\_CORE} + \text{UOPS\_EXECUTED.PORT015\_All\_Thread} - \text{UOPS\_RETIRED.ANY\_ALL\_THREAD}$$

上記の比率をマイクロオペレーション (μop) の平均発行比率で割るとサイクル数に変換できます。上記のイベントは、マイクロフュージョンやマクロフュージョンにおいても補正を行わず、この方法が使用できるように設計されています。

上記のイベントの最後の 2 つはスレッド単位でカウントできるため、「スレッド単位」の測定値は、発行されたマイクロオペレーション (μop) 数とリタイアしたマイクロオペレーション (μop) 数の差から求められます。これは、(実行中のサイクルを浪費する前に RS 内で排除される) 誤って予測されたマイクロオペレーション (μop) によって少し多めにカウントされますが、通常、これはわずかな収集で済みます。

Wasted Work/thread (スレッドあたりの無駄な処理) = (UOPS\_ISSUED.ANY + UOPS\_ISSUED.FUSED) - UOPS\_RETIRED.ANY

表 B-4 予測ミスにより無駄になった処理のサイクル・アカウンティング

概要	
目的	実行されたが、予測ミスのためにリタイアしなかったマイクロオペレーション (μop) を評価
方法	実行とリタイアメント間のマイクロオペレーション (μop) フローの差を調査
PMU とパイプラインの焦点	マイクロオペレーション (μop) の実行とリタイアメント
イベントコード/ Umask	マイクロオペレーション (μop) を実行する場合は、イベントコード B1H, Umask = 3FH コア単位でカウントする場合は、イベントコード C2H, Umask = 1, AllThread = 1
EvtSelc	マイクロオペレーション (μop) をカウントする場合は、CMask, Invert, Edge フィールドを 0 に設定
基本式	“Wasted work (無駄な処理)” = UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_ALL_THREAD - UOPS_RETIRED.ANY_ALL_THREAD
ドリルダウンの範囲	カウント: 分岐予測ミスのコスト
バリエーション	サイクル・アカウンティングの場合は、マイクロオペレーション (μop) の平均発行速度で除算 スレッドあたりのコストを概算する場合は、AllThread = 0 に設定

予測ミスのペナルティの 3 番目の要素である命令スタベーションは、適切なパスに関連する命令がコアから離れており、RAT 内でのマイクロオペレーション (μop) が不足して実行がストールしている場合に発生します。マイクロオペレーション (μop) が発行されない原因は、フロントエンドのスタベーションまたはバックエンドでリソースを利用できないことのいずれかです。したがって、次の方法でリソース割り当ての出力を明示的に測定できます。

- Y マイクロオペレーション (μop) がアウトオブオーダー (OOO) エンジンに発行されなかったサイクルの総数をカウントします。
- Y 割り当てるリソース (RS, ROB エントリー、ロードバッファ、ストアバッファなど) を利用できないサイクル数をカウントします。

HT が有効でない場合、命令スタベーションは単に次のように計算されます。

Instruction Starvation (命令スタベーション) =  
UOPS\_ISSUED.STALL\_CYCLES - RESOURCE\_STALLS.ANY

HT が有効な場合、RS に供給されるマイクロオペレーション (μop) は 2 つのスレッド間で切り替えられます。発行がストールしているサイクルの 50% が他方のスレッドにマイクロオペレーション (μop) を供給している可能性があるため、理想的な状況では、上記の条件では多めにカウントされることが考えられます。その場合、他方のスレッドによってマイクロオペレーション (μop) が発行されているサイクルの数を引くように式を変更します。

Instruction Starvation (スレッドごとの命令スタベーション) =  
UOPS\_ISSUED.STALL\_CYCLES - RESOURCE\_STALLS.ANY -  
UOPS\_ISSUED.ACTIVE\_CYCLES\_OTHER\_THREAD

スレッド単位の式では、命令スタベーションは若干多めにカウントされます。これは、同じコア内にある他方のスレッドがマイクロオペレーション (μop) を発行する間に、当該スレッドで RESOURCE\_STALL 条件が存在している可能性があるためです。代替式は次のようになります。

CPU\_CLK\_UNHALTED.THREAD - UOPS\_ISSUED.CORE\_CYCLES\_ACTIVE-RESOURCE\_STALLS.ANY

表 B-5 にこれらのモデルをまとめています。

表 B-5 命令スタベーションのサイクル・アカウンティング

概要	
目的	予測ミス後にマイクロオペレーション (μop) の発行を待機しているサイクルを評価
方法	マイクロオペレーション (μop) の発行とリソース割り当て間のサイクルの差を調査
PMU とパイプラインの焦点	マイクロオペレーション (μop) の発行とリソース割り当て
イベントコード/ Umask	発行されたマイクロオペレーション (μop) の場合は、イベントコード 0EH, Umask = 1 リソース割り当てがストールしているサイクルの場合は、イベントコード A2H, Umask=1
EvtSelc	マイクロオペレーション (μop) の発行がストールしているサイクルをカウントする場合は、 CMask = 1, Inv = 1 に設定 マイクロオペレーション (μop) の発行がアクティブなサイクルをカウントする場合は、 CMask = 1, Inv = 0 に設定 UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD に対する他方のスレッドの貢献を評価する場合は、2 つのカウンターをそれぞれ AllThread = 0, AllThread = 1 に設定
基本式	命令スタベーション(HT は無効) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY;
ドリルダウンの範囲	カウント: 分岐予測ミスのコスト
バリエーション	次の式により、スレッドあたりの値を評価 Instruction Starvation (命令スタベーション) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD

表 B-6 にパフォーマンス・イベントの詳細を示します。

表 B-6 マイクロオペレーション (μop) フロー用イベントの CMask/Inv/Edge/スレッドの粒度

イベント名	Umask	イベント コード	Cmask	Inv	Edge	すべての スレッド
UOPS_EXECUTED.PORT234_CORE	80H	B1H	0	0	0	1
UOPS_EXECUTED.PORT015_ALL_THREAD	40H	B1H	0	0	0	1
UOPS_RETIRED.ANY_ALL_THREAD	1H	C2H	0	0	0	1
RESOURCE_STALLS.ANY	1H	A2H	0	0	0	0
UOPS_ISSUED.ANY	1H	0EH	0	0	0	0
UOPS_ISSUED.STALL_CYCLES	1H	0EH	1	1	0	0
UOPS_ISSUED.ACTIVE_CYCLES	1H	0EH	1	0	0	0
UOPS_ISSUED.CORE_CYCLES_ACTIVE	1H	0EH	1	0	0	1

### B.4.1.2 基本ブロックのドリルダウン

INST\_RETIRED.ANY (リタイアした命令) イベントは、命令あたりのサイクル数の比率 (CPI) を評価するため一般的に使用されます。その他の重要な利用法は、基本ブロック実行回数を評価することにより、特に高いパフォーマンスが要求される基本ブロックを決定することです。

サンプリング・ツール (インテル® VTune™ Amplifier など) では、サンプルが特定の IP 値周辺に集まる傾向があります。これは、INST\_RETIRED.ANY またはサイクル・カウント・イベントを使用する際に該当します。ホットサンプルの逆アセンブルリストを表示すると、一部の命令は高いサンプルカウントに関連しますが、隣接する命令はサンプルに関連していないことがあります。

これは、基本ブロックの定義により、基本ブロック内のすべての命令が同じ回数だけリタイアするためです。ホットな基本ブロックのドリルダウンは、基本ブロック内の命令のサンプルカウントを平均化することで正確になります。



$$\text{Basic Block Execution Count (基本ブロック実行回数)} = \text{Sum (基本ブロック内の命令のサンプルカウント)} * \text{Sample\_after\_value} / (\text{基本ブロック内の命令の数})$$

ホットなループまたはホットでないループ構造に関連する基本ブロックの特定は、逆アセンブリ・リストを調査して、上記の手法に従って各ループ構造のトリップカウントを評価することで系統的に行うことができます。条件分岐のない単純なループでは、このトリップカウントは、ループブロックの直前/直後にあるブロックの基本ブロックの実行回数に対するループブロックの基本ブロックの実行回数の比率となります。複数ブロックの平均化を慎重に行うことで、精度を高めることができます。

これにより、高いトリップカウントを持つループを特定してチューニング作業に集中できます。この手法は、固定カウンタを使用することによって実装できます。

レイテンシーが長い命令 (fmul, fadd, imul) の依存関係チェーンでは、レイテンシーが長い命令の出力を使用できる一方で、ディスパッチがストールする可能性があります。divide/sqrt 実行ユニットを使用する命令を除き、一般にこのようなストールのカウンタを支援するイベントは用意されていません。

このような場合は、ARITH イベントを使用することで、これらの命令の実行回数とそれらが実行ユニットを占有した継続期間 (サイクル数) の両方をカウントできます。ARITH.CYCLES\_DIV\_BUSY イベントは、divide/sqrt 実行ユニットのいずれかが占有されたサイクルをカウントします。

## B.4.2 ストールサイクルの分解とコア・メモリー・アクセス

ストールサイクルの分解は、標準的な近似手法を用いて行いますが、これは、パフォーマンスに影響を与えるイベントごとにペナルティーが逐次的に発生することを前提としています。したがって、有効な処理に利用できるサイクルの総損失数は、イベント数  $N_i$  にイベントのタイプごとの平均ペナルティー  $P_i$  を掛けたものです。

$$\text{Counted\_Stall\_Cycles} = \text{Sum} ( N_i * P_i )$$

これは、PMU イベントによってカウントされる (またはカウントできる) パフォーマンスに影響を与えるイベントだけを考慮します。最終的には、カウントできないストールの原因は複数ありますが、それらによる影響の総量は次のように概算できます。

$$\text{Unaccounted stall cycles (不明なストールサイクルの数)} = \text{Stall\_Cycles} - \text{Counted\_Stall\_Cycles} = \text{UOPS\_EXECUTED.CORE\_STALLS\_CYCLES} - \text{Sum} ( N_i * P_i )_{\text{both\_threads}}$$

このシーケンシャル・ペナルティー・モデルは、非常に単純であり、通常は個々のマイクロアーキテクチャーの問題による影響を多めにカウントするため、不明な構成要素は負数になる場合があります。

B.4.1.1 節で述べたように、UOPS\_EXECUTED.CORE\_STALL\_CYCLES は、スレッド単位ではなくコア単位でカウントされるため、多めにカウントされる可能性があります。そのような場合、スレッド単位でカウント可能なポート 0、1、5 のマイクロオペレーション (μop) ストールを使用することが望ましいことがあります。

$$\text{Unaccounted stall cycles (スレッドごとの不明なストールサイクルの数)} = \text{UOPS\_EXECUTED.PORT015\_THREADED\_STALLS\_CYCLES} - \text{Sum} ( N_i * P_i )$$

この不明な要素は、パフォーマンス・イベントの不足によってカウントされなかったか、単にデータ収集時に無視された要素を表すために使用されます。

ストールの基準として、「リタイアメント」ポイントを適用することもできます。PEBS イベントである UOPS\_RETIRED.STALL\_CYCLES には、スレッド単位で評価でき、リタイアするマイクロオペレーション (μop) に関連する IP をハードウェアで取得できるという利点があります。つまり、OS カーネルのクリティカル・セクションにおいて、IP の分布が割り込みの STI/CLI 遅延による影響を受けないため、OS で処理されるプロファイルをより正確に取得できます。

## B.4.2.1 マイクロアーキテクチャー条件のコストの測定

ここで説明する方法でストールサイクルの分類を行う場合、まずパフォーマンス・ペナルティーが大きい条件（例えば、ペナルティーが 10 サイクルよりも大きいイベントなど）に注目します。アウトオブオーダー（OOO）実行とコンパイラーの複合作用により、ペナルティーが小さいイベント（ $P < 5$  サイクル）は隠蔽されることがあります。

アウトオブオーダー（OOO）エンジンは、命令ストリーム内の両方のタイプのストールを管理しており、命令の依存関係によっていずれかのタイプのストールが発生した場合、実行ユニットをビジー状態に保とうとします。通常、大きなペナルティーの操作はメモリーアクセスに関連し、レイテンシーが長い `divide/sqrt` 命令などの影響を受けます。

ペナルティーが最も大きいイベントは、キャッシュ階層の L1 または L2 内に存在しないキャッシュラインを必要とするロード操作に関連します。したがって、発生回数だけでなく、どのようなペナルティーであるかを知る必要があります。

レイテンシーの標準的な測定手法は、キューで要求されるサイクル数の平均を測定するものです。

$$\text{Latency (レイテンシー)} = \text{Sum (CYCLES\_Queue\_entries\_outstanding)} / \text{Queue\_inserts}$$

ここで、「Queue\_inserts」は、そのキュー内の未処理のサイクルを引き起こしたエントリーの総数を表します。ただし、キューへの挿入に関連するペナルティー（キャッシュミス）は、レイテンシーをキューの平均占有率で割ったものです。この補正は、オーバーラップしているペナルティーに関連する過剰なカウントを回避するために必要です。

$$\text{Avg\_Queue\_Depth (キューの深さの平均)} = \text{Sum (CYCLES\_Queue\_entries\_outstanding)} / \text{Cycles\_Queue\_not\_empty}$$

事象ごとのペナルティー（コスト）は次のようになります。

$$\text{Penalty (ペナルティー)} = \text{Latency} / \text{Avg\_Queue\_Depth} = \text{Cycles\_Queue\_not\_empty} / \text{Queue\_inserts}$$

別の考え方をすると、キューを占有するイベントのすべてのペナルティーの合計は、キューが空でない時間を超えることはないことを認識することです。

$$\text{Cycles\_Queue\_not\_empty (キューが空ではないサイクル)} = \text{Events} * \langle \text{Penalty} \rangle$$

前述の標準的な手法は、概念的には単純です。しかし、実際には、ワークロード内での大量のメモリー参照や、さまざまな状態/ロケーション固有のレイテンシーにより、標準的なサンプリング手法は実用的ではありません。インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサ上では、イベントごとのプリサイズ・サンプリング（PEBS）の利用が推奨されます。

サンプリングによるペナルティーのプロファイリング（IP の測定を局所化するために行われる）では、精度が問題となることがあります。L2 ミスのレイテンシーは 40 ~ 400 サイクルとさまざまであるため、必要とされる数のサンプル収集は介入的になる傾向があります。

後述するプリサイズ・レイテンシー・イベントは、サンプリングを行う際の正確で柔軟性のある測定手法です。各サンプルでは、待機時間とデータソースの両方が記録されるため、データソースごとの平均レイテンシーを評価できます。また、PEBS ハードウェアは、バッファ一杯になるまで PMI を生成しないイベントのバッファリングをサポートしているため、ワークロードに介入することなくこのような評価を効率良く行うことができます。

また、コア内で発生し、さまざまな条件、局所性、またはキャッシュ整合性要件によりトラフィックの遅延が生じたメモリーアクセスのコストを測定するため、コア PMU の多数のパフォーマンス・イベントを使用できます。メモリーアクセスのレイテンシーは、L3 の局所性、ローカル・メモリー・コントローラーまたはリモート・コントローラーに装着された DRAM、キャッシュ整合性によって異なります。表 B-7 に、レイテンシーの概算値を示します。

表 B-7 インテル® Xeon® プロセッサ-5500 の概算レイテンシー

データソース	レイテンシー
L3 ヒット、ライン排他	42 サイクル
L3 ヒット、ライン共有	63 サイクル
L3 ヒット、別のコアで変更	73 サイクル
リモート L3	100 - 150 サイクル
ローカル DRAM	50 ナノ秒
リモート DRAM	90 ナノ秒

### B.4.3 コア PMU のプリサイスイベント

イベントごとのプリサイス・サンプリング (PEBS) により、PMU はイベントを発生させた命令の完了時にアーキテクチャー・ステートと IP を取得できます。これにより、プロファイリングとチューニングにおいて 2 つの利点が得られます。

- Y 命令空間内のイベント条件の位置を正確にプロファイリングできます。
- Y 取得したレジスター状態の PEBS レコードを使用して、後処理フェーズで命令の引数を再構成できます。

インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサの PEBS 機能は大幅に拡張されており、より多くのそして豊富なプリサイスイベントを使用できます。

この機構は、カウンターのオーバーフローを使用して PEBS データを取得しているため、後続のイベントでデータが取得され、割り込みが発生します。

命令が完了したときの IP 値は次の命令を指すため、取得後の IP 値は「IP +1」と表されることがあります。

プリサイスイベントは、その特性上「リタイア時」にイベントが生成されます。ここでは、プリサイスイベントを、ロードとストアのリタイアメントに関連するメモリー・アクセス・イベントと、すべての命令または特定の非メモリー命令 (分岐、FP アシスト、インテル® SSE マイクロオペレーション (μop)) のリタイアメントに関連する実行イベントに分けて説明します。

#### B.4.3.1 プリサイス・メモリー・アクセス・イベント

すべてのプリサイス・メモリー・アクセス・イベントには、共通する 2 つの重要な性質があります。

- Y イベントを生成した命令の IP がハードウェアによって取得されるため、正確に命令を特定できます。取得される IP は次の命令を示しますが、サンプルを単純に 1 つ上の命令に移動します。記録された IP が基本ブロックの最初の命令を指していてもこの方法は機能します。分岐命令はデータをロード/ストアすることがないため、イベントを生成した命令は直前の基本ブロックの最後の命令であることがあります。そのため、取得されたレジスター状態の PEBS のレコードを使用して、後処理フェーズで命令の引数を再構成できます。
- Y PEBS バッファーには、16 個の汎用レジスター R1 ~ R16 の値がすべて含まれます (R1 は RAX と呼ばれます)。ロードまたはストアのアドレスは、逆アセンブルと組み合わせることで再構成できるため、データ・アクセスのプロファイリングに使用できます。インテル® パフォーマンス・チューニング・ユーティリティーは、これを正確に実行し、各種の強力な分析手法を提供します。

プリサイス・メモリー・アクセス・イベントは、通常、非常に長期にわたる実行ストールの原因となるロードイベントに注目します。このイベントはデータソースごとに分類され、典型的なレイテンシーおよび NUMA 構成におけるデータの局所性を示します。プリサイス・ロード・イベントは、L2、L3、DRAM に対するロード操作のアクセスイベントのみをカウントします。その他すべてのイベントには、L1D/L2 ハードウェア・プリフェッチ要求が含まれます。多くのイベントは RFO 要求も含まれますが、これはストアおよびハードウェア・プリフェッチによるものです。

4 つの汎用カウンターはすべて、プリサイス・イベント・データを収集するようにプログラムできます。ロード命令とストア命令の仮想アドレス再構成機能により、キャッシュラインとページの利用効率を分析できます。キャッシュラインとページは物理アドレスによって定義されますが、下位ビットが同じであるため、仮想アドレスを使用できます。

PEBS が命令の完了時にレジスター値を取得する場合、ポインター追跡型のロード操作が取得されないことに注意してください。これは、ロード命令を逆参照アドレスから推測できないためです。

基本的な PEBS メモリー・アクセス・イベントは、次のカテゴリーに分類されます。

- Y MEM\_INST\_RETIRED: このカテゴリーは、リタイアした命令のうち、ロード操作を含む命令をカウントします。イベントコード 0BH で選択します。
- Y MEM\_LOAD\_RETIRED: このカテゴリーは、リタイアしたロード命令のうち、Umask 値によって選択された特定の条件が発生した命令をカウントします。イベントコードは 0CBH です。
- Y MEM\_UNCORE\_RETIRED: このカテゴリーは、リタイアし、アンコア・サブシステムからデータを取得したメモリー命令をカウントします。イベントコードは 0FH です。
- Y MEM\_STORE\_RETIRED: このカテゴリーは、リタイアした命令のうち、ストア操作を含む命令をカウントします。イベントコードは 0CH です。
- Y ITLE\_MISS\_RETIRED: このカテゴリーは、リタイアした命令のうち、ITLB ミスが発生した命令をカウントします。イベントコードは 0C8H です。

上記の PEBS メモリーイベントの Umask 値および関連する名前サフィックスは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」に示されています。

上に示したプリサイスイベントにより、ロードによるキャッシュミスはデータソースごとに特定できます。ただし、NUMA 構成に関するキャッシュラインの "ホーム" ロケーションは特定されません。イベント MEM\_UNCORE\_RETIRED.LOCAL\_DRAM および MEM\_UNCORE\_RETIRED.NON\_LOCAL\_DRAM は例外で、これらをインストルメント済みの malloc 呼び出しと組み合わせると、アプリケーションで使用される重要な隣接するバッファの NUMA "ホーム" を特定できます。

すべての MEM\_LOAD\_RETIRED イベントの和は、MEM\_INST\_RETIRED.LOADS カウントと等しくなります。

L1D ミスのカウントには、MEM\_LOAD\_RETIRED.L1D\_HIT を除くすべての MEM\_LOAD\_RETIRED イベントを使用します。その場合、MEM\_INST\_RETIRED.LOADS - MEM\_LOAD\_RETIRED.L1D\_HIT の差ではなく、個々の MEM\_LOAD\_RETIRED イベントをすべて使用の方が適しています。これは、プリサイスイベントの総数は、対象のイベントを発生させた命令を正しく特定できますが、この節で後述する PEBS のシャドウ化により、イベントの分布が適切でないことがあるためです。

L1D\_MISSES =  
 MEM\_LOAD\_RETIRED.HIT\_LFB + MEM\_LOAD\_RETIRED.L2\_HIT +  
 MEM\_LOAD\_RETIRED.L3\_UNSHARED\_HIT + MEM\_LOAD\_RETIRED.OTHER\_CORE\_HIT\_HITM +  
 MEM\_LOAD\_RETIRED.L3\_MISS

MEM\_LOAD\_RETIRED.L3\_UNSHARED\_HIT イベントメトリックには若干説明が必要です。インクルーシブな L3 には、ラインのコピーを保持するコアを特定するビットパターンが備えられています。1 つのビットのみが要求元のコア (非共有状態のヒット) に対して設定されている場合、ほかのコアをスヌープせずに L3 からラインが返されます。複数のビットが設定されている場合、ラインは共有状態であり、L3 内のコピーは最新であるため、この場合もほかのコアをスヌープせずに L3 からラインが返されます。

ラインが別のコアによる所有権のための読み出し (RFO) 状態である場合は、L3 内のコピーは排他状態です。その後、そのコアによってラインが変更され排出されると、L3 内のライトバックされたコピーは変更状態になり、スヌープの必要はなくなります。MEM\_LOAD\_RETIRED.L3\_UNSHARED\_HIT はこれらをすべてカウントします。このイベントは、MEM\_LOAD\_RETIRED.L3\_HIT\_NO\_SNOOP と呼ばれるべきです。

同じ理由で、イベント MEM\_LOAD\_RETIRED.L3\_HIT\_OTHER\_CORE\_HIT\_HITM は、MEM\_LOAD\_RETIRED.L3\_HIT\_SNOOP と命名されるべきでしょう。

変更されたラインもまた、別のソケットから取得されると、メモリーにライトバックされます。その結果、リモート HITM アクセスは、ホーム DRAM からのアクセスのように見えます。したがって、MEM\_UNCORE\_RETIRED.LOCAL\_DRAM および MEM\_UNCORE\_RETIRED.REMOTE\_DRAM イベントも、リモートプロセッサのキャッシュ内の変更されたラインによる L3 ミスをカウントします。

MEM\_LOAD\_RETIRED.DTLB\_MISSES の動作は、インテル® Core™2 プロセッサとは異なります。以前は、非ブライサスイベントと同様、このイベントはページに対する最初のミスだけをカウントしていました。このイベントは、ミスを発生させたすべてのロードをカウントするように変更されています (2 番目以降のミスも含む)。

### B.4.3.2 ロード・レイテンシー・イベント

インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのインテル® プロセッサは、イベントコード 0BH と Umask 値 10H (LATENCY\_ABOVE\_THRESHOLD) を使用して、「ロード・レイテンシー・イベント」である MEM\_INST\_RETIRED をサポートします。このイベントは、ロードをサンプリングして、命令の実行と実際にデータが供給されるまでのサイクルの数を記録します。測定されたレイテンシーが、MSR 0x3F6 (ビット 15:0) にプログラムされた最小レイテンシーよりも大きい場合、カウンターはカウントアップされます。

カウンターのオーバーフローによって PEBS 機構を実現しています。PMU は、レイテンシーのしきい値を満たす次のイベントで、測定されたレイテンシー、仮想アドレスまたはリニアアドレス、データソースを PEBS レコード・フォーマットで PEBS バッファーに書き込みます。仮想アドレスは既知のロケーションに取得されるため、サンプリング・ドライバーも仮想アドレスから物理アドレスへの変換を行い物理アドレスを取得する可能性があります。物理アドレスによって NUMA のホーム・ロケーションが特定され、原則としてキャッシュ占有の詳細な分析が可能になります。

また、リタイアメントの前にアドレスが取得されるため、“MOV RAX, [RAX+const]” のエンコードを追跡するポインタアドレスも取得されます。レイテンシーのしきい値を指定するには MSR\_PEBS\_LD\_LAT\_THRESHOLD MSR が必要になるため、一定の期間中に 1 つの最小レイテンシー値だけをコア上でサンプリングできます。これを可能にするため、インテル® パフォーマンス・ツールは、このイベント利用を 4 つのカウンターに制限することによってスケジューリングを簡素化しています。表 B-8 に、インテル® Performance Tuning Utility (インテル® PTU) およびインテル® VTune™ Amplifier がロード・レイテンシー・イベント向けに使用するイベント・プログラミング構成の例を示します。

さまざまな最小レイテンシーのしきい値が MSR\_PEBS\_LD\_LAT\_THRESHOLD (アドレス 0x3F6) に指定されます。

表 B-8 ロード・レイテンシー・イベント・プログラミング

ロード・レイテンシーのプリサスイベント	MSR 0x3F6	Umask	イベントコード
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_4	4	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8	8	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_10	16	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_20	32	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_40	64	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_80	128	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_100	256	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_200	512	10H	0BH
_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8000	1024	10H	0BH

ロード・レイテンシー・イベントの PEBS アシスト機構によって各 PEBS レコードに書き込まれる 3 つのフィールドのうちの 1 つがデータソースの局所性情報をエンコードします。

表 B-9 ロード・レイテンシー PEBS レコードのデータソース・エンコード

エンコード	説明
0x0	未知の L3 キャッシュミス。
0x1	最小レイテンシーのコアキャッシュをヒット。この要求は L1 データキャッシュが対応。
0x2	保留中のコアキャッシュをヒット。同じキャッシュライン・アドレスに対して未処理のコア・キャッシュミスがすでに進行しています。データはまだデータキャッシュ内に存在せず、まもなくキャッシュにコミットされるフィルバッファがあります。
0x3	この要求は L2 が対応。
0x4	L3 ヒット。コヒーレンシー操作を必要としないアンコア内の L3 キャッシュにヒットしたローカルまたはリモートホーム要求 (スヌープ)。
0x5	L3 ヒット (ほかのコア・ヒット・スヌープ)。L3 キャッシュにヒットし、変更されたコピーが見つからなかったコア間のスヌープを持つ別のプロセッサ・コアによって処理された、ローカルまたはリモートホーム要求 (クリーン)。
0x6	L3 ヒット (ほかのコア HITM)。L3 キャッシュにヒットし、変更されたコピーが見つかったコア間のスヌープを持つ別のプロセッサ・コアによって処理された、ローカルまたはリモートホーム要求 (HITM)。
0x7	予約済み
0x8	L3 ミス (リモート・キャッシュ・フォワードイング)。L3 キャッシュをミスし、変更されたコピーが見つからなかったパッケージ間のスヌープに続く転送済みデータによって処理された、ローカルホーム要求 (リモートホーム要求はカウントされません)。
0x9	予約済み。
0xA	L3 ミス (ローカル DRAM が共有状態 (S) に遷移) L3 キャッシュをミスし、ローカル DRAM によって処理されたローカルホーム要求。
0xB	L3 ミス (リモート DRAM が共有状態 (S) に遷移) L3 キャッシュをミスし、リモート DRAM によって処理されたリモートホーム要求。
0xC	L3 ミス (ローカル DRAM が排他状態 (E) に遷移) L3 キャッシュをミスし、ローカル DRAM によって処理されたローカルホーム要求。
0xD	L3 ミス (リモート DRAM が排他状態 (E) に遷移) L3 キャッシュをミスし、リモート DRAM によって処理されたリモートホーム要求。
0xE	I/O、入出力操作の要求。
0xF	キャッシュ不可能メモリーに対する要求。

レイテンシー・イベントは、サイクル・アカウンティング分解のパナルティーを測定するための推奨される手法です。この PEBS イベントによって PMI が発生するたびに、レイテンシーに関連するロードとキャッシュラインのデータソースが PEBS バッファに記録されます。このキャッシュラインのデータソースは、データソース・フィールドの下位 4 ビットおよび上記の表から導かれます。収集されたデータから、16 個のソースのそれぞれに対する平均レイテンシーを評価できます。最小レイテンシーは一度に 1 つずつしか収集できないため、MLC ヒットとリモートソケット DRAM のレイテンシーを評価しにくい場合があります。ただし、すべてのオフコアソースに対し適切に分散するには、32 サイクルの最小レイテンシーが科せられます。インテル® PTU バージョン 3.2 を使用すると、データ・プロファイリング・モードでレイテンシーの分布を表示し、そのイベントに対する高度なイベントフィルター処理を適用できます。

### B.4.3.3 プリサイズ実行イベント

コア PMU 内の PEBS 機能は、ロード命令およびストア命令を超えるものであり、選択されたタイプのリタイアした分岐および予測が外れた (その後リタイアした) 分岐に関する、分岐、near コール、条件分岐をすべてプリサイズイベントによってカウントできます。これらのイベント対し、PEBS バッファは分岐のターゲットを含みます。最後の分岐レコード (LBR) も取得されている場合は、分岐命令のロケーションも判断できます。

分岐が処理されると、PEBS バッファ内の IP 値も LBR の最後のターゲットとして認識されます。条件分岐がジャンプしなかった場合、分岐せずにリタイアした分岐は PEBS バッファ内の IP の 1 つ前の命令となります。

near コールがリタイアした場合、イベント・ベース・サンプリング (EBS) により正確な関数呼び出しの回数を収集できます。これは、関数をインライン展開するかどうか判断する測定基準となります。呼び出し回数を測定するには、呼び出しをサンプリングする必要があり、その他のトリガーは、適切に修正されることが保証されないバイアスの原因となります。

プリサイス分岐イベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード C4H に示されます。

プリサイスイベントに関連したサンプリング結果には誤差が生じる可能性があります。これは、PMU カウンターのオーバーフローと PEBS ハードウェアを補正する間の時間的な遅延によるものです。タイミングシャドウにより、この期間中はイベントを検出できません。この影響を理解するため、関数呼び出しチェーンについて考えてみます。実行時間が長い関数「foo」は、実行時間が非常に短い 3 つの関数「foo1」、「foo2」、「foo3」を呼び出します。「foo1」は「foo2」を呼び出し、「foo2」は「foo3」を呼び出し、その後実行時間が長い関数「foo4」が続きます。foo1、foo2、foo3 の実行時間がシャドウ期間よりも短い場合、PEBS でサンプリングされた呼び出し分布は大きくゆがんだものになります。次に例を示します。

- Y foo の呼び出しでオーバーフローが発生すると foo1 の呼び出しが実行されるまでに PEBS 機構が実施され、foo から foo1 の呼び出しを示すサンプルが収集されます。
- Y しかし、foo1、foo2、または foo3 の呼び出しによってオーバーフローが発生すると、foo4 の本体で呼び出しが実行されるまで PEBS 機構は実施されません。そのため、foo2、foo3、foo4 の呼び出しは、PEBS でサンプリングされた呼び出しとして示されません。

シャドウは、すべての PEBS イベントの分布に影響します。また、リタイアした分岐イベント(PEBS または非 PEBS) と LBR 内の最後のエンタリーを組み合わせると特定される基本ブロックの実行回数の分布にも影響されます。PMU カウンターのオーバーフローと LBR の確定の間に遅延がなければ、最後の LBR エンタリーから分岐したリタイア済みの分岐をサンプリングし、そこから基本ブロック実行回数を特定できます。最後に分岐した分岐と前回のターゲット間のすべての命令は一度だけ実行されます。

このようなサンプリングを使用することで、均一なサンプリングによってリタイアした「ソフトウェア」命令イベントを生成し、そのイベントを基に基本ブロックの実行回数を特定できます。残念ながら、シャドウでは短い基本ブロックの末端にある分岐は LBR の最後のエンタリーにはならないため、測定結果が片寄る傾向があります。これは、基本ブロック内のすべての命令は同じ回数だけ実行されるように構成されているためです。

呼び出し回数と基本ブロックの実行回数に対するシャドウの影響は、LBR 内のエンタリーを平均化することによって大幅に軽減できる可能性があります。これについては、LBR に関する節で説明します。

一般に、ワークロード内のすべての命令に分岐が占める割合は 10% を超えるため、ループの最適化では、高いトリップカウントを持つループに注目する必要があります。ループカウントに関しては、終了条件の判定において誘導変数をトリップカウントと比較するのが一般的です。これは、たとえ高度な最適化が適用されても、ループの本体内で誘導変数が使用されている場合に該当します。したがって、アンロール操作のループシーケンスは次のようになります。

```
add rcx, 8
cmp rcx, rax
jnge triad+0x27
```

この場合、2 つのレジスター rax および rcx が、トリップカウントとインダクション変数です。リタイアした条件分岐イベント向けの PEBS バッファを取得すると、比較命令の 2 つのレジスターの平均値を評価できます。平均値が大きい方がトリップカウントです。これにより、平均値、RMS、最小値、最大値、さらに記録された値の分布まで評価できます。

#### B.4.3.4 最後の分岐レコード (LBR)

LBR は、分岐処理されてリタイアした各分岐のソースとターゲットを取得します。インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサは、ローテート・バッファ内の 16 組のソース/ターゲットアドレスを追跡できます。タイプ別および特権レベル別の分岐命令のフィルター処理は、専用機能である MSR\_LBR\_SELECT を使用することによって可能となります。つまり、LBR 機構は、リング 0 またはリング 3 特権レベル、あるいはその両方 (デフォルト) で発生する分岐を取得するようにプログラムできます。さらに、記録された分

岐処理済み分岐をフィルター処理することもできます。MSR\_LBR\_SELECT を使用して指定できるフィルター処理オプションのリストについては、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 17 章「Debugging, Branch Profiles and Time-Stamp Counter」で説明されています。

デフォルトでは、すべての特権レベル (すべてのビットが 0) ですべての分岐が取得されます。また、ビット 1 (リング 3 を取得)、ビット 3 (near コールを取得)、ビット 6、ビット 7 を除くすべてのビットを 1 に設定することで、リング 3 コールと無条件ジャンプだけが LBR 内に残されます。このようにプログラミングすることで、LBR は最後の 16 個の処理済みの呼び出しとリタイアした無条件ジャンプ、およびそれらのターゲットをバッファー内に保持できます。

その後、PMU サンプリング・ドライバーは、この限定された「呼び出しチェーン」を任意のイベントとともに取得することによって「コールツリー」コンテキストを提供可能になります。残念ながら、無条件ジャンプを含むと問題が生じます。これは、特にループ内に if-else 構文が存在する場合に当てはまります。

すべてのレベルで頻繁に呼び出される関数では、コンテキストを明確にするためリターンを含めて追加します。ただし、取得可能な呼び出しチェーンの入れ子数が減少します。明らかな使用法は、レイテンシーが極めて長いロードのサンプリングをトリガーすることです。これにより、頻繁に競合するロックされた変数にアクセスするサンプルを改善し、呼び出しチェーンを取得してロックを使用するコンテキストを特定します。

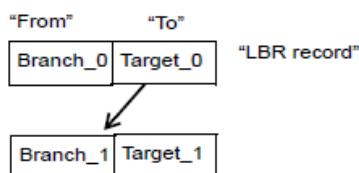
### 呼び出し回数と関数の引数

BR\_INST\_RETIRED.NEAR\_CALL イベントによってトリガーされた PMI の LBR が取得された場合、LBR 内の最後のエントリーを参照するだけで呼び出し元の関数ごとに呼び出しカウントを判定できます。PEBS IP は LBR 内の最後のターゲット IP と等しいため、呼び出し元の関数のエントリーポイントを示します。同様に、LBR バッファー内の最後のソースは、呼び出し元の関数内からのコールサイトとなります。完全な PEBS レコードを取得した場合、64 ビット OS では引数の数が制限されている関数の呼び出し回数と関数の引数の両方をサンプリングできます。

### LBR と基本ブロックの実行回数

もう 1 つの興味深い用法は、フィルターを用いずに BR\_INST\_RETIRED.ALL\_BRANCHES イベントと LBR を使用して基本ブロックの実行速度を評価することです。LBR は分岐したすべての分岐を取得するため、分岐 IP (ソース) と LBR バッファー内の直前のターゲット間のすべての基本ブロックが 1 度実行されます。特定のロードモジュールの基本ブロックの実行回数を簡単に評価する方法は、すべての基本ブロックの開始位置のマップを作成することです。その後、PEBS アドレス (ターゲットであるが、おそらく分岐処理された分岐のアドレスでないために LBR バッファー内に存在する必要がない) で開始され、目的のロードモジュールと一致しないアドレスが見つかるまで LBR を逆方向に探索する BR\_INST\_RETIRED.ALL\_BRANCHES の PEBS の集合によってトリガーされた各サンプルについて、実行された基本ブロックをすべてカウントします。この値は「number\_of\_basic\_blocks」と呼ばれ、すべてのブロックの実行回数が  $1/(\text{number\_of\_basic\_blocks})$  ずつカウントアップされます。この方法により、アクティブな分岐の分岐率と未分岐率を取得できます。

LBR にリストされる分岐は、ソース IP と (同じモジュール内の) 前回のターゲット IP の間にあるすべての分岐命令が分岐するわけではありません。これを次の図に示します。



イベントカウントごとに、Target\_0 と Branch\_1 の間にあるすべての命令が 1 度リタイアします。  
 イベントカウントごとに、Target\_0 と Branch\_1 の間にあるすべての基本ブロックが 1 度実行されます。  
 Target\_0 と Branch\_1 の間にあるすべての分岐命令は分岐しません。

図 B-6 LBR レコードと基本ブロック



この 16 組の LBR レコードは、サンプリング・プロセスで特定の命令が偏って集計された PEBS サンプルの誤差の修正に役立ちます。図 B-7 に、PEBS サンプルの分布が偏っている状態を示します。

正常な実行のフローにおける基本ブロックの数について考えてみます。基本ブロックには、実行に 20 サイクル要するものもあれば、2 サイクルのものもあります。シャドウは 10 サイクルを要します。オーバーフロー条件が発生するたびに、補正された PEBS の遅延が少なくとも 10 サイクル発生します。PEBS が補正されると、次のイベント生成条件で PEBS レコードが取得されます。PEBS レコードを使用してサンプリングされた命令アドレスの分布は、図 B-7 の中央に示すように偏っています。この概念上の例では、すべての分岐がこれらの基本ブロック内で分岐されることを前提としています。

PEBS サンプルが偏って分布している状態では、最後の基本ブロックの分岐 IP は、最もサンプリング数の少ない分岐 IP アドレス (2 番目の基本ブロック) の 5 倍記録されていることが分かります。

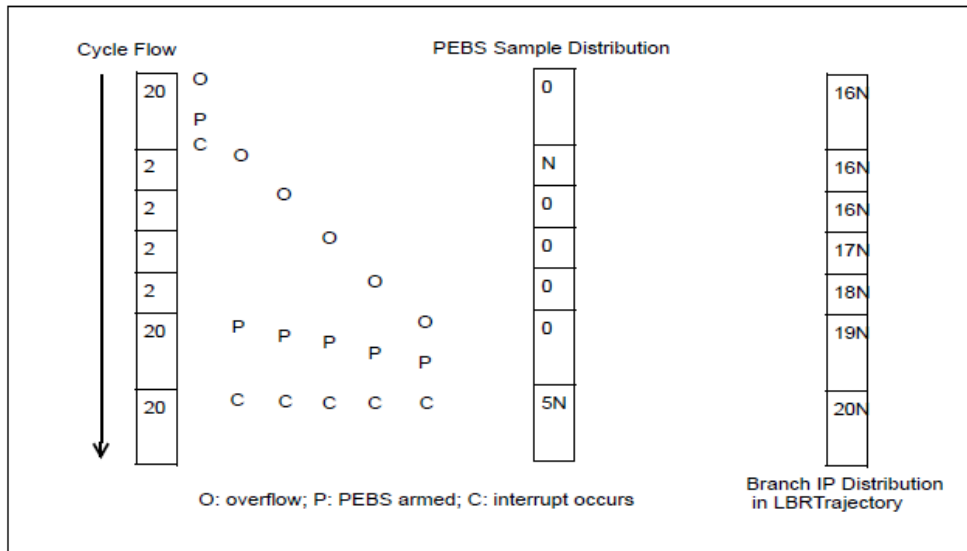


図 B-7 LBR レコードによる偏った PEBS サンプルの分布の修正

この状況では、基本ブロックによってサンプルが取得されないように見えたり、多くのサンプルが何度も取得されたりします。この例では、各エンタリーに  $1/(\text{LBR の軌道内の基本ブロックの数})$  を上積みすることは、一番右のテーブル内の数字を 16 で割ることと同等です。そのため、PEBS サンプルを直接発生させないものを含め、すべての基本ブロックではるかに正確な実行回数  $((1.25 \rightarrow 1.0) * N)$  が得られます。

これは、インテル® Core™2 プロセッサには、さまざまな方法で利用できるプリサイズ命令リタイアイベントが存在するためです。また、uops\_retired、多様なインテル® SSE 命令クラス、FP アシストなどのプリサイズイベントもあります。FP アシストイベントは、インテル® SSE FP 命令ではなく、x87 FP アシストだけを検出することに注意してください。すべてのアシストの検出については、パイプライン・フロントエンドの節で説明します。

命令リタイアイベントには、特別な使用方法があります。分布は均一ではありませんが、それぞれの合計は正確です。基本ブロック内のすべての命令で記録された値が平均化されていれば、一定の基本ブロックの実行回数を抽出できます。上記のカウント済みループ手法を適用できない場合、基本ブロックの実行比率を用いてループのトリップカウントを算出します。

また、PEBS バージョン (汎用カウンタ) の命令リタイアイベントを使用することにより、たとえ STI/CLI セマンティクスであっても、OS の実行を正確にプロファイルできます。これは、クリティカル・セクションの完了後に PEBS 割り込みが発生しますが、データはすでに適切に収集されているためです。cmask 値が非常に大きい値に設定され、invert 条件が適用されている場合、結果はすべて真であり、イベントはコアサイクル (halted および unhalted) をカウントします。

そのため、サイクル数とリタイアした命令数の両方を正確にプロファイルできます。UOPS\_RETIRED.ANY イベントも正確であり、これを使用してリング 0 の実行をプロファイルすることで実行精度を改善できます。この目的に利用

できるプリサイスイベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード C0H、C2H、C7H、F7H に示されています。

### B.4.3.5 コア・メモリー・アクセス・レイテンシーの測定

局所性に関連したパフォーマンスの問題やキャッシュ・コヒーレンシーの問題をドリルダウンするには、パフォーマンス監視イベントを使用する必要があります。各プロセッサ・コアには、L2 ミスによるメモリー・アクセス・トラフィックの要求をバッファーに格納するエントリーをアンコア・サブシステムに割り当てるスーパーキューが備わっています。表 B-10 に、L2 ミスに関連したパフォーマンスの問題をドリルダウンできる、コア PMU 内で利用可能な各種パフォーマンス・イベントを示します。

表 B-10 L2 ミスをドリルダウンするためのコア PMU イベント

コア PMU イベント	Umask	イベントコード
OFFCORE_REQUESTS.DEMAND.READ_DATA1	01H	B0H
OFFCORE_REQUESTS.DEMAND.READ_CODE1	02H	B0H
OFFCORE_REQUESTS.DEMAND.RFO1	04H	B0H
OFFCORE_REQUESTS.ANY.READ	08H	B0H
OFFCORE_REQUESTS.ANY.RFO	10H	B0H
OFFCORE_REQUESTS.UNCACHED_MEM	20H	B0H
OFFCORE_REQUESTS.L1D.WRITEBACK	40H	B0H
OFFCORE_REQUESTS.ANY	80H	B0H

**注:**

1. \*DEMAND\* イベントは、L1D キャッシュ・ハードウェア・プリフェッチによって行われたすべての要求を含みません。

表 B-11 に、スーパーキュー操作に関連したパフォーマンスの問題をドリルダウンできる、コア PMU 内で利用可能な各種パフォーマンス・イベントを示します。

表 B-11 スーパーキュー操作のためのコア PMU イベント

コア PMU イベント	Umask	イベントコード
OFFCORE_REQUESTS_BUFFER_FULL	01H	B2H

また、L2 ミスは、データ移動元の属性と応答属性によってさらにドリルダウンできます。データ移動元の属性と応答属性を指定するマトリクスは、アドレス 1A6H で専用の MSROFFCORE\_RSP\_0 を使用します。表 B-12 および表 B-13 を参照してください。

表 B-12 オフコア応答をドリルダウンするコア PMU イベント

コア PMU イベント	OFFCORE_RSP_0 MSR	Umask	イベントコード
OFFCORE_RESPONSEL	表 B-13 を参照してください。	01H	B7H

表 B-13 OFFCORE\_RSP\_0 MSR のプログラミング

要求タイプ	位置	説明	注記
	0	要求データ Rd = DCU 読み出し (パーシャル、DCU プリフェッチを含む)	
	1	要求 RFO = DCU RFO	
	2	要求 Ifetch = IFU フェッチ	
	3	ライトバック = L2_EVICT/DCUWB	
	4	PF データ Rd = L2 プリフェッチ読み出し	
	5	PF RFO = L2 プリフェッチ RFO	
	6	PF Ifetch = L2 プリフェッチ命令フェッチ	
	7	その他	非テンポラルなストアを含む
	8	L3_HIT_UNCORE_HIT	排他ライン
	9	L3_HIT_OTHER_CORE_HIT_SNP	クリーンなライン
	10	L3_HIT_OTHER_CORE_HITM	変更されたライン
	11	L3_MISS_REMOTE_HIT_SCRUB	複数のコアが使用
	12	L3_MISS_REMOTE_FWD	1 つのコアが使用するクリーンなライン
	13	L3_MISS_REMOTE_DRAM	
	14	L3_MISS_LOCAL_DRAM	
	15	Non-DRAM	非 DRAM 要求

表 B-13 によれば、理論上は MSR\_OFFCORE\_RSP\_0 の設定の組み合わせは  $2^{16}$  個許容されます。しかし、8 ビット値のサブセットを組み合わせて「要求タイプ」と「応答タイプ」を指定する方がより効果的です。表 B-14 に、より一般的な 8 ビットマスク値を示します。

表 B-14 OFFCORE\_RSP\_0 MSR の一般的な要求タイプと応答タイプ

要求タイプ	マスク	応答タイプ	マスク
ANY_DATA	xx11H	ANY_CACHE_DRAM	7FxxH
ANY_IFETCH	xx44H	ANY_DRAM	60xxH
ANY_REQUEST	xxFFH	ANY_L3_MISS	F8xxH
ANY_RFO	xx22H	ANY_LOCATION	FFxxH
CORE_WB	xx08H	IO	80xxH
DATA_IFETCH	xx77H	L3_HIT_NO_OTHER_CORE	01xxH
DATA_IN	xx33H	L3_OTHER_CORE_HIT	02xxH
DEMAND_DATA	xx03H	L3_OTHER_CORE_HITM	04xxH
DEMAND_DATA_RD	xx01H	LOCAL_CACHE	07xxH
DEMAND_IFETCH	xx04H	LOCAL_CACHE_DRAM	47xxH
DEMAND_RFO	xx02H	LOCAL_DRAM	40xxH
OTHER1	xx80H	REMOTE_CACHE	18xxH
PF_DATA	xx30H	REMOTE_CACHE_DRAM	38xxH
PF_DATA_RD	xx10H	REMOTE_CACHE_HIT	10xxH
PF_IFETCH	xx40H	REMOTE_CACHE_HITM	08xxH
PF_RFO	xx20H	REMOTE-DRAM	20xxH
PREFETCH	xx70H		

注:

- MSR\_OFFCORE\_RSP\_0 を値 4080H に設定すると、PMU が間違ったカウントを報告する場合があります。ローカル DRAM に対する非テンポラルなストアはカウントされません。

### B.4.3.6 コアごとの帯域幅を測定

個々のコアに対するすべてのメモリー・トラフィック帯域幅を測定するのは複雑ですが、コア PMU やアンコア PMU は、コアあたり帯域幅の重要な要素を測定する機能を備えています。

マイクロアーキテクチャー・レベルでは、(非テンポラルな書き込みにある程度類似した) L2 からライトバック/排出のため L3 にはバッファリングがあります。変更されたラインを L2 から排出すると、ラインは L3 へライトバックされます。L3 内のラインは、(必要であるとしても) しばらくして L3 から排出されるときにのみメモリーに書き込まれます。また、L3 はアンコア・サブシステムの一部であり、コアの一部ではありません。

ラインの変更による L3 排出のためのメモリーへのライトバックは、アンコア PMU ロジック内の個々のコアに関連付けることができません。そのため、すべてのコアの書き込み総帯域幅をアンコア PMU 内のイベントによって測定することができません。読み出し帯域幅と非テンポラルな書き込み帯域幅は、コア単位で測定できます。2 つの物理プロセッサが搭載されたシステムでは、メモリー帯域幅の NUMA 特性により、これらの 2 つのコンポーネントの測定値をソケットごとのコア帯域幅に区分する必要があります。

ソケットごとの読み出し帯域幅は次のイベントによって測定できます。

```
OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_LOCAL_DRAM
OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_REMOTE_DRAM
```

すべてのソケットの読み出し総帯域幅は次のイベントによって測定できます。

```
OFFCORE_RESPONSE_0.DATA_IFETCH.ANY_DRAM
```

非テンポラルなストアのソケットごとの帯域幅は次のイベントによって測定できます。

```
OFFCORE_RESPONSE_0.OTHER.L3_MISS_LOCAL_CACHE_DRAM
OFFCORE_RESPONSE_0.OTHER.L3_MISS_REMOTE_DRAM
```

非テンポラルなストアの総帯域幅は次のイベントによって測定できます。

```
OFFCORE_RESPONSE_0.OTHER.ANY.CACHE_DRAM
```

"CACHE\_DRAM" エンコーディングを使用すると、表 B-14 の脚注に記載した問題を回避できます。上記のイベントにはいずれも、変更されたキャッシュ可能ラインのライトバックに関連する帯域幅は含まれません。

### B.4.3.7 キャッシュミスに関するその他の L1/L2 イベント

OFFCORE\_RESPONSE\_0 イベントおよび後述する PEBS に加えて、同様に使用できるイベントがいくつかあります。また、offcore\_response\_0 イベントコードはカウンター 0 のみでサポートされるため、offcore\_response\_0 イベントを補助する追加のイベントも用意されています。

アーキテクチャーで定義されたイベント LONGEST\_LAT\_CACHE\_ACCESS を使用して L2 ミスをカウントすることもできます。しかし、このイベントは L1D/L2 ハードウェア・プリフェッチによる要求も含んでいるため、その効用は限定されます。前述の OFFCORE\_REQUESTS イベントに加えて、一部の L2 アクセスイベントは、L2 アクセスと L2 ミスの両方をタイプ別にドリルダウンするために使用できます。L2\_RQSTS イベントと L2\_DATA\_RQSTS イベントを使用して、分類されたアクセスタイプを識別します。すべての L2 アクセスイベントにおいて、PREFETCH は L2 ハードウェア・プリフェッチのみを表します。DEMAND は、L1D ハードウェア・プリフェッチによるロードおよび要求を含みます。

L2\_LINES\_IN イベントと L2\_LINES\_OUT イベントは、インテル® Core™2 プロセッサのイベントとは少し異なります。L2\_LINES\_OUT イベントは、クリーンおよびダーティーで排出されたラインを分離し (すなわち、ライトバック)、それらが L1D 要求または L2 ハードウェア・プリフェッチによって排出されたかどうかを判断するために使用できます。

イベント L2\_TRANSACTIONS は、L2 との相互作用をすべてカウントします。

書き込みとロックされた書き込みは、複合イベント L2\_WRITE でカウントされます。

L2\_RQSTS、L2\_DATA\_RQSTS、L2\_LINES\_IN、L2\_LINES\_OUT、L2\_TRANSACTIONS、L2\_WRITE などの派生イベントの詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 24H、26H、F1H、F2H、F0H、27H に示されています。

### B.4.3.8 TLB ミス

2 番目に数が多いメモリアクセス遅延は、リニアアドレスから物理アドレスへの変換が TLB 内の限られたエントリでマッピングされていることに関連します。第 1 レベルの TLB でのミスは、通常はアウトオブオーダー (OOO) 実行とコンパイラーのスケジューリングによって隠蔽可能な非常に小さなペナルティーです。共有状態の TLB でミスが発生するとページウォークが起動されますが、このペナルティーは実行時に表れます。

この (非 PEBS) TLB ミスイベントは、次の 3 つに分類できます。

- Y DTLB ミス (DTLB\_MISSES) および派生イベントは、イベントコード 49H でプログラムされます。
- Y ロード DTLB ミス (DTLB\_LOAD\_MISSES) および派生イベントは、イベントコード 08H でプログラムされません。
- Y ITLB ミス (ITLB\_MISSES) および派生イベントは、イベントコード 85H でプログラムされます。

ストア DTLB ミスは、DTLB ミスとロード DTLB ミスの差から評価できます。

それぞれ umask 値でプログラムされる多数のサブイベントがあります。上記のイベントの派生イベントの Umask に関する詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」を参照してください。

### B.4.3.9 L1 データキャッシュ

L1 データキャッシュ操作の分析に使用できるいくつかの PMU イベントがあります。これらのイベントは、4 つの汎用カウンターのうちの最初の 2 つ (すなわち、IA32\_PMC0 と IA32\_PMC1) によってのみカウントできます。ほとんどの L1D イベントは明らかです。

L1D の参照総数は、L1D\_ALL\_REF によってカウントできますが、キャッシュ可能な参照のみ、またはすべての参照がカウントされます。キャッシュ可能な参照は、L1D\_CACHE\_LOAD および L1D\_CACHE\_STORE によってロードおよびストアに分類できます。これらのイベントは、それぞれの Umask 値によってさらに MESI ステート別に分類され、I ステート参照はキャッシュミスを示します。

L1D 内で変更されたラインが排出されると L2 ヘライトバックが発生し、これらは、L1D\_WB\_L2 イベントによってカウントされます。これらは、umask 値により、さらに L2 内のキャッシュラインを MESI ステート別に分類できます。

ロックされた参照もまた、L1D\_CACHE\_LOCK イベントによってカウントでき、これも、L1D ラインの MESI ステート別に分類されます。

L1D に取り込まれるラインの総数、つまり M ステートに達した数と、スヌープにより排出される変更済みラインの数は、L1D イベントおよびその Umask のバリエーションによってカウントされます。

L1D イベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 28H、40H、41H、42H、43H、48H、4EH、51H、52H、53H、80H、83H に示されています。

ロードをアクティブなストアバッファから転送できないケースがあります。これは、主に長いロードがより短いストアとオーバーラップすることに関係します。この状況を検出するイベントはありません。また、「フォルス・ストア・フォワードリング」と呼ばれるケースでは、アドレスの下位 12 ビットだけが一致します。これは、4K エイリアシングと呼ばれることもあり、イベントコード 07H と Umask 01H を持つイベント「PARTIAL\_ADDRESS\_ALIAS」によって検出できます。

## B.4.4 フロントエンドの監視イベント

分岐予測ミスの影響は、コードの変更や拡張インライン展開によって排除できる可能性があります。その他大部分のフロントエンドのパフォーマンス低下は、コードの生成時に対処する必要があります。そのような問題の分析はコンパイラ開発者にとって有用です。

### B.4.4.1 分岐予測ミス

B.4.3.3 節で PEBS に関連して説明したリタイアした分岐イベントに加えて、分岐予測ミス機能は、LBR を使用した分岐先の分岐ロケーションと、PEBS バッファ内に取得されるターゲット・ロケーションを特定するように拡張されています。また、分岐予測に関連するその他の多くの PMU イベント (イベントコード E6, E5, E0, 68, 69) は、パフォーマンス・チューニングよりもハードウェア設計に関連します。

分岐予測ミスは、それ自体はパフォーマンス・ボトルネックの指標ではありません。これは、ディスパッチのストールおよび命令スタベーション条件 (UOPS\_ISSUED.C1:11 – RESOURCE\_STALLS.ANY) に関連付けられます。このようなストールは、命令キャッシュミスや ITLB ミスに関連する可能性があります。このような状況には、プリサイズ ITLB ミスイベントが役立つ場合があります。命令キャッシュ・ミスイベントと ITLB ミスイベントは、イベントコード 80H, 81H, 82H, 85H, AEH に示されます。

### B.4.4.2 フロントエンドのコード生成メトリック

そのほかのフロントエンド・イベントは、コード生成がアウトオブオーダー (OOO) エンジンに対する命令のデコードおよびマイクロオペレーション (μop) の発行の問題と適切に作用しない状況を特定するのに役立ちます。例えば、レンジ変更プリフィクスに関連する 16 ビット即値、ROB 読み出しポートのストール、命令アライメントとループ検出の干渉、命令をデコードする帯域幅の制約の問題などがあります。LSD (Loop Stream Detector) の動作は、シグナル監視機能で CMASK 値を使用して監視します。これらのイベントのうちのいくつかは、「インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B」の第 19 章「Performance Monitoring Events」にあるイベントコード 17H, 18H, 1EH, 1FH, 87H, A6H, A8H, D0H, D2H に示されます。

一部の命令 (FSIN, FCOS、その他の超越関数命令) は、MSROM のアシストによってデコードされます。複雑なマイクロオペレーション (μop) フローをデコードするため MSROM のアシストを受ける命令が頻繁に発行されることがあります。これは、そのような状況を減らすため命令選択を改善する必要性を示します。UOPS\_DECODED.MS イベントを使用して、命令選択の改善によるメリットが得られる可能性があるコード領域を特定できます。

その他、デノーマル FP 値や QNaN など非正規 FP 値に対して起動される FP アシストによっても、このイベントがトリガーされる可能性があります。そのような場合のペナルティーは、アシストに必要なマイクロオペレーション (μop) の実行と、適切な状態を確保するために行われるパイプラインのクリアです。

その結果この状況は、MACHINE\_CLEAR.CYCLES による非常に明確なシグネチャーとマイクロコード・シーケンサー UOPS\_DECODED.MS によって挿入されるマイクロオペレーション (μop) を持ちます。実行ペナルティーはこの 2 つの和となります。これらのイベントコードは、D1H および C3H に示されています。

## B.4.5 アンコア・パフォーマンス監視イベント

アンコア・サブシステムには、図 B-4 の図に示した L3、IMC、インテル® QPI の各ユニットが含まれます。アンコア・サブシステム内のアンコア PMU は、8 つの汎用カウンターと 1 つの固定カウンターで構成されます。アンコア

内の固定カウンターは、コアとは異なる周波数で動作するアンコア・クロック・ドメインの unhalted クロックサイクルを監視します。

アンコアは、PMI 割り込みを単独で生成することができません。コア PMU が論理プロセッサ単位でコア PMI を発生させるのに対して、アンコア PMU は、プロセッサ・コア内の割り込みハードウェアを使用してコア単位で PMI を発生します。アンコアのカウンターがオーバーフローすると、どのコアにシグナルを送信して PMI を発生させるか、ビットパターンが使用されます。アンコア PMU は、カウンターがオーバーフローしたイベントを発生させたコア、プロセッサ ID、またはスレッド ID を認識できません。そのため、アンコアイベントのサンプリングを行う最も合理的な手法は、パッケージ内のすべての論理プロセッサで PMI を発生させることです。

キューの占有と挿入を監視するさまざまなイベントがあります。そのほかに、キャッシュライン転送、DRAM ページングポリシーの統計情報、スヌープのタイプ/応答などをカウントするイベントも用意されています。アンコアは、メモリーに対する総帯域幅を測定できる唯一の場所です。これについては、アンコアのすべてのコンポーネントおよびそのイベントを示した後で説明します。

### B.4.5.1 グローバルキューの占有

各プロセッサ・コアは、L2 ミスによるメモリー・アクセス・トラフィックの要求をバッファーに格納するスーパーキューを持っています。アンコアは、これらのプロセッサ・コアからのトランザクション要求を処理するためグローバルキューを保持し、L3、IMC、またはインテル® QPI の各リンクから到達したデータ・トラフィックをバッファーに格納します。

グローバルキュー (GQ) 内には、3 つのトランザクション・タイプに対応する次の 3 つの「トラッカー」があります。

- Y パッケージ上の読み出し要求: 32 エントリーのトラッカーキュー。
- Y パッケージ上の書き込み要求: 16 エントリーのトラッカーキュー。
- Y 「ピア」から到達した要求: 12 エントリーのトラッカーキュー。

「ピア」は、インテル® QuickPath インターコネクトからの任意の要求を表します。

3 つのトラッカーすべてに対し、占有、挿入、一杯であるサイクル、空でないサイクルを監視できます。また、ロード要求がステージを通過する際に、各ステージに関連する占有と挿入を監視できます。これにより、ロードによるアンコア・メモリー・アクセスの「サイクル・アカウンティング」による詳細な調査が可能となります。

アンコア・カウンターでキューの占有率を監視する場合、最初にすべてのアンコアキューを空にする必要があります。これは、バスロックを発行するソフトウェア・ツールのドライバーによって行なわれます。この操作は、カウンターを最初にプログラムするときのみ行います。それ以降、カウンターはキューの状態を正しく反映するため、(例えば、別のバスロックが発行されることなく) サンプリングが繰り返し実行されます。

GQ の割り当て (UNC\_GQ\_ALLOC) および GQ のトラッカー占有 (UNC\_GQ\_TRACKER\_OCCUP) を監視するアンコアイベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 03H および 02H に示されています。この 3 つのトラッカーは、Umask 値を指定して選択します。これらの派生イベントのモニタには、読み出しトラッカーを表す「RT」、書き込みトラッカーを表す「WT」、ピア・プローブ・トラッカーを表す「PPT」という表記が使用されます。

データが供給された直後に占有が停止する場合、キュー占有の平均継続期間によってレイテンシーを測定できます。キュー占有の平均継続期間は、 $UNC\_GQ\_TRACKER\_OCCUP.X/UNC\_GQ\_ALLOC.X$  の比率によって測定できます。ここで、「X」は特定の Umask 値を表します。読み出しトラッカーの総占有期間は、次の式によって測定されます。

$$\text{Total Read Period (総読み出し期間)} = UNC\_GQ\_TRACKER\_OCCUP.RT/UNC\_GQ\_ALLOC.RT$$

この期間にはブッキングとクリアの時間が含まれるため、データ供給のレイテンシーよりも長くなります。次の測定値について考えます。

$$\text{LLC response Latency (LLC 応答レイテンシー)} = \text{UNC\_GQ\_TRACKER\_OCCUP.RT\_TO\_LLC\_RESP} / \text{UNC\_GQ\_ALLOC.RT\_TO\_LLC\_RESP}$$

これは基本的に定数です。スヌープの合計時間を含んでおらず、変更されたラインを別のコアから取得します (例えば、L3 のスキャン時間のみと、変更されたラインがこのソケット内に存在するかどうかを確認します)。

L3 ヒットの合計レイテンシーは、次の 3 つの加重平均となります。

- Y 要求を行ったコアによってのみラインが使用された単純なヒットのレイテンシー
- Y 複数のコアによるクリーンなラインへのアクセスに関するレイテンシー
- Y 複数のコアによってアクセスされたダーティーなラインへのアクセスに関するレイテンシー

ロードに関するこの 3 つの L3 ヒットの構成要素は、OFFCORE\_RESPONSE の派生イベントを使用して分類できます。

- Y OFFCORE\_RESPONSE\_0.DEMAND\_DATA.L3\_HIT\_NO\_OTHER\_CORE
- Y OFFCORE\_RESPONSE\_0.DEMAND\_DATA.L3\_HIT\_OTHER\_CORE\_HIT
- Y OFFCORE\_RESPONSE\_0.DEMAND\_DATA.L3\_HIT\_OTHER\_CORE\_HITM

OFFCORE\_RESPONSE\_0.DEMAND\_DATA.LOCAL\_CACHE イベントは、レイテンシーを求めるための分母として使用します。個々のレイテンシーはマイクロベンチマークによって測定する必要がありますが、ロード帯域幅の影響が含まれるため、プリサイズ・レイテンシー・イベントを使用する方がはるかに効果的です。

L3 ミス構成要素は次の 3 つの加重平均です。

- Y 別のソケット上のキャッシュにおける L3 ヒットのレイテンシー (前のセクションで説明)
- Y ローカル DRAM に対するレイテンシー
- Y リモート DRAM に対するレイテンシー

ローカル DRAM アクセスとリモート・ソケット・アクセスは、さらに多くのアンコアイベントによって分類できます。これについては後述します。

$$\text{Miss to fill latency (ミスからフィルまでのレイテンシー)} = \text{UNC\_GQ\_TRACKER\_OCCUP.RT\_LLC\_MISS} / \text{UNC\_GQ\_ALLOC.RT\_LLC\_MISS}$$

\*RTID\* ニーモニックに関連する Umask 値を使用するアンコア GQ イベントにより、GQ と QHL 間の通信に関連するミスからフィルまでのレイテンシーのサブコンポーネントの監視が可能になります。

上記の 3 つのトラッカーが空でない (エントリーが 1 つ以上ある) または一杯でない場合にサイクルを監視するアンコア PMU イベントがあります。これらのイベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 00H および 01H に示されています。

一般に、アンコア PMU は特定のイベント条件を発生させるプロセッサ・コアを区別しないため、レイテンシーをキューの平均占有率で割ることによってペナルティーを判定する手法はアンコアには適用できません。異なるコアのエントリーがオーバーラップしても、ペナルティーはオーバーラップしないためストールサイクルは減少せず、各コアは、単独で全レイテンシーの影響を受けます。

コア単位の変更を評価するには、目的のコアからのエントリーのサイクル数を取得する必要があります。これには、\*NOT\_EMPTY\_CORE\_N タイプのイベントが必要となりますが、そのようなイベントは存在しません。したがって、サ



イクルを分類する際に、全レイテンシーを適用してペナルティを概算する必要があります。前述のように、個々のサンプルのレイテンシーとともにデータソースも収集されるため、最善の方法は PEBS レイテンシーイベントを使用することです。

前述の読み出しトラッカーの個々の構成要素もまた、cmask 値を 1 または 32 に設定し、それを関連する RT 占有イベントに適用することにより、ビジーまたはフルとして監視できます。

表 B-15 占有サイクルのアンコア PMU イベント

アンコア PMU イベント	Cmask	Umask	イベントコード
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_FULL	32	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_FULL	32	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_FULL	32	08H	02H
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_BUSY	1	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_BUSY	1	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_BUSY	1	08H	02H

### B.4.5.2 グローバルキューポートイベント

GQ データバッファのトラフィックは、個別のポートを経由して異なるサブシステムに出入りするフローを制御します。

- Y コアトラフィック: 2 つのポートでデータトラフィックを処理します。各ポートは、1 組のプロセッサ・コア専用に使われます。
- Y L3 トラフィック: 1 つのポートで L3 データトラフィックを処理します。
- Y インテル® QPI トラフィック: 1 つのポートでインテル® QPI ロジックへのトラフィックを処理します。
- Y IMC トラフィック: 1 つのポートで統合型メモリー・コントローラーへのデータトラフィックを処理します。

L3 トラフィックおよびコアトラフィック用ポートは、サイクルごとに一定のビットを転送します。ただし、インテル® QuickPath インターコネクト・プロトコルでは、インテル® QPI および IMC 読み出しポート上で 8/16 バイトのいずれかでデータが転送されます。したがって、このイベントはデータ転送および総帯域幅を測定するために使用できません。

トラフィックのフローを区別できるアンコア PMU イベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 04H および 05H に示されています。

### B.4.5.3 グローバルキュースヌープイベント

コアから、あるいはリモートパッケージまたは I/O ハブからのキャッシュライン要求は、GQ によって処理されます。いずれかのコアからのキャッシュライン要求をアンコアが受信すると、GQ はまず L3 をチェックして、ラインがパッケージに存在するかどうかを確認します。L3 はインクルーシブであるため、このチェックは迅速に行われます。ラインが L3 に存在し、要求元のコアによって所有されている場合、データは L3 からコアに直接返されます。

ラインが複数のコアによって所有されている場合、GQ はほかのコアをスヌープして、変更されたコピーが存在するかどうかを確認します。存在する場合は、L3 が更新され、ラインが要求元のコアに送信されます。

L3 ミスが発生している場合、GQ はローカル・メモリー・コントローラーに（またはインテル® QPI リンク経由で）ラインの要求を送信する必要があります。データがリモート L3 に存在する、またはローカル DRAM に存在しない場合、要求は、インテル® QPI 経由でリモート L3（またはリモート DRAM）に送信されます。各物理パッケージはローカルの統合メモリー・コントローラーを持つため、GQ は、要求されたキャッシュラインの「ホーム」ロケーションを物理アドレスから特定する必要があります。ホームがローカルパッケージ上にあることがアドレスから特定されると、GQ は、同時にローカル・メモリー・コントローラーに対して要求を行います。ホームがリモートパッケージに属していることが特定されると、インテル® QPI 経由で送信された要求はリモート IMC にアクセスします。

また GQ は、インテル® QuickPath インターコネクタからのキャッシュライン要求に対するスヌープ応答を処理します。このスヌープ・トラフィックは、ピア・プローブ・トラッカー内のキュー・エンタリーに対応しています。

スヌープ応答は、ローカルホームとリモートホームのデータの要求に分けられます。ラインが変更された状態であり、GQ が読み出し要求に回答している場合、ラインはメモリーにライトバックされる必要があります。これは、ラインが再び変更されるため RFO への応答は無駄な作業であると考えられます。そのため、RFO に対してライトバックは行われません。

アンコア PMU によって監視可能なローカル・ホーム・イベントのスヌープ応答は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 06H に示されています。リモート・ホーム・イベントのスヌープ応答は、イベントコード 07H に示されます。

いくつかの関連イベントは、ほかのキャッシュ・エージェント (プロセッサまたは IOH) からのスヌープに回答して、MESI 状態の遷移をカウントします。これらの一部は MSR のプログラミングに依存しており、MSR は 1 つしかないため、一度に 1 つずつしか測定できません。インテル® パフォーマンス・ツールは、これらのイベントを 1 つの汎用アンコア・カウンターに制限することで正しくスケジュールします。

#### B.4.5.4 L3 イベント

L3 ヒットおよび L3 ミスの数は GQ トラッカー割り当てイベントから特定できますが、アンコア PMU イベントの方が簡単に使用できるものがあります。これらは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるアンコア・イベント・リストのイベントコード 08H および 09H に示されています。

割り当てられたラインと干渉されたラインの MESI ステートの分類は、イベントコード 0AH および 0BH を使用して、アンコアイベント `LINES_IN`、`LINES_OUT` で監視することもできます。詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」に示されます。

#### B.4.6 インテル® QuickPath インターコネクタのホームロジック (QHL)

データが L3 をミスし、アンコアの GQ によってトランザクション要求が送信されると、インテル® QPI ファブリックは、ローカル DRAM コントローラーから、または別の物理パッケージのリモート DRAM コントローラーからの要求を処理します。GQ は、要求されたキャッシュラインの「ホーム」ロケーションを物理アドレスから特定します。ホームがローカルパッケージ上にあることが特定されると、GQ は、ローカルメモリー・コントローラー (統合メモリー・コントローラー (IMC)) に対して要求を送ります。ホームがリモートパッケージに属していることが特定されると、インテル® QPI 経由で送信された要求はリモート IMC にアクセスします。

インテル® QPI ロジックと IMC は、アンコア・サブシステムにある異なるユニットです。インテル® QPI ロジックは、「キャッシュ・エージェント」および「ホーム・エージェント」の概念から、ローカル IMC とリモート IMC を識別します。具体的には、インテル® QPI プロトコルにより、各ソケットが「キャッシュ・エージェント」と「ホーム・エージェント」のどちらを持っているかが考慮されます。

Y キャッシュ・エージェントはアンコアの GQ および L3 である (存在する場合は、IOH)。

Y ホーム・エージェントは IMC である。

L3 ミスにより、すべてのキャッシュ・エージェントおよびホーム・エージェントのラインが (場所に関係なく) 同時に照会されます。

要求されたラインを別のソースがより迅速に供給できる場合、QHL 要求に優先することがあります。パッケージ上の要求によるローカル・ホーム・ラインに対する L3 ミスは、同時に QHL とインテル® QPI にも送られます。リモートのキャッシュ・エージェントがラインを最初に供給した場合、その QHL への要求に対して、トランザクションが完了し

たことを示すシグナルが送信されます。リモートのキャッシュ・エージェントが読み出し要求に回答して、変更されたラインを返す場合、更新されたラインをライトバックして DRAM のデータを更新する必要があります。

インテル® QPI が、ローカルのホームラインに対するスヌープ要求を GQ と QHL の両方に送信する場合にも、同様の制御シグナルフローがあります。そのラインを L3 が保持する場合、L3/GQ によってトランザクションが完了したことを QHL に知らせる必要があります。L3 (またはコア) のラインが変更され、リモートパッケージからのスヌープ要求がロードに関連する場合、QHL は、ライトバックを完了し、そのラインをインテル® QPI に転送してトランザクションを完了する必要があります。

アンコア PMU は、QHL のオペコード一致機能を使用して、アンコアのキャッシュライン・アクセスとライトバックのトラフィックを監視するイベントを提供します。オペコード一致機能については、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 33 章「Performance Monitoring」で説明しています。オペコード一致機能を使用するアンコア PMU イベントは、イベントコード 35H に示されています。表 B-16 に、QHL オペコード一致機能をプログラムに有用な設定を示します。

表 B-16 一般的な QHL オペコード一致機能のプログラミング

ロード・レイテンシーのプリサイズイベント	MSR 0x396	Umask	イベントコード
UNC_ADDR_OPCODE_MATCH.IOH.NONE	0	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDSD	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPIWB	40001D00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.NONE	0	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDI	40001900_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDSD	40001A00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPIWB	40001D00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.NONE	0	4H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDSD	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPIWB	40001D00_00000000	1H	35H

上記の定義済みのオペコード一致機能のエンコーディングは、HITM アクセスを監視するために使用できます。これは、HITM 転送を要求するコードのプロファイルを可能にする唯一のイベントです。

図 B-8 ~ 図 B-15 に、キャッシュラインのローカルホームおよびリモートキャッシュの MESI ステートのさまざまな組み合わせにおける、L3 ミス発生後のデータ読み出しおよび所有権読み出し (RFO) に関連する一連のインテル® QPI プロトコル交換を示します。データがリモート L3 に存在する場合でもリモート QHL から送信されるケースに特に注意してください。それらは、M ステートのラインを持つリモート L3 による読み出しデータの場合です。

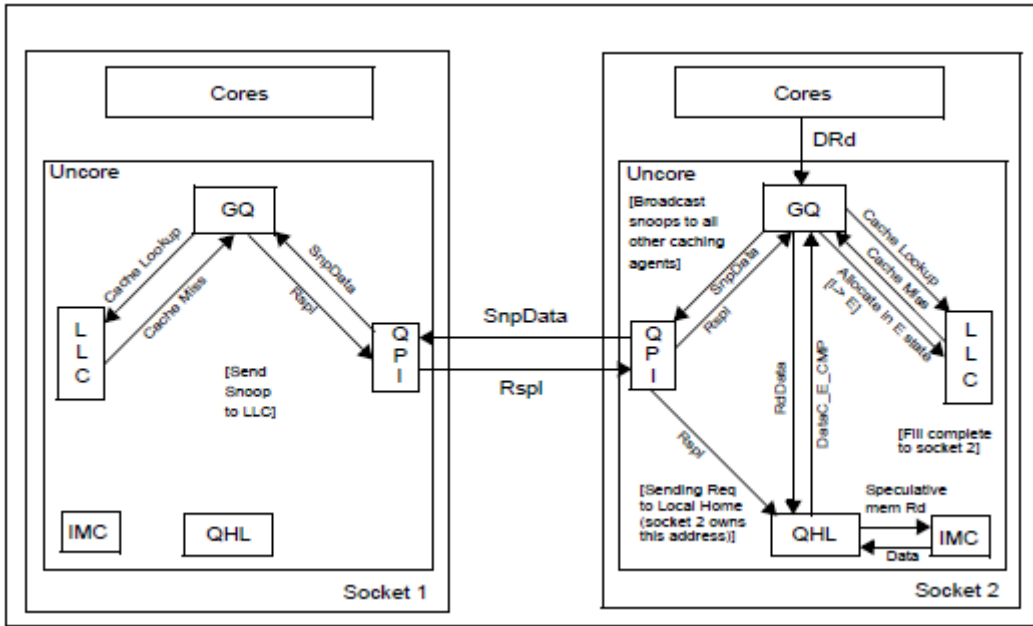


図 B-8 ローカルホームに対する LLC ミス発生後の RdData 要求 (CLEAN 応答)

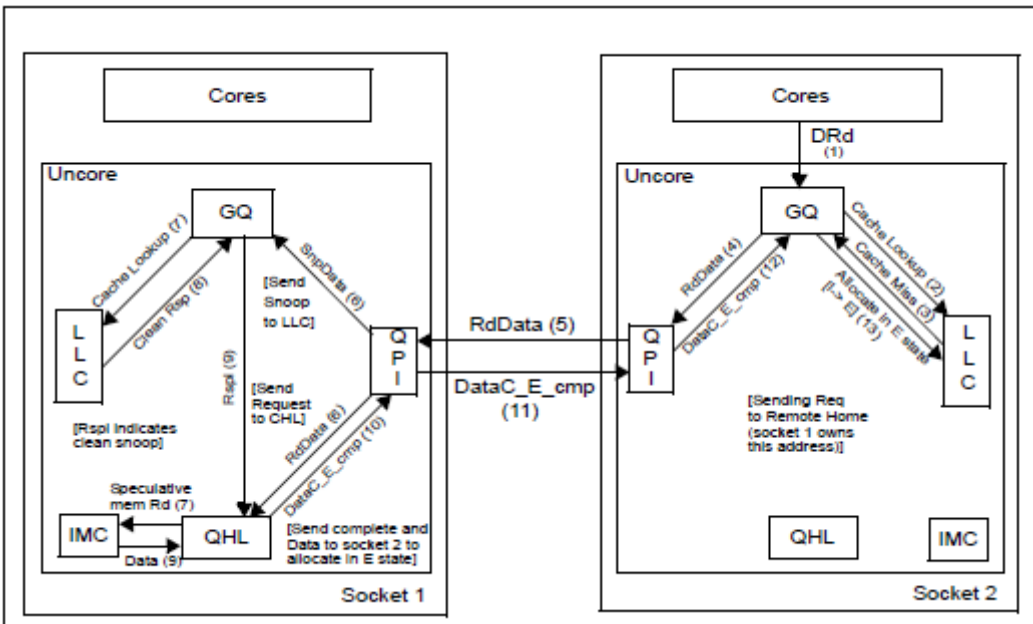


図 B-9 リモートホームに対する LLC ミス発生後の RdData 要求 (CLEAN 応答)

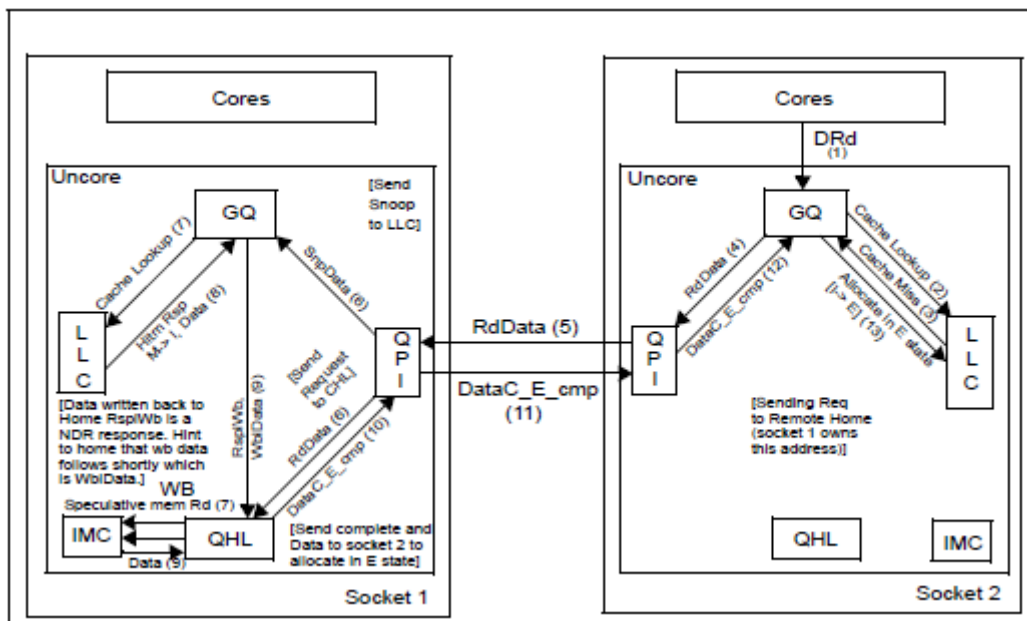


図 B-10 リモートホームに対する LLC ミス発生後の RdData 要求 (HITM 応答)

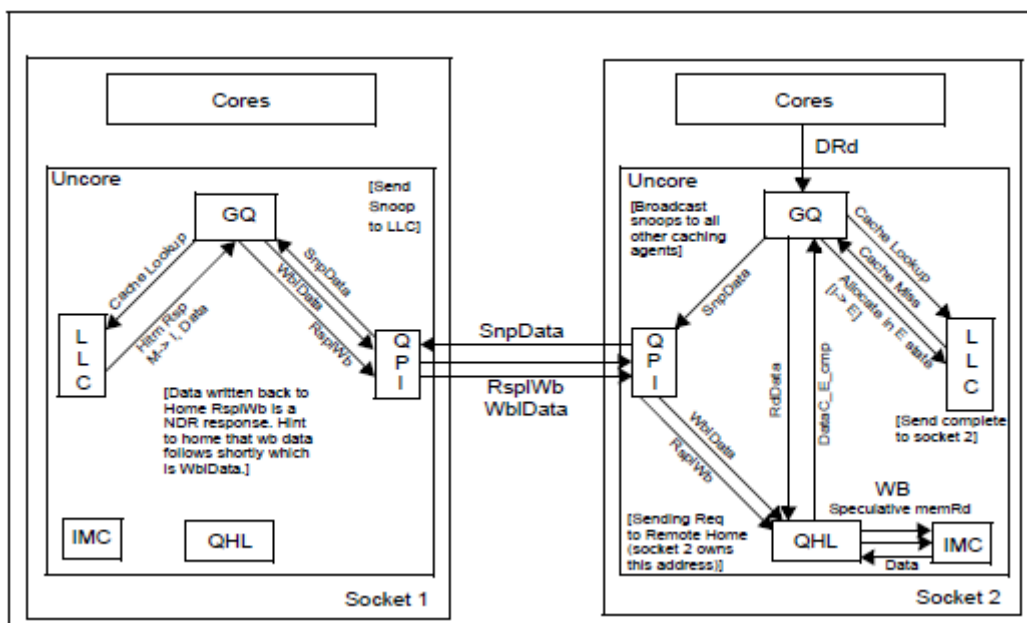


図 B-11 ローカルホームに対する LLC ミス発生後の RdData 要求 (HITM 応答)

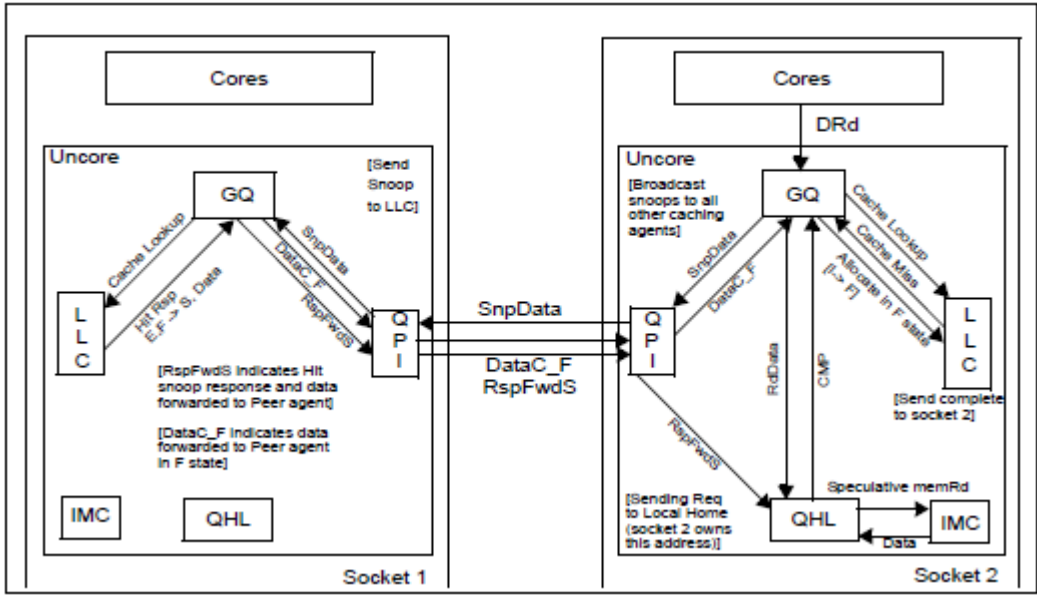


図 B-12 ローカルホームに対する LLC ミス発生後の RdData 要求 (HIT 応答)

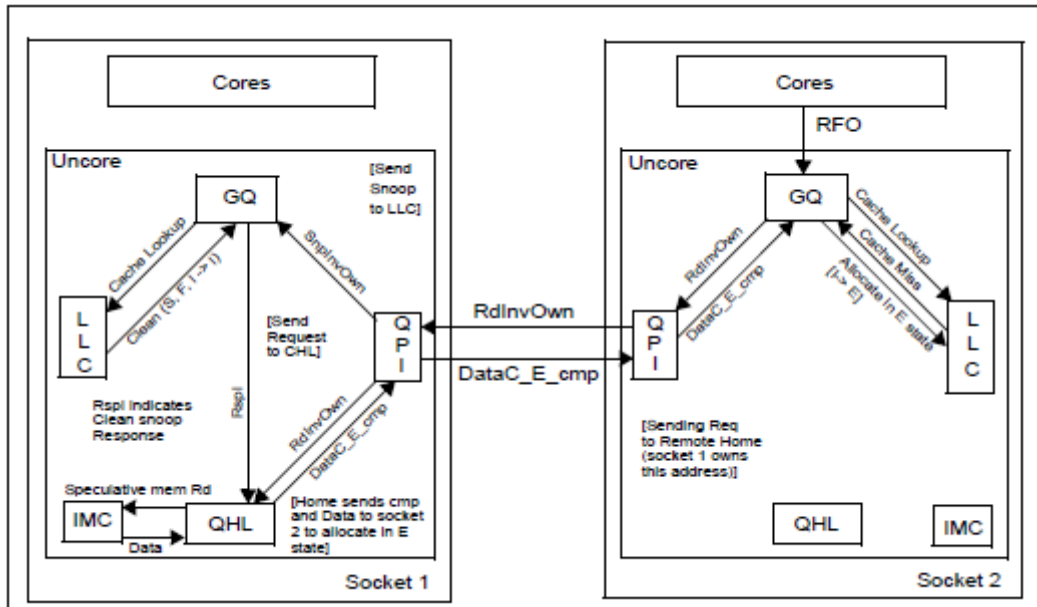


図 B-13 リモートホームに対する LLC ミス発生後の RdInvOwn 要求 (CLEAN 応答)

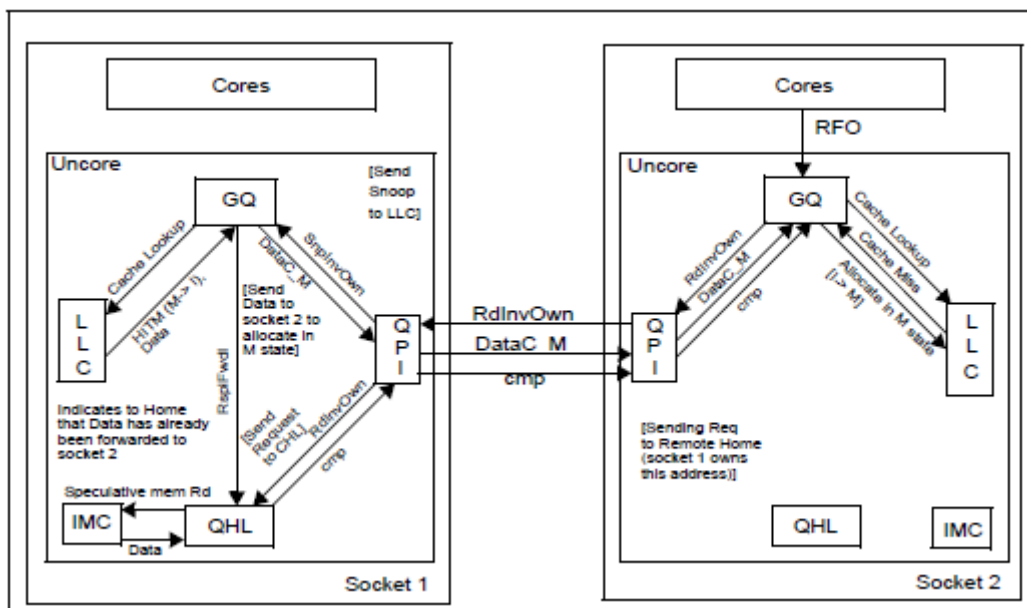


図 B-14 リモートホームに対する LLC ミス発生後の RdInvOwn要求 (HITM 応答)

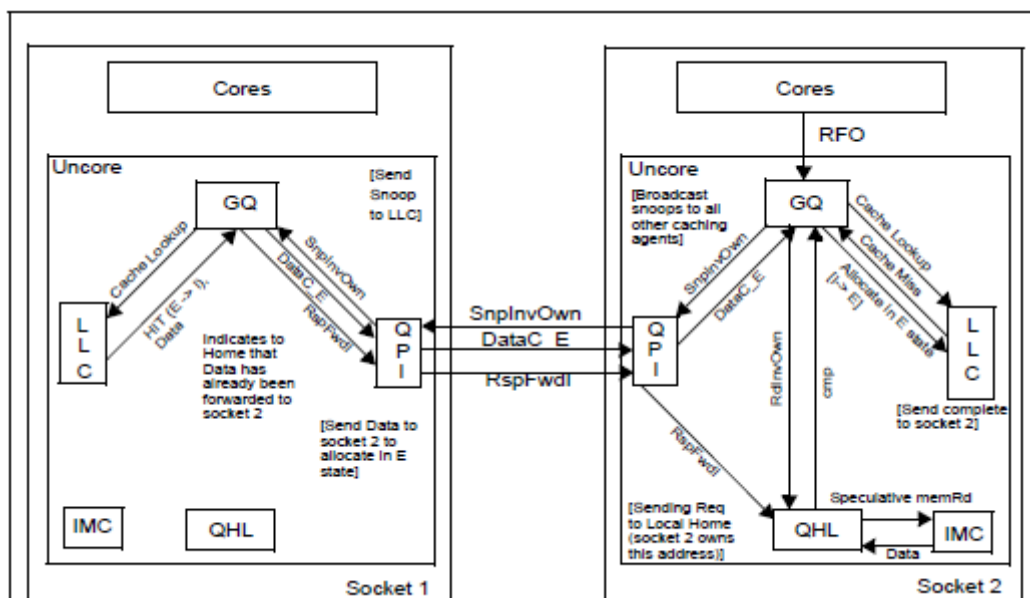


図 B-15 ローカルホームに対する LLC ミス発生後の RdInvOwn要求 (HIT 応答)

ラインは、ローカル/リモートのいずれが「ホームであった」にかかわらず、発生元の GQ がラインを受信する前に DRAM にライトバックされる必要があります。そのため、ラインは常に QHL から送信されたように見えます。RFO では、この処理を行いません。しかし、リモートの RFO (SnpInvOwn) に応答しており、ラインが S または F ステートである場合、キャッシュラインが無効化され、ラインは QHL から送信されます。問題は、データソースがそれほど明確でないことがあるという点です。

### B.4.7 アンコアからの帯域幅測定

読み出しの帯域幅は、OFFCORE\_RESPONSE\_0.DATA\_IN.LOCAL\_DRAM や OFFCORE\_RESPONSE\_0.DATA\_IN.REMOTE\_DRAM などのイベントを使用して、コアごとに測定できます。総帯域幅には書き込みも含まれますが、ほとんどの場合、これらは L3 の変更されたラインの排出によって発生するため、コアから監視できません。したがって、あるコアによって使用され、変更されたラインは、無関係のタスクを実行している別のコアの書き込みによる排出のため DRAM にライトバックされる可能性があります。アンコアでは、変更されたキャッシュラインおよび (例えば、非テンポラルなストリーミング・ストアによって書き込まれた) キャッシュ不可ライ

ンのライトバックは異なる方法で処理され、ライトバックにより各種イベントが多様な方法でカウントアップされます。

DRAM に書き込まれるラインはすべて、UNC\_IMC\_WRITES.FULL.\* イベントによってカウントされます。これには、変更されたキャッシュラインのライトバックと (例えば、非テンポラルなインテル® SSE ストア命令によって生成された) キャッシュ不可ラインの書き込みが含まれます。リモートソケットからのキャッシュ不可ラインの書き込みは、UNC\_QHL\_REQUESTS.REMOTE\_WRITES によってカウントされます。ローカルコアからのキャッシュ不可ラインのライトバックは、UNC\_QHL\_REQUESTS.LOCAL\_WRITES によってカウントされません。このイベントは、ローカルにキャッシュされたラインのライトバックだけをカウントします。

UNC\_IMC\_NORMAL\_READS.\* イベントは読み込みのみをカウントします。UNC\_QHL\_REQUESTS.LOCAL\_READS と UNC\_QHL\_REQUESTS.REMOTE\_READS は、読み出しと「InvtoE」トランザクションをカウントします。これらは、キャッシュ不可の書き込み (USWC/UC の書き込みなど) に対して発行されます。これにより、UNC\_QHL\_REQUESTS.LOCAL\_READS + UNC\_QHL\_REQUESTS.REMOTE\_READS - UNC\_IMC\_NORMAL\_READS.ANY の差を計算することによってキャッシュ不可の書き込みを算出できます。

帯域幅の評価に有用なアンコア PMU イベントは、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」にあるイベントコード 20H、2CH、2FH に示されています。

## B.5 インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge のパフォーマンス・チューニング手法

この節では、パフォーマンス監視イベントを使用したさまざまなパフォーマンス・チューニング手法について説明します。一般的に、一部の手法はほかのマイクロアーキテクチャーにも応用できますが、ほとんどのパフォーマンス・イベントはインテル® マイクロアーキテクチャー開発コード名 Sandy Bridge に固有なものです。

### B.5.1 パフォーマンスのボトルネックとソース行の関連付け

パフォーマンス分析ツールは、イベントをサンプリングして命令ポインターアドレス (IP) からホットスポットを特定することにより、パフォーマンスのボトルネックが潜在するコード領域を特定可能にします。

このサンプリング手法では、パフォーマンス・カウンターのオーバーフローにより発生するパフォーマンス監視割り込み (PMI) に応答するサービスルーチンが必要になります。イベント発生条件のパフォーマンス監視イベントの検出と実際の命令ポインターアドレスの間にはある程度のずれが生じることがあります。これは、「スキッド」と呼ばれます。言い換えれば、イベントスキッドは、問題を発生した命令とイベントがタグ付けされている命令間の距離のことです。一般的にスキッドにはいくつかの注意すべき点があります。

- Y プリサイズイベントでは、リタイアした次の IP に対して、定義された 1 命令分のイベントスキッドが生じます。発生した命令が分岐である場合、このイベントは分岐ターゲットでタグ付けされており、分岐命令から分離できます。そのため、プリサイズイベントを使用したサンプリングでは、ボトルネックのコード領域を正確に特定する際にノイズの影響が減少する可能性が高くなります。
- Y 一般に、パフォーマンス・イベントを使用する場合、イベント生成条件がパフォーマンスに与える影響が大きいほどスキッドは短くなりますが、その逆も当てはまります。次の例は、この規則を説明しています。
  - ストア・フォワーディング・ブロックの問題は、10 サイクルを超える遅延を引き起こす可能性があります。ストア・フォワーディング・ブロック・イベントをサンプリングすると、ほとんどの場合、ブロックされたロードの後に続く数個の命令にタグ付けされます。
  - 反対に、正常に転送されたロードをサンプリングすると、スキッドが長くなるため、パフォーマンス・チューニングにはあまり役立ちません。
- Y スキッドは、イベント発生条件が命令のリタイアメントに近いほど短くなります。パイプラインのフロントエンドで発生したイベントは、実行時またはリタイア時に処理されるイベントよりも、原因となる命令から離れた命令にタグ付けされる傾向があります。



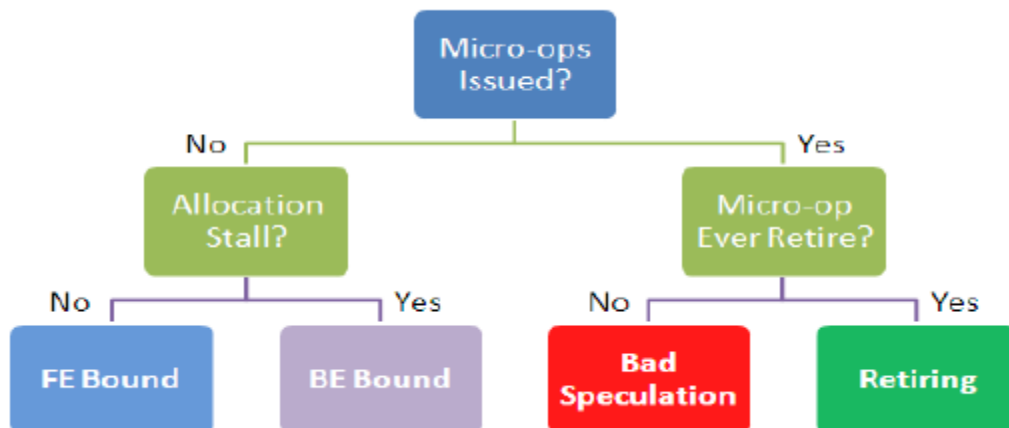
- Y イベント CPU\_CLK\_UNHALTED.THREAD によってカウントされるサイクルは、パイプライン内のより大きなボトルネックの後の命令に対して、より大きなカウントをタグ付けすることがあります。1 つの命令に対してサイクル数が累積されていれば、おそらく直前の命令でボトルネックが発生しています。
- Y フロントエンドで発生している低コストの問題の原因を判定することは非常に困難です。フロントエンドのイベントは、問題を発生している実際の命令の直前の IP までスキッドする可能性があります。

## B.5.2 階層的なトップダウン・パフォーマンス特性方式とパフォーマンス・ボトルネックの特定

インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge は、マイクロアーキテクチャー・パイプラインのどの部分がストールしているか絞り込むのを支援するいくつかのパフォーマンス・イベントを導入しています。これは、階層的なアプローチにより CPU サイクルがマイクロアーキテクチャー・パイプラインで時間を費やすのワークロードを特性化するのを可能にします。上位レベルに、CPU サイクルを分類する次のような 4 つの領域があります。パイプラインのどの部分がストールしているかを判定するには、フロントエンドによって供給されたマイクロオペレーション (μop) をキューに格納し、アウトオブオーダー・バックエンド (2.4.1 節を参照) へ供給するバッファを調べる必要があります。このバッファは、マイクロオペレーション (μop) キューと呼ばれます。マイクロオペレーション (μop) キューに関連する、次の 4 種類のストール条件があります。

- Y フロントエンドのストール: パイプラインのバックエンドがマイクロオペレーション (μop) を要求しているときに、フロントエンドが 1 サイクルあたり 4 個未満のマイクロオペレーション (μop) を供給しています。このようなストールが発生すると、アウトオブオーダー (OOO) エンジンのリネーム/割り当て処理が待機状態となります。そのため、実行はフロントエンドに制約がある (制限される) と言われます。
- Y バックエンドのストール: パイプラインのバックエンドに追加のマイクロオペレーション (μop) を受け入れるリソースが不足しているため、マイクロオペレーション (μop) キューからマイクロオペレーション (μop) が供給されていません。このようなストールが発生する場合、実行はバックエンドに制約がある (制限される) といわれます。
- Y バッド・スペキュレーション: 正常にリタイアしない命令のスペキュレーティブ・エグゼキューション (投機的な実行) をパイプラインが行っています。最もよくあるケースは、パイプラインが分岐ターゲットを予測する場合の分岐予測ミスです。これは、分岐の実行を待機する代わりに、パイプラインを一杯の状態に維持するために行われます。プロセッサの予測が誤っている場合、推測された命令をリタイアせずに、パイプラインをフラッシュする必要があります。
- Y リタイア: 最終的にリタイアするマイクロオペレーション (μop) をマイクロオペレーション (μop) キューが供給します。一般に、マイクロオペレーション (μop) はプログラムコードから生成されます。ただし、マイクロコード・シーケンサーがマイクロオペレーション (μop) を供給してパイプライン内の問題を処理するアシストは例外です。

次の図は、命令の実行を論理的に分類する方法を示します。



次の数式をインテル® マイクロアーキテクチャー開発コード名 Sandy Bridge のコア PMU パフォーマンス・イベントとともに使用すると、各カテゴリーに要する実行スロット割合を算出できます。

```
%FE_Bound =
    100 * (IDQ_UOPS_NOT_DELIVERED.CORE / N);
%Bad_Speculation =
    100 * ( (UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + 4 *
    INT_MISC.RECOVERY_CYCLES) / N);
%Retiring = 100 * ( UOPS_RETIRED.RETIRE_SLOTS / N);
%BE_Bound = 100 * (1 - (FE_Bound + Retiring + Bad_Speculation));
```

N は、実行スロットの総数を表し、サイクル数に 4 を掛けた値となります。

Y N = 4\*CPU\_CLK\_UNHALTED.THREAD

次の節では、バックエンドのストール、フロントのストール、バッド・スペキュレーションという 3 つのカテゴリーのペナルティ・サイクルの原因について説明します。各節では、プロセス、モジュール、関数、命令単位に適用される数式を使用します。

### B.5.2.1 バックエンド依存の特性

%BE\_Bound メトリックが報告された場合、ユーザーはバックエンドで考えられる次のレベルの問題を掘り下げて調査する必要があります。ここで紹介する方法論では、サイクルごとの実行ユニットの占有率に基づいてバックエンドのストールを調査します。当然のこととして、すべての実行リソースがビジーに保たれると最適なパフォーマンスを達成することができます。現在、この方法論では**バックエンド依存**を**メモリー依存**と**コア依存**の 2 つのカテゴリーに分類しています。

“メモリー依存” は、メモリー・サブシステムに関連するストールに相当します。例えば、キャッシュミスは最終的に実行のスタベーションを引き起こす可能性があります。また、“コア依存” は実行もしくは OOO クラスタのストールに関連するため、若干扱いにくいことがあります。これらのストールは、実行のスタベーションや適切ではない実行ポートの利用が原因であることが明らかです。例えば、長いレイテンシーの除算操作は、実行ポートに対して特定の  $\mu\text{op}$  タイプがプレッシャーを与え、サイクルで利用されるポートが減少することからしばらくの間命令スタベーションを引き起こし、実行をシリアル化する可能性があります。

これを算出するには、実行ユニットでパフォーマンス監視イベントを使用します：

```
%BE_Bound_at_EXE =
    (CYCLE_ACTIVITY.CYCLES_NO_EXECUTE + UOPS_EXECUTED.THREAD:c1 -
    UOPS_EXECUTED.THREAD:c2) / CLOCKS
```

CYCLE\_ACTIVITY.CYCLES\_NO\_EXECUTE イベントは、 $\mu\text{op}$  がまったく実行されなかったスタベーション・サイクルの完了をカウントします。

UOPS\_EXECUTED.THREAD:c1 と UOPS\_EXECUTED.THREAD:c2 は、1 サイクルで少なくとも 1 または 2 つの  $\mu\text{op}$  が実行されたサイクルをカウントします。したがって、イベントカウントの差から OOO バックエンドが 1  $\mu\text{op}$  しか実行できなかったサイクルを測定できます。

%BE\_Bound\_at\_EXE メトリックは、実行ユニットのパイプライン・ステージでカウントされるため、アロケーション・ステージでカウントされる Backend\_Bound 比率とは一致しません。しかし、両方のカウンターは実行がバックエンド依存 (両方が高い場合) であることを確認するのに使用されるため、ここでは冗長性は許容されます。

## B.5.2.2 コア依存特性

“バックエンド依存” のワークロードは、次のメトリックにより “コア依存” として分類できます。

$$\%Core\_Bound = \%Backend\_Bound\_at\_EXE - \%Memory\_Bound$$

“%Memory\_Bound” メトリックについては B.5.2.3 節で説明しています。ワークロードがいったん “コア依存” として識別されると、実行ポートのプレッシャーや FP チェーンによる長いレイテンシーの算術演算など、ターゲットのパフォーマンス・カウンターを介して OOO や実行に関連する問題をドリルダウンすることができます。

## B.5.2.3 メモリー依存特性

メモリー・パイプラインのパフォーマンス問題を特性化する手法は、単純な計算を使用してメモリーストールのペナルティーを推測する傾向があります。通常、特定のキャッシュレベル・アクセスへのミス数は、CPU 仕様ごとにキャッシュレベル向けに事前定義されたレイテンシーを掛けることで、ペナルティーを推測します。これはインオーダー・プロセッサでは問題ないかもしれませんが、高度なアウトオブオーダー・プロセッサでは、メモリーアクセスがオーバーラップし、スケジューラーがレイテンシーを隠匿する傾向があるため、CPU サイクルにおけるメモリーアクセスが過度に評価されることがあります。スケジューラーは、メモリー・アクセス・データを必要としない  $\mu\text{op}$  によって実行ストールをビジーに保つことで、メモリー・アクセス・ストールを隠匿することができます。したがって、メモリーアクセスのペナルティーは、スケジューラーがディスパッチの準備ができていない場合に実行ユニットがスタベーションに陥ることによって生じます。 $\mu\text{op}$  がメモリーアクセスによるデータを待機しているか、ディスパッチされていない  $\mu\text{op}$  と依存関係がある可能性があります。

インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge では、メモリーアクセスの調査に使用できる新しいパフォーマンス・イベント “CYCLE\_ACTIVITY.STALLS\_LDM\_PENDING” が提供されています。

“memory bound (メモリー依存)” メトリックを定義するため、そのイベントを使用します。このイベントは、実行スタベーションを引き起こす完了していない続行中のメモリーロード要求のサイクルを測定します。 $\mu\text{op}$  がストアや HW プリフェッチの完了を (直接) 待機しない場合、ロード要求操作のみをカウントすることに注意してください。

$$\%Memory\_Bound = CYCLE\_ACTIVITY.STALLS\_LDM\_PENDING / CLOCKS$$

ワークロードがメモリー依存である場合、キャッシュ階層と DRAM システムメモリーに関連するパフォーマンス特性をさらに詳しく特性化することができます。

L1 キャッシュのレイテンシーは、一般に、すべてのストールの中で最も短い ALU ユニットのストールに匹敵する短いレイテンシーです。しかし、特定の状況では、先行するストアによってブロックされるロードのように、L1 から提供されるまでロードは高いレイテンシーを被る可能性があります。L1 ヒットにはフィルバッファが割り当てられていません。代わりに、完了しなかったロードをカウントする LDM ストール・サブイベントを使用できます。

$$\%L1\ Bound = (CYCLE\_ACTIVITY.STALLS\_LDM\_PENDING - CYCLE\_ACTIVITY.STALLS\_L1D\_PENDING) / CLOCKS$$

前述したように、L2 Bound (L2 依存) は次のように検出できます。

$$\%L2\ Bound = (CYCLE\_ACTIVITY.STALLS\_L1D\_PENDING - CYCLE\_ACTIVITY.STALLS\_L2\_PENDING) / CLOCKS$$

同様に、L3 ミスを減算することで L3 Bound (L3 依存) を計算できます。しかし、L3\_PENDING を測定する等価なイベントがありません。そのため、補正係数に関連する L3\_HIT と L3\_MISS ロード・カウント・イベントを使用することで概算を推測できます。長いレイテンシーが L3 とメモリーにある場合、この推測は許容されます。補正係数 MEM\_L3\_WEIGHT は、外部メモリーから L3 キャッシュ・レイテンシーへの比率です。第 3 世代インテル® Core™ プロセッサ・ファミリーには、係数 7 を使用します。この補正係数は、CPU とメモリー周波数に依存します。

$$\%L3\ Bound = CYCLE\_ACTIVITY.STALLS\_L2\_PENDING * L3\_Hit\_fraction / CLOCKS$$

ここで、L3\_Hit\_fraction は次のとおりです。

$$MEM\_LOAD\_UOPS\_RETIRED.LLC\_HIT / (MEM\_LOAD\_UOPS\_RETIRED.LLC\_HIT + MEM\_L3\_WEIGHT * MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS)$$

第 3 世代インテル® Core™ プロセッサの DRAM トラフィックを算出するため、L2\_PENDING の残数が MEM Bound に使用されます。

$$\%MEM\ Bound = CYCLE\_ACTIVITY.STALLS\_L2\_PENDING * L3\_Miss\_fraction / CLOCKS$$

ここで、L3\_Miss\_fraction は次のとおりです。

$$WEIGHT * MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS / (MEM\_LOAD\_UOPS\_RETIRED.LLC\_HIT + WEIGHT * MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS)$$

場合によっては、コア外部のすべてのメモリーストールを Uncore Bound (アンコア依存) と呼べるでしょう。

$$\%Uncore\ Bound = CYCLE\_ACTIVITY.STALLS\_L2\_PENDING / CLOCKS$$

### B.5.3 バックエンドのストール

バックエンドのストールの主な原因は、メモリー・サブシステムのストールと実行ストールの 2 つです。最初に、バックエンド・ストールの原因を理解するため、リソースのストール・イベントを使用します。

マイクロオペレーション (μop) をスケジューラーに送る前に、リネームステージでリソースを割り当てる必要があります。パイプラインのバックエンドで重大なボトルネックが発生すると、パイプラインが停滞し、リソースが足りなくなります。RESOURCE\_STALLS イベントは、リソースを割り当てることができなかった場合にストールサイクルを追跡します。このイベントは、割り当てに利用できないリソースを追跡できるように、各リソースを個別のサブイベントに分類します。このイベントをカウントすることで、パイプラインのバックエンドにおける問題の原因特定に役立つ可能性があります。

以下に説明するリソースのストール比率は、CPU\_CLK\_UNHALTED.THREAD によってカウントされるサイクルのプロセス、モジュール、関数、さらに命令単位で計測でき、同じ単位でタグ付けされたペナルティを表します。

#### 固有のイベントの使用

RESOURCE\_STALLS.ANY: リソースが不足しているため、リネームステージがマイクロオペレーション (μop) をスケジューラーに送出できないストールサイクルをカウントします。不足しているリソースは、このステージで割り当てる必要があります。ブロックしている命令のリタイアメントに近い場合、イベントスキッドは短くなる傾向があります。このイベントは、他の RESOURCE\_STALL サブイベントによってカウントされるすべてのストールを示すだけでなく、RESOURCE\_STALLS2 のサブイベントも含みます。この比率が高い場合、サブイベントをカウントすることによって、ストールの原因をより適切に分類できます。

$$\%RESOURCE.STALLS.COST = 100 * RESOURCE\_STALLS.ANY / CPU\_CLK\_UNHALTED.THREAD;$$

RESOURCE\_STALLS.SB: 通常、ストア・マイクロオペレーション (μop) は割り当て可能な状態にあります。レイテンシーが長いストアが進行中であるため、すべてのストア・バッファ・エントリが占有されている場合に発生します。通常、このイベントは、割り当て時にストールするストア命令の後の IP にタグ付けされます。

$$\%RESOURCE.STALLS.SB.COST = 100 * RESOURCE\_STALLS.SB / CPU\_CLK\_UNHALTED.THREAD;$$

RESOURCE\_STALLS.LB: 通常、ロード・マイクロオペレーション (μop) は割り当て可能な状態ですが、レイテンシーが長いロードが進行中であるため、すべてのロード・バッファ・エントリーが占有されているサイクルをカウントします。多くの場合、ロードバッファ一杯になる前に、レイテンシーが長いロードに依存するマイクロオペレーション (μop) によってスケジューラーのキューが一杯になります。

$\%RESOURCE\_STALLS.LB.COST = 100 * RESOURCE\_STALLS.LB / CPU\_CLK\_UNHALTED.THREAD;$

上記のケースでは、イベント RESOURCE\_STALLS.RS は並行してカウントされることがあります。データの局所性における損失をさらに調査するため、B.5.4.2 節で説明する上位キャッシュラインの置換を検討します。最初に L1 D キャッシュの置換に集中します。スケジューラー・スロットは、通常、パイプラインのバックアップ時に最初に消費されるリソースです。しかし、レイテンシーが長いロードや実行ステージでバックアップされる命令など、バックエンドにおけるボトルネックが原因であることもあります。このため、スケジューラー・エントリーの不足にタグ付けされたストールを詳しく調査する前に、他のリソースのストールを調べることを推奨します。このイベントのスキッドは短くなる傾向があります。

$\%RESOURCE\_STALLS.RS.COST = 100 * RESOURCE\_STALLS.RS / CPU\_CLK\_UNHALTED.THREAD;$

RESOURCE\_STALLS.ROB: すべてのリオーダーバッファ (ROB) のエントリーが処理済みであるため割り当てがストールしているサイクルをカウントします。このイベントは、RESOURCE\_STALLS.RS ほど頻繁には発生しません。新しいマイクロオペレーション (μop) は順番にリタイアする必要があるため、通常、より新しいリタイアを保留しているマイクロオペレーション (μop) によってパイプラインがバックアップされていることを示します。

$\%RESOURCE\_STALLS.ROB.COST = 100 * RESOURCE\_STALLS.ROB / CPU\_CLK\_UNHALTED.THREAD;$

RESOURCE\_STALLS2.BOB\_FULL: 分岐マイクロオペレーション (μop) は割り当て可能な状態ですが、プロセッサ内で進行中の分岐の数が上限に達したため割り当てがストールしたときにカウントされます。

$\%RESOURCE\_STALLS.BOB.COST = 100 * RESOURCE\_STALLS2.BOB / CPU\_CLK\_UNHALTED.THREAD;$

## B.5.4 メモリー・サブシステム ストール

以下の項では、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge 固有のパフォーマンス監視イベントを使用したメモリー・サブシステムのストールを特定する方法について説明します。

### B.5.4.1 ロード・レイテンシーのアカウンティング

ロード操作の局所性の分類は、プロセス、モジュール、関数、命令など、任意の粒度で実行できます。ロード操作がボトルネックであることが判明したら、プリサイズロードをブレイクダウンしてさらに詳しく調査します。この方法でボトルネックを分類できないときは、ロードに影響する可能性があるその他の問題をチェックします。

次のイベントを使用すると、ボトルネックとなっているロードのコストを算出し、メモリー階層レベルの比率 (%) をブレイクダウンできます。プリサイズイベントのサンプリングは、すべてのツールでサポートされるわけではありません。これらのイベントの正確 (プリサイズ) なバージョン (イベント名はサフィックス PS で終わります) が利用するツールでサポートされない場合、非プリサイズなバージョンを使用できます。

プリサイズ・ロード・イベントは、このイベントを次のリタイアした命令 (IP+1) にタグ付けします。階層レベルごとのロード・レイテンシーについては、表 2-18 を参照してください。

#### 要求イベント

MEM\_LOAD\_UOPS\_RETIRED.L1\_HIT\_PS: 第 1 レベルのデータキャッシュである L1D キャッシュにヒットしたロード要求をカウントします。ロード要求は、非スペキュレーティブ・ロード・マイクロオペレーション (μop) です。

MEM\_LOAD\_UOPS\_RETIRED.L2\_HIT\_PS: 第 2 レベルのキャッシュである L2 にヒットしたロード要求をカウントします。

MEM\_LOAD\_UOPS\_RETIRED.L3\_HIT\_PS: 第 3 レベルの共有キャッシュである LLC にヒットしたロード要求をカウントします。

MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRE.XSNP\_MISS: 第 3 レベルの共有キャッシュにヒットしたロード要求のうち、他のコアのキャッシュにも存在すると想定され、キャッシュラインがすでに排出されているものをカウントします。

MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRE.XSNP\_HIT\_PS: 他のコアのキャッシュ内のキャッシュラインにヒットしたロード要求のうち、キャッシュラインが更新されていないものをカウントします。

MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRE.XSNP\_HITM\_PS: 他のコアのキャッシュ内のキャッシュラインにヒットしたロード要求のうち、そのコアによってキャッシュラインに書き込まれたものをカウントします。このイベントは、ロックの競合やフォルス・シェアリングなど、マルチスレッド・アプリケーションで発生する可能性があるパフォーマンスのボトルネックを調査する際に重要です。

MEM\_LOAD\_MISC\_RETIRED.LLC\_MISS\_PS: LLC にミスしたロード要求をカウントします。これは、ロードするデータがシステムのメモリーから供給されていることを意味します。

MEM\_LOAD\_UOPS\_RETIRED.HIT\_LFB\_PS: ライン・フィル・バッファ (LFB) にヒットしたロード要求をカウントします。LFB エントリーは、L1D キャッシュ内でミスが発生するたびに割り当てられます。ロードがこのロケーションにヒットした場合、先行するロード、ストア、またはハードウェア・プリフェッチがすでに L1D キャッシュ内でミスし、データフェッチが進行中であることを意味します。したがって、LFB におけるヒットのコストはさまざまです。このイベントは、L1D キャッシュ内ではミスしますが、LLC 内ではミスしないキャッシュライン分割ロードをカウントする可能性があります。

32 バイトのインテル® AVX ロードでは、L1D キャッシュ内でミスしたロードは、すべて L1D キャッシュ内のヒットまたは LFB 内のヒットとして示されます。その他のメモリー階層のレベルでは、ヒット数は示されません。L1D キャッシュ内でインテル® AVX ロードがミスすると、ほとんどのロードはライン・フィル・バッファ (LFB) からデータを取得します。

## プリサイスロードのブレイクダウン

ロードソースごとの比率 (%) の分類では、単一の IP、関数、モジュール、プロセスなど、任意の粒度でタグ付けできます。これは、単一の命令のロードがキャッシュ階層のどこで見つかったブレイクダウンするのに役立ちます。次の数式は、LLC からロードしたデータが供給される時間の割合を計算する方法を示しています。すべての階層レベルについて、同様の数式を使用できます。

$$\%LocL3.HIT = 100 * MEM\_LOAD\_UOPS\_RETIRED.LLC\_HIT\_PS / \$SumOf\_PRECISE\_LOADS;$$

$$\$SumOf\_PRECISE\_LOADS =$$

$$\begin{aligned} & MEM\_LOAD\_UOPS\_RETIRED.HIT\_LFB\_PS + MEM\_LOAD\_UOPS\_RETIRED.L1\_HIT\_PS + \\ & MEM\_LOAD\_UOPS\_RETIRED.L2\_HIT\_PS + MEM\_LOAD\_UOPS\_RETIRED.LLC\_HIT\_PS + \\ & MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRE.XSNP\_MISS + \\ & MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRE.XSNP\_HIT\_PS + \\ & MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRE.XSNP\_HITM\_PS + \\ & MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS\_PS; \end{aligned}$$

## ロードペナルティーの推測

以下の数式は、特定のメモリー階層からのロードがパフォーマンスの低下にどの程度影響しているか推測するのに役立ちます。プログラム可能な CPU\_CLK\_UNHALTED.THREAD イベントは、同じ粒度でタグ付けされたサイクルのペナルティーを表します。プリサイスイベントと同様、命令レベルでは、長いロードのサイクルコストは 1 IP だけ

キッドする傾向があります。以下の計算は、イベントが正確 (プリサイズ) であるため、任意の粒度のプロセス、モジュール、関数、または命令に適用できます。目的の粒度における総クロック数の 10% 以上であれば調査する必要があります。

依存性の高いロードがコードに存在する場合、MEM\_LOAD\_UOPS\_RETIRED.L1\_HIT\_PS イベントを使用して、そのロードが L1D キャッシュの 5 サイクルのレイテンシーでヒットするかどうかを判断できます。

L2 レイテンシーのコスト算出

$$\%L2.COST = 12 * MEM\_LOAD\_UOPS\_RETIRED.L2\_HIT\_PS / CPU\_CLK\_UNHALTED.THREAD;$$

L3 ヒットのコスト算出

$$\%L3.COST = 26 * MEM\_LOAD\_UOPS\_RETIRED.L3\_HIT\_PS / CPU\_CLK\_UNHALTED.THREAD;$$

他のコアのキャッシュへのヒットのコスト算出

$$\%HIT.COST = 43 * MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRED.XSNP\_HIT\_PS / CPU\_CLK\_UNHALTED.THREAD;$$

メモリー・レイテンシーのコスト算出

$$\%MEMORY.COST = 200 * MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS\_PS / CPU\_CLK\_UNHALTED.THREAD;$$

実際のメモリー・レイテンシーは、メモリー・パラメーターによって大きく異なります。平行に発生したメモリー・トラフィックの量によっては、特定のメモリー階層のコストが削減されることがあります。通常、上記の概算値は、(ポインター追跡を行っているような) ワorstケースであると考えられます。

多くの場合、キャッシュミスは命令のリタイアメントまで遅延し、一括して現れます。プリサイズロードをブレイクダウンすることで、ロードに供給されるデータの階層レベルの分布を概算できます。

特定のキャッシュレベルの影響が大きい場合、大量のキャッシュラインの置換がコードのどの部分で発生しているかを見つめます。これは、メモリー階層レベルのブレイクダウンによって検出されたコードのホットな領域と一致することもあります。多くの場合はそうなりません。例えば、大規模なデータ構造の定期的な走査により、キャッシュのレベルが意図せずにクリアされる可能性があります。

変更されていないデータまたは変更されたデータが別のコアでヒットし、その概算コストが高く、かつコード領域が「ホット」である場合、スレッド間のロック、共有、またはフォールス・シェアリングが原因である可能性があります。

L1D キャッシュから離れたメモリー階層レベルのロード・レイテンシーが、ロードにかかったサイクルの数を正当化するものでない場合、次のいずれかを試みます。

Y 汎用レジスターから (メモリーではなく) XMM レジスターへのスピルを招く不要なロード操作を排除します。

Y B.5.4.4 節で説明するロード命令に影響を与える問題を引き続き調査します。

## B.5.4.2 キャッシュラインの置換の分析

アプリケーションで多数のキャッシュミスが発生する場合、どのキャッシュラインが最も頻繁に置き換えられているかを調査するとよいでしょう。置換はハードウェア・プリフェッチとストア操作によって引き起こされますが、大量のキャッシュライン置換に関連している命令とアプリケーションが時間を費やしているコード領域が常に一致するとは限りません。一般に、大きな配列やデータ構造を横断すると、大量のキャッシュライン置換が発生する可能性があります。

## 要求イベント

L1D.REPLACEMENT: 1 次データキャッシュを置換します。

L2\_LINES\_IN.ALL: L2 キャッシュに格納されるキャッシュラインです。

## イベントの使用

パフォーマンスの低下を引き起こす可能性がある置換は、プロセス、モジュール、関数レベルで特定できます。これは、次の 2 つのステップで行います。

- Y プリサイズロードのブレイクダウンにより、ロードが実行され、最大のペナルティーが発生するメモリー階層レベルを特定します。
- Y 以下の数式を使用して、そのレベルで大部分の置換が発生させ、さらに下位のレベルにおいてこれらの大きなペナルティ・ロードに関連するコード領域を特定します。

例えば、ロードが LLC にヒットすることで高いペナルティーが存在する場合、L2 および L1 で置換が発生しているコードをチェックします。以下の数式において、分子はモジュールまたは関数でカウントされた置換数です。分母の置換の総和は、プロセス全体のすべてのキャッシュレベルでの置換の和です。これにより、大量の置換を発生させるコード領域を特定することができます。

### L1D キャッシュ置換

$$\%L1D.REPLACEMENT = L1D.REPLACEMENT / \text{SumOverAllProcesses}(L1D.REPLACEMENT);$$

### L2 キャッシュ置換

$$\%L2.REPLACEMENT = L2\_LINES\_IN.ALL / \text{SumOverAllProcesses}(L2\_LINES\_IN.ALL);$$

## B.5.4.3 ロック競合の分析

マルチスレッド・アプリケーションのスケラビリティ分析では、ロックの競合を正確に特定することが重要です。一般的な ring3 ロックは、ほとんどの場合アトミック命令を実行します。アトミック命令とは、メモリーアドレスを含む XCHG 命令、メモリー・デスティネーションとロック・プリフィクスを含む ADD、ADC、AND、BTC、BTR、BTS、CMPXCHG、CMPXCH8B、DEC、INC、NEG、NOT、OR、SBB、SUB、XOR、または XADD 命令を指します。プリサイズイベントを使用すれば、任意のロック競合について知ることができます。多くのロック API は、ring3 でアトミックな命令を開始し、ring0 にジャンプすることで競合しているロックを迂回します。つまり、競合が少ないシナリオでは、大量のロック API はコストが非常に大きくなる可能性があります。ロックされた命令の競合回数を算出するには、アトミック命令のメモリー・デスティネーションを含むキャッシュラインが別のコア内で変更された回数を測定します。

## 要求イベント

MEM\_UOPS\_RETIRED.LOCK\_LOADS\_PS: IP+1 のプリサイズスキッドでリタイアしたアトミック命令の数をカウントします。

MEM\_LOAD\_UOPS\_LLC\_HIT\_RETIRED.XSNP\_HITM\_PS: 別のコアの変更されたキャッシュラインにヒットしたロードの発生回数をカウントします。このイベントは、ロック競合やフォルス・シェアリングなど、マルチコアシステムで発生する可能性があるパフォーマンスのボトルネックを調査するには重要です。



## イベントの使用

ロック競合の係数は、別のコアと競合するために大きなペナルティーを持つ、ロックによって実行された操作の比率 (%) を示します。通常、ロック競合の係数が 5% を超えている場合、ホットロックを調査することが望ましく、著しいロック競合は、複数スレッドのパフォーマンスに影響する場合があります。

```
%LOCK.CONTENTION =
    100 * MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS / MEM_UOPS_RETIRED.LOCK_LOAD_PS;
```

### B.5.4.4 そのほかのメモリアクセスの問題

#### ストアフォワードのブロック

ストア・フォワーディングが可能でない場合、依存するロードはブロックされます。ストア・フォワーディング・ブロックの平均ペナルティーは 13 サイクルです。多くのストア・フォワーディングのブロックは以前のアーキテクチャーにおいて改善されているため、今日のコードにおける最も典型的な問題は、大きなロードではなく、小さなメモリー空間へのストアに関連しています。

#### 要求イベント

LD\_BLOCKS.STORE\_FORWARD: アーキテクチャーが短いストアを長いロードにフォワードできない、またはまれに発生するアライメントの問題によってストア・フォワーディングがブロックされた回数をカウントします。

#### イベントの使用

ストア・フォワーディング・ブロックのコストを算出するには、次の数式を使用します。イベント LD\_BLOCKS.STORE\_FORWARD は、ロードの後に続く IP にタグ付けされる傾向があるため、この問題は命令レベルで調査することが推奨されます。ただし、プロセス、モジュール、関数、または IP 粒度で比率を調べることはできません。

```
%STORE.FORWARD.BLOCK.COST =
    100 * LD_BLOCKS.STORE_FORWARD * 13 / CPU_CLK_UNHALTED.THREAD;
```

ストア・フォワーディングによりブロックされているロードが検出されたら、ストアの場所を特定する必要があります。一般に、ストア・フォワーディングによりブロックされる問題の 65% は、ロードの直前に実行された 10 命令以内にあるストアによって発生します。

ストア・フォワーディング・ブロックの最も一般的な問題は、長いロードにフォワードできない短いストアです。例えば、生成された次のコードは、バイト・ポインター・アドレスに書き込んでから、4 バイト (dword) のメモリー領域から読み出しています。

```
and byte ptr [ebx], 7f
and dword ptr [ebx], ecx
```

ストア・フォワーディング・ブロックを回避する最善の方法は、ロードではなくストア操作を修正することです。

#### キャッシュライン分割

インテル® マイクロアーキテクチャー開発コード名 Nehalem 以降、L1D キャッシュは分割レジスターを備えているため、2 つのキャッシュラインにまたがるロードおよびストアを処理できるようになりました。これにより、以前のマイクロアーキテクチャーでは 20 サイクルかかりましたが、分割レジスターを利用できる場合、分割ロードのコストは約 5 サイクルに軽減されます。通常、分割ストアは隠蔽されますが、多数の分割ストアがある場合、ストアバッファ

が一杯になり割り当てがストールしたり、分割ロードの処理に必要な分割レジスターが消費される可能性があります。キャッシュラインの分割を排除することにより、明らかな利益が得られます。

## 要求イベント

MEM\_UOPS\_RETIRED.SPLIT\_LOADS\_PS: 2 つのキャッシュラインにまたがるロード要求の数をカウントします。このイベントは正確 (プリサイズ) です。

MEM\_UOPS\_RETIRED.SPLIT\_STORES\_PS: 2 つのキャッシュラインにまたがるストアの数をカウントします。このイベントは正確 (プリサイズ) です。

## イベントの使用

分割ロードは、通常、大部分のコストが実行された次の IP にタグ付けされるため、非常に簡単に検出できます。次の比率は、分割後に任意の粒度 (プロセス、モジュール、関数、IP) で使用できます。

$$\%SPLIT.LOAD.COST = 100 * MEM\_UOPS\_RETIRED.SPLIT\_STORES\_PS * 5 / CPU\_CLK\_UNHALTED.THREAD;$$

通常、ストアは命令のリタイアメントを遅延させないため、コストの概算により分割ストアのペナルティーを検出することは容易ではありません。大量の分割ストアを検出するには、その IP でリタイアしたストアの総数で割ります。

$$SPLIT.STORE.RATIO = MEM\_UOPS\_RETIRED.SPLIT\_STORES\_PS / MEM\_UOPS\_RETIRED.ANY\_STORES\_PS;$$

## 4K エイリアシング

ロードとストア間の 4KB エイリアシングの競合は、ロードの再発行を引き起こします。以下のモデルでは、概算値として 5 サイクルが使用されています。

## 要求イベント

LD\_BLOCKS\_PARTIAL.ADDRESS\_ALIAS: ロードのうち、先行するストアとアドレスの一部が一致しているために再発行された数をカウントします。

## イベントの使用

$$\%4KALIAS.COST = 100 * LD\_BLOCK\_PARTIAL.ADDRESS\_ALIAS * 5 / CPU\_CLK\_UNHALTED.THREAD;$$

## ロードとストアのアドレス変換

リニアアドレスから物理アドレスへ変換するため、2 レベルのトランスレーション・ルックアサイド・バッファー (TLB) があります。第 1 レベルの TLB である DTLB 内のミスは、第 2 レベルの TLB である STLB にヒットし、7 サイクルのペナルティーが発生します。

STLB 内でミスが発生すると、プロセッサは、アドレス変換を含むページテーブルのすべてのエントリーを検索する必要があります。この検索 (ページウォーク) のコストは、ページ・テーブル・エントリーの位置によって異なります。ページウォークの継続期間により、STLB ミスのコストをかなり正確に概算できます。

## 要求イベント

DTLB\_LOAD\_MISSES.STLB\_HIT: DTLB にミスしたロードのうち、STLB にヒットしたロードをカウントします。このイベントはスキッドが短いため、IP レベルで使用できます。

DTLB\_LOAD\_MISSES.WALK\_DURATION: STLB ミスの後に実行されるページウォークの継続期間 (サイクル数) をカウントします。イベントスキッドは通常、1 命令であり、命令、関数、モジュール、またはプロセスの粒度で問題を検出できます。

MEM\_UOPS\_RETIRED.STLB\_MISS\_LOADS\_PS: STLB 内で変換ミスが発生したロードに対するプリサイズイベント。このイベントは、ページのページウォークを開始した最初のロードのみをカウントします。

## イベントの使用

ロードに対する STLB ヒットのコスト:

$$\%STLB.HIT.COST = 100 * DTLB\_LOAD\_MISSES.STLB\_HIT * 7 / CPU\_CLK\_UNHALTED.THREAD;$$

ページウォークのコスト:

$$\%STLB.LOAD.MISS.WALK.COST = 100 * DTLB\_LOAD\_MISSES.WALK\_DURATION / CPU\_CLK\_UNHALTED.THREAD;$$

頻繁に STLB ミスが発生する命令やソース行を正確に把握するには、プリサイズ STLB ミスイベントを IP レベルで使用します。

$$\%STLB.LOAD.MISS = 100 * MEM\_UOPS\_RETIRED.STLB\_MISS\_LOADS\_PS / MEM\_UOPS\_RETIRED.ANY\_LOADS\_PS;$$

ページウォークの継続期間が長い (何百サイクルにもなる) 場合、LLC からページテーブルが排除されている兆候を示します。ページウォークの平均コストを判定するには、次の比率を使用します。

$$STLB.LOAD.MISS.AVGCOST = DTLB\_LOAD\_MISSES.WALK\_DURATION / DTLB\_LOAD\_MISSES.WALK\_COMPLETED;$$

ロードほどではありませんが、ストアの STLB ミスはボトルネックとなる可能性があります。ストアそのものがボトルネックになっている場合、サイクルはストアに続く IP にタグ付けされます。

$$\%STLB.STORE.MISS = 100 * MEM\_UOPS\_RETIRED.STLB\_MISS\_STORES\_PS / MEM\_UOPS\_RETIRED.ANY\_STORES\_PS;$$

DTLB/STLB ミスを減らすことで、データの局所性を高めることができます。したがって、商用グレードのメモリー・アロケータを使用してデータの局所性を改善することを考えます。プロファイルに基づく最適化機能を提供するコンパイラーでは、モジュール全体を操作できる場合、グローバル変数をリオーダーすることでデータの局所性が改善される可能性があります。ページウォークに長い時間がかかる場合、サーバー・アプリケーションや HPC アプリケーションでは、ラージページの利用を検討してください。

## B.5.5 実行ストール

以下の項では、インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge 固有のパフォーマンス監視イベントを使用したアウトオブオーダー・エンジンのストールを特定する方法について説明します。

### B.5.5.1 長い命令レイテンシー

マイクロアーキテクチャーの変更によって、既存のコード内の一部のレガシー命令のレイテンシーが長くなる場合があります。このような状況のいくつかは検出できます。

Y 3 つのオペランドを持つ低速の LEA 命令 (3.5.1.2 節を参照)

Y フラグとマイクロオペレーション (μop) のマージ: 「shift cl」命令で必要となります (3.5.2.6 節を参照)

パイプラインの初めにイベント生成条件が検出されるため、通常、サブイベントのスキッドが最大で 10 命令になる傾向があります。

## イベントの使用

このイベントを、命令レベルではなく、プロセス、モジュール、関数の粒度で使用すると、イベントスキッドを気にせず、効果的にパフォーマンスの問題を特定できます。命令 IP の粒度で問題を特定するには、イベントで特定された関数に対して静的分析を行います。これらのイベントのコード・レイテンシーへの影響を評価するには、同じ粒度のサイクル数で割ります。全体的な影響を概算するには、これらの問題による合計サイクル数から開始し、影響が大きい場合は、サブイベントを使用して正確な原因を引き続き調査します。

### 特定のシナリオでかかった合計サイクル数

フラフマージの比率:

$$\%FLAGS.MERGE.UOP = 100 * PARTIAL\_RAT\_STALLS.FLAGS\_MERGE\_UOP\_CYCLES / CPU\_CLK\_UNHALTED.THREAD;$$

低速の LEA 命令の割り当て:

$$\%SLOW.LEA.WINDOW = 100 * PARTIAL\_RAT\_STALLS.SLOW\_LEA\_WINDOW / CPU\_CLK\_UNHALTED.THREAD;$$

## B.5.5.2 アシスト

アシストは、通常、アシスト処理を支援するマイクロコード・シーケンサーを起動します。マイクロコード・シーケンサーによって生成されたマイクロコードのサイクル数を判定することは、多くの場合、アシストの総コストを判定する適切な方法です。アシストの総コストが高い場合、アシストを詳細なタイプに分類すると効果的です。

マイクロコード・シーケンサーのサイクル数を使用したアシストの総コストを算出:

$$\%ASSISTS.COST = 100 * IDQ.MS\_CYCLES / CPU\_CLK\_UNHALTED.THREAD;$$

### 浮動小数点アシスト

x87 命令のデノーマル入力は FP アシストを必要とするため、何百サイクルのコストが生じる可能性があります。

$$\%FP.ASSISTS = 100 * FP\_ASSIST.ANY / INST\_RETIRED.ANY;$$

### インテル® SSE とインテル® AVX 間の遷移

インテル® SSE コードとインテル® AVX コード間の遷移については、12.3.1 節で詳しく説明しています。標準的なコストはおおよそ 75 サイクルです。

$$\%AVX2SSE.TRANSITION.COST = 75 * OTHER\_ASSISTS.AVX\_TO\_SSE / CPU\_CLK\_UNHALTED.THREAD;$$

$$\%SSE2AVX.TRANSITION.COST = 75 * OTHER\_ASSISTS.SSE\_TO\_AVX / CPU\_CLK\_UNHALTED.THREAD;$$

2 ページにまたがる 32 バイトのインテル® AVX ストア命令は、およそ 150 サイクルのコストを生じるアシストを必要とします。32 バイトのインテル® AVX ストアの後に続く IP にタグ付けされた大量のマイクロコードは、アシストが発生したことを示唆します。

```
%AVX.STORE.ASSIST.COST = 150 * OTHER_ASSISTS.AVX_STORE / CPU_CLK_UNHALTED.THREAD;
```

## B.5.6 投機の問題

この節では、パイプラインのフラッシュ招く、分岐命令の予測ミスについて説明します。

### B.5.6.1 分岐予測ミス

分岐予測ミスの最大の課題は、予測ミスが発生した分岐を特定することです。分岐予測ミスには、約 20 サイクルのペナルティーが課せられます。コストは、予測ミスによって、およびデコード済み命令キャッシュ内またはレガシー・デコード・パイプライン内で正しいパスが見つかるかどうかによって異なります。

#### 要求イベント

R\_MISP\_RETIRE.ALL\_BRANCHES\_PS は、分岐ターゲットを不適切に予測した分岐をカウントするプリサイズイベントです。これは、次の命令にスキッドするプリサイズイベントであるため、分岐予測ミスの後の正しいパスの最初の命令にタグ付けされます。このイベントは、プロセス、モジュール、関数、または命令の粒度で適用できます。

#### イベントの使用

分岐予測ミスのコストを算出するには、次の数式を使用します。

```
%BR.MISP.COST = 20 * BR_MISP_RETIRE.ALL_BRANCHES_PS / CPU_CLK_UNHALTED.THREAD;
```

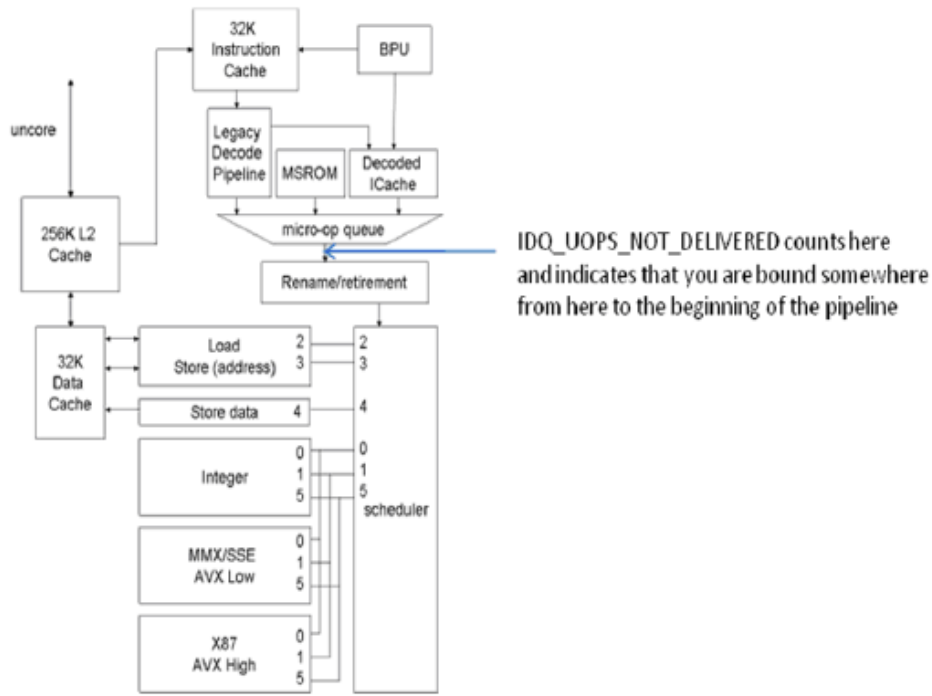
## B.5.7 フロントエンドのストール

フロントエンドのストールについては、B.5.2 節で説明した解析の結果が、30% 以上フロントエンドに関連していない限り詳しく調査する必要はありません。この節では、パイプラインのフロントエンドで遅延を発生させる可能性がある問題について説明します。フロントエンドで検出されたイベントは、スキッドを予測できません。そのため、ペナルティーを IP レベルで関連付けません。このようなイベントは、関数レベル、モジュールレベル、プロセスレベルで調査します。

### B.5.7.1 マイクロオペレーション (μop) の供給比率を理解する

#### カウンターの使用

イベント IDQ\_UOPS\_NOT\_DELIVERED は、マイクロオペレーション (μop) を要求している間に、最大 4 つのマイクロオペレーション (μop) がリネームステージに提供されなかったときにカウントされます。パイプラインが停滞すると、リネームステージは、それ以降のマイクロオペレーション (μop) をフロントエンドに要求しません。次の図は、このイベントがマイクロオペレーション (μop) キューとリネームステージにあるマイクロオペレーション (μop) を追跡する仕組みを示します。



0、1、2、3 のマイクロオペレーション (μop) がフロントエンドから供給されている場合、IDQ\_UOPS\_NOT\_DELIVERED イベントによりサイクルの分布を分類できます。

フロントエンドが有効であるか、実行がバックエンド依存であるサイクル数の比率 (%):

$$\%FE.DELIVERING = 100 * ( CPU\_CLK\_UNHALTED.THREAD - IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_LE\_3\_UOP\_DELIV.CORE ) / CPU\_CLK\_UNHALTED.THREAD;$$

フロントエンドが 1 サイクルあたり 3 個のマイクロオペレーション (μop) を供給しているサイクル数の比率 (%):

$$\%FE.DELIVER.3UOPS = 100 * ( IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_LE\_3\_UOP\_DELIV.CORE - IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_LE\_2\_UOP\_DELIV.CORE ) / CPU\_CLK\_UNHALTED.THREAD;$$

フロントエンドが 1 サイクルあたり 2 個のマイクロオペレーション (μop) を供給しているサイクル数の比率 (%):

$$\%FE.DELIVER.2UOPS = 100 * ( IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_LE\_2\_UOP\_DELIV.CORE - IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_LE\_1\_UOP\_DELIV.CORE ) / CPU\_CLK\_UNHALTED.THREAD;$$

フロントエンドが 1 サイクルあたり 1 個のマイクロオペレーション (μop) を供給しているサイクル数の比率 (%):

$$\%FE.DELIVER.1UOPS = 100 * ( IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_LE\_1\_UOP\_DELIV.CORE - IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_0\_UOPS\_DELIV.CORE ) / CPU\_CLK\_UNHALTED.THREAD;$$

フロントエンドが 1 サイクルあたり 0 個のマイクロオペレーション (μOP) を供給しているサイクル数の比率 (%):

$$\%FE.DELIVER.0UOPS = 100 * ( IDQ\_UOPS\_NOT\_DELIVERED.CYCLES\_0\_UOPS\_DELIV.CORE ) / CPU\_CLK\_UNHALTED.THREAD;$$

1 サイクルあたりに供給されるマイクロオペレーション (μop) の平均数: この比率は、バックエンド依存である場合に、フロントエンドが 1 サイクルあたり 4 個のマイクロオペレーション (μop) を供給できることを前提としています。

$$\text{AVG.uops.per.cycle} = (4 * (\%FE.DELIVERING) + 3 * (\%FE.DELIVER.3UOPS) + 2 * (\%FE.DELIVER.2UOPS) + (\%FE.DELIVER.1UOPS)) / 100$$

1 サイクルあたりに供給されるマイクロオペレーション (μop) の分布を確認することで、発生する可能性があるフロントエンドのボトルネックに関するヒントが得られます。デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えによる LCP やペナルティーなどの問題があると、複数のサイクルにわたってマイクロオペレーション (μop) が供給されない傾向があります。フェッチ帯域幅の問題やデコーダーのストールが発生すると、1 サイクルあたりに供給されるマイクロオペレーション (μop) が 4 個未満になります。

### B.5.7.2 マイクロオペレーション (μop) キューのソースを理解する

マイクロオペレーション (μop) キューは、次のソースからマイクロオペレーション (μop) を取得します。

- Y デコード済み命令キャッシュ
- Y レガシー・デコード・パイプライン
- Y マイクロコード・シーケンサー (MS)

標準的な分布は、デコード済み命令キャッシュから供給されるマイクロオペレーション (μop) が約 80%、レガシー・デコード・パイプラインからの供給が 15%、マイクロコード・シーケンサーからの供給が 5% です。レガシー・デコード・パイプラインから過度なマイクロオペレーション (μop) が供給されると、デコード済み命令キャッシュが効果的に機能していないという警告される可能性があります。マイクロコード・シーケンサーから供給されるマイクロオペレーション (μop) の大部分 (複雑な命令や文字列操作など) は影響しないかもしれませんが、コードアシスト処理が望ましくない状況 (インテル® SSE コードからインテル® AVX コードへの遷移など) も考えられます。

#### 必要なカウンターの説明

IDQ.DSB\_UOPS: デコード済み命令キャッシュからマイクロオペレーション (μop) キューに供給されたマイクロオペレーション (μop)

IDQ.MITE\_UOPS: レガシー・デコード・パイプラインからマイクロオペレーション (μop) キューに供給されたマイクロオペレーション (μop)

IDQ.MS\_UOPS: マイクロコード・シーケンサーから供給されたマイクロオペレーション (μop)

#### カウンターの使用

デコード済み命令キャッシュから供給されるマイクロオペレーション (μop) の比率 (%):

$$\%UOPS.DSB = \text{IDQ.DSB\_UOPS} / \text{ALL\_IDQ\_UOPS};$$

レガシー・デコード・パイプラインから供給されるマイクロオペレーション (μop) の比率 (%):

$$\%UOPS.MITE = \text{IDQ.MITE\_UOPS} / \text{ALL\_IDQ\_UOPS};$$

マイクロコード・シーケンサーから供給されるマイクロオペレーション (μop) の率 (%):

$$\%UOPS.MS = \text{IDQ.MS\_UOPS} / \text{ALL\_IDQ\_UOPS};$$

ALL\_IDQ\_UOPS = (IDQ.DSB\_UOPS + IDQ.MITE\_UOPS + IDQ.MS\_UOPS);

アプリケーションがフロントエンド依存ではない場合、マイクロオペレーション (μop) がレガシー・デコード・パイプラインから供給されているか、デコード済み命令キャッシュから供給されているかはあまり重要ではありません。マイクロコード・シーケンサーから過度のマイクロオペレーション (μop) が供給されている場合、アシストが問題であるかどうかを調査することが推奨されます。

調査が必要になるケースを以下に示します:

- Y (%FE\_BOUND > 30%) と (%UOPS.DSB < 70%)
  - "フロントエンド依存である" ケースを定義するしきい値として 30% を適用します。このしきい値はほとんどの状況に適用できますが、ワークロードにより少し異なることがあります。
  - デコード済み命令キャッシュからマイクロオペレーション (μop) が供給されない理由を調査する。
  - レガシー・デコード・パイプラインに影響する可能性がある問題を調査する。
- Y (%FE\_BOUND > 30%) と (%UOP\_DSB > 70%)
  - 小さすぎて効果的でないコード領域が実行されている可能性があるため、デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えを調査します。
  - 分岐の予測ミスは FE のパフォーマンスに影響を与えるため、バッド・スペキュレーション数を調べます。
  - 32 バイト・チャンクのヒットごとに供給されるマイクロオペレーション (μop) の平均数を判定します。32 バイト・チャンク間で供給される処理済み分岐が多い場合、1 サイクルごとに供給されるマイクロオペレーション (μop) に影響します。
  - デコード済み命令キャッシュからのマイクロオペレーション (μop) の供給が問題である可能性もありますが、ここでは取り上げません。
- Y (%FE\_BOUND < 20%) と (%UOPS\_MS > 25%)
  - "フロントエンド依存ではない" ケースを定義するしきい値として 20% を適用します。このしきい値はほとんどの状況に適用できますが、ワークロードにより少し異なることがあります。

次の手順は、マイクロコード・シーケンサーからマイクロオペレーション (μop) が供給された原因を判別する上で役立ちます。使用頻度の高い順に示します。

- レイテンシーが長い命令 - 4 マイクロオペレーション (μop) を超えるすべての命令は、マイクロコード・シーケンサーを起動します。一部の命令 (超越関数など) は、マイクロコードから多数のマイクロオペレーション (μop) が生成されます。
- 文字列操作 - 文字列操作は、大量のマイクロコードを生成する可能性があります。状況によっては、トリップカウントが 3 を超える文字列操作 (REP MOVSB など) により、70 サイクルを超えるコストのアシストが発生します。
- アシスト - B.5.5.2 節を参照してください。

### B.5.7.3 デコード済み命令キャッシュ

デコード済み命令キャッシュには、レガシー・デコード・パイプラインと比べて多くの利点があります。複数のマイクロオペレーション (μop) にデコードされる命令やレンジ変更プリフィクス (LCP) のストールなど、レガシー・デコード・パイプラインの多数のボトルネックが排除されます。

デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えは、デコード済み命令キャッシュ内でルックアップが失敗した場合にのみ発生し、通常、パイプラインのフロントエンドで 0 ~ 3 サイクルのコストが生じます。

#### 要求イベント

デコード済み命令キャッシュイベントはすべて長いスキッドを持ち、通常、イベントがタグ付けされた命令は問題の原因ではありません。したがって、この問題は、プロセス、モジュール、関数の粒度でのみ適用されます。



DSB2MITE\_SWITCHES.PENALTY\_CYCLES: デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えに起因するサイクルをカウントします。ただし、バックエンド依存のため、マイクロオペレーション (μop) キューがマイクロオペレーション (μop) を受け入れることができない場合を除きます。

DSB2MITE\_SWITCHES.COUNT: デコード済み命令キャッシュとレガシー・デコード・パイプライン間の切り替えの数をカウントします。

DSB\_FILL.ALL\_CANCEL: デコード済み命令キャッシュへのフィルが取り消されたときにカウントされます。

DSB\_FILL.EXCEED\_DSB\_LINES: 32 バイト・チャンクに対して、デコード済み命令キャッシュに割り当てられたライン数が 3 を超えたためにフィルが取り消されたときにカウントされます。

## イベントの使用

ここで検討する内容はフロントエンド・イベントに関連しているため、特定の命令にイベントをタグ付けしないようにします。

デコード済み命令キャッシュからレガシー・デコード・パイプラインへの切り替えのコストを判定:

```
%DSB2MITE.SWITCH.COST =
    100 * DSB2MITE_SWITCHES.PENALTY_CYCLES / CPU_CLK_UNHALTED.THREAD;
```

デコード済み命令キャッシュからレガシー・フロントエンドへの切り替えあたりの平均コストを判定:

```
AVG.DSB2MITE.SWITCH.COST =
    DSB2MITE_SWITCHES.PENALTY_CYCLES / DSB2MITE_SWITCHES.COUNT;
```

## デコード済み命令キャッシュにおけるミスの原因を判定

デコード済み命令キャッシュ内ではパーシャルヒットはありません。32 バイト・チャンクのルックアップの一部となっているマイクロオペレーション (μop) のいずれかが見つからない場合は、そのトランザクションに対するすべてのマイクロオペレーション (μop) で、デコード済み命令キャッシュミスが発生します。

デコード済み命令キャッシュ内でマイクロオペレーション (μop) が見つからない主な理由は 3 つあります。

- Y 32 バイトのコードチャンクの一部が、3 ウェイ方式のデコード済み命令キャッシュに収まりません。
- Y デコード済み命令キャッシュに対して、頻繁に実行されるコードセクションが大きすぎます。クライアント・アプリケーションは「ホット」な小規模のコードセットを含む傾向があるため、これは、サーバー・アプリケーションに当てはまります。
- Y デコード済み命令キャッシュがフラッシュされています (ITLB のエントリーが排出された場合など)。

32 バイト・コードの一部がデコード済み命令キャッシュの 3 ウェイラインに収まらない場合、プロセス、モジュール、関数、または命令の粒度で DSB\_FILL.EXCEED\_DSB\_LINES イベントを使用します。

```
%DSB.EXCEED.WAY.LIMIT = 100 * DSB_FILL.EXCEED_DSB_LINES / DSB_FILL.ALL_CANCEL;
```

## B.5.7.4 レガシー・デコード・パイプラインにおける問題

マイクロオペレーション (μop) キューに入るマイクロオペレーション (μop) の大部分がレガシー・デコード・パイプラインから供給される場合、そのステージに影響するボトルネックがあるかどうかをチェックする必要があります。レガシー・デコード・パイプラインにおける最も一般的なボトルネックは次のとおりです。

- Y 十分な命令を提供しないフェッチ  
これは、ホットなコードが完全にアライメントされていない場合に発生します。例えば、実行のためにフェッチしているホットコードが 15 番目のバイトにある場合、1 バイトだけがフェッチされます。
- Y レングス変更プリフィクスが命令長デコーダー内でストールする場合  
2 ~ 4 個のマイクロオペレーション (μop) にデコードされた命令では、デコーダーのスルーポットでバブルが発生します。これは、デコーダーの直前にある命令キューが一杯になると、これらの命令によってペナルティーが生じる可能性があることを意味します。

`%ILD.STALL.COST = 100 * ILD_STALL.LCP * 3 / CPU_CLK_UNHALTED.THREAD;`

### B.5.7.5 命令キャッシュ

大きなホット・コード・セクションを含むアプリケーションでは、命令キャッシュおよび ITLB に関する問題が発生する傾向があります。これは、サーバー・アプリケーションでより多く見られます。

#### 要求イベント

ICACHE.MISSES: 命令キャッシュをミスした命令バイトフェッチの数をカウントします。

#### イベントの使用

命令キャッシュミスが問題に影響しているかどうかを判定するには、同じ粒度 (プロセス、モデル、または関数) を使用して、リタイアした命令イベントのカウント数と比較します。リタイアした命令イベントのカウントの 1% を超える場合は、重大な問題であると考えられます。

`ICACHE.PER.INST.RET = ICACHE.MISSES / INST_RETIRED.ANY;`

命令キャッシュミスが重大な問題を招いている場合、プロファイルに基づく最適化を使用して、ホット・コード・セクションのサイズを減らしてみます。ほとんどのコンパイラーには、テキストの並べ替えオプションが用意されています。これは、ページ数および (それほど多くありませんが) アプリケーションがターゲットとするページ数を減らすうえで役立ちます。

アプリケーションがマクロを多用している場合、関数に変換するか、インテリジェント・リンクを使用して繰り返し実行されるコードを排除します。

## B.6 インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサのパフォーマンス・イベントの使用

インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサのマイクロアーキテクチャーには、固有のパフォーマンス・イベントがあります。『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章を参照してください。

### B.6.1 パフォーマンス・カウンターの結果を理解する

各パフォーマンス・イベントは、コアがアクティブなときに、コアで発生したマイクロアーキテクチャー条件を検出します。コアは、以下の場合にアクティブとなります。

- Y コードを実行している (HALT 命令は除く)。
- Y 他方のコアまたはプラットフォーム上の論理プロセッサによってスヌープされている。これはコアが HALT 状態の場合にも発生します。

一部のマイクロアーキテクチャーの条件は、複数のコアによって共有されるサブシステムに適用されます。また、一部のパフォーマンス・イベントでは、イベントマスク (またはユニットマスク) による物理プロセッサ境界またはバス・エージェント境界での条件を可能にします。

一部のイベントでは、物理プロセッサ上の全コアでのカウントではなく、特定のコアに関連したマイクロアーキテクチャー条件のカウントが可能です (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章 に記載されている L2 およびバス関連イベントを参照してください)。

マルチスレッド・ワークロードがすべてのコアを連続して使用しない場合、コア固有の条件をカウントするパフォーマンス・カウンターは、HALT したコア上である程度進行してから、カウントを中止します。または、ユニットマスクを使用すると、いずれかのプロセッサ・コアに起因する条件の発生を引き続きカウントすることもできます。通常は、ユニット・マスク・フィールドの最上位 2 ビット (IA32\_PERFEVTSELx MSR のビット 15:14) を調整することによって、このような非対称な事象を識別できます (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 17 章を参照)。

HALT したコア上では、そのコアがスヌープされても進行しない 3 つのサイクル・カウント・イベントがあります。それは、Unhalted コアサイクル、Unhalted 参照サイクル、Unhalted バスサイクルです。この 3 つのイベントはすべて、イベント 3CH によって選択されたユニットで検出されます。

一部のイベントは、マイクロアーキテクチャー条件は検出できますが、開始したコアや物理プロセッサを識別する能力は限定されます。例えば、ユニットマスク 20H を使って bus\_drdy\_clocks をプログラムすると、バス上のすべてのエージェントを含めることができます。この場合、各コアのパフォーマンス・カウンターはほぼ同じ値を報告します。カウントを解釈するパフォーマンス・ツールでは、バス・アクティビティーは 1 つのコア (各コアの合計は使用しない) からのイベントカウントと同一になる必要があることを、考慮しなければなりません。

上記は、11B でイベントマスク内のコア固有のサブフィールド (IA32\_PERFEVTSELx MSR のビット 15:14) をプログラムした場合にも当てはまります。各コアのパフォーマンス・カウンターによって報告される結果は、ほぼ同じになります。

## B.6.2 比率の解釈

ワークロードのさまざまな特性を解析するには、2 つのイベントの比率が有用です。その比率は、次のように複数の粒度で取得できます: (1) アプリケーション・スレッドごと、(2) 論理プロセッサごと、(3) コアごと、(4) 物理プロセッサごと。

ソフトウェア開発の観点からは (1) の比率が最も有用ですが、マルチスレッド・アプリケーション上でアプリケーション・スレッドごとにプロセッサ・アフィニティーを明示的に管理する必要があります。その他は、ハードウェア使用率に関する情報を提供します。

通常、一度の実行での測定結果 (比率にかかわるすべてのイベントの測定結果) を収集します。その理由は以下のとおりです。

- Y マルチスレッド・ワークロードの比率を測定する場合、一度の実行で全イベントの結果を取得すると、各スレッドにどのイベントカウンター値が属しているか把握できます。
- Y ライトバックなど一部のイベントは、異なる実行では動作に決定論性がない可能性があります。そのような場合、一度の実行で収集された測定結果のみが有効な比率となります。

## B.6.3 特定のイベントに関する注意事項

この節では、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章に記載されているパフォーマンス・イベントを解釈する際の、イベント固有の注意事項について説明します。

- Y **L2\_Reject\_Cycles, イベント番号 30H** — このイベントは、2 次キャッシュが新しいアクセス要求を拒否した間のサイクルをカウントします。
- Y **L2\_No\_Request\_Cycles, イベント番号 32H** — このイベントは、1 次キャッシュから要求、または 2 次キャッシュへのプリフェッチが発行されていなかったサイクルをカウントします。
- Y **Unhalted\_Core\_Cycles, イベント番号 3C, ユニットマスク 00H** — このイベントは、アクティブなコアによって認識された最小時間単位をカウントします。

ほとんどのオペレーティング・システムでは、HLT 命令によってアイドルタスクが実装されています。その場合、アイドルタスクのクロック数はカウントされません。拡張版 Intel SpeedStep® テクノロジーによる遷移によって、コアの動作周波数が変化することがあります。そのため、このイベントを利用して一定時間サンプリングを行うと、結果に差が生じる可能性があります。

- Y **Unhalted\_Ref\_Cycles, イベント番号 3C, ユニットマスク 01H** — このイベントは、カウントされるサイクルごとの間隔が一定になることが保証されます。具体的には、コアがアクティブなときにバス・クロック・サイクルに合わせてカウントされます。サイクルは、コアクロック周波数をセットするバス比率を掛けることによってコア・クロック・ドメインに変換できます。
- Y **Serial\_Execution\_Cycles, イベント番号 3C, ユニットマスク 02H** — このイベントは、一方のコアがコードをアクティブに実行し(非 HALT 状態)、物理プロセッサの他方のコアが停止しているバスサイクルをカウントします。
- Y **L1\_Pref\_Req, イベント番号 4FH, ユニットマスク 00H** — このイベントは、データ・キャッシュ・ユニット (DCU) の 2 次キャッシュからデータ・キャッシュラインをプリフェッチする要求回数をカウントします。2 次キャッシュがビジーの場合、要求が拒否されることがあります。拒否された要求は再送信されます。
- Y **DCU\_Snoop\_to\_Share, イベント番号 78H, ユニットマスク 01H** — このイベントは、他方のコアが必要とするキャッシュラインを DCU がスヌープした回数をカウントします。キャッシュラインが L1 命令キャッシュまたは他のコアのデータキャッシュ内に存在しないか、または他方のコアが書き込みを必要としているにもかかわらず読み出し専用で設定されています。このようなスヌープは、DCU ストアポートを介して行われます。DCU スヌープが頻繁に発生すると、DCU へのストアと競合するため、ストアのレイテンシーが増加したり、パフォーマンスが影響を受ける可能性があります。
- Y **Bus\_Not\_In\_Use, イベント番号 7DH, ユニットマスク 00H** — このイベントは、バスの完了を待機しているトランザクションがないコアのバスサイクル数をカウントします。
- Y **Bus\_Snoops, イベント番号 77H, ユニットマスク 00H** — このイベントは、バス上で検出された外部スヌープに対する CLEAN、HIT、または HITM 応答の数をカウントします。

シングルプロセッサ・システムでは、CLEAN 応答と HIT 応答が発生することはありません。マルチプロセッサ・システムでは、このイベントは、一方のプロセッサで L2 ミスが発生し、ミスしたデータを他方のプロセッサで発見できなかったことを示します。

シングルプロセッサ・システムでは、HITM 応答は、L1 ミス (命令またはデータ) が発生し、ミスしたキャッシュラインが変更状態にある他のコアで見つかったことを示します。マルチプロセッサ・システムでも、このイベントは、L1 ミス (命令またはデータ) が発生し、ミスしたキャッシュラインが変更状態にある他のコアで見つかったことを示します。

## B.7 パフォーマンス分析のドリルダウン手法

ソフトウェア・パフォーマンスは、アプリケーション・コードとプロセッサのマイクロアーキテクチャーの特性が密接に関連します。これらの相互の影響は、パフォーマンス監視イベントによって推測できます。各マイクロアーキテクチャーは、それぞれのサブシステムを対象とする多数のパフォーマンス・イベントを備えています。主要なパフォーマンス・イベントを選択する手法を使用すると、パフォーマンスのボトルネックをよく理解し、コード・チューニング作業の効率を向上できます。

最近のインテル® 64 および IA-32 プロセッサは、アウトオブオーダー実行エンジンを使用するマイクロアーキテクチャーを採用しています。さらに、プログラムを順に実行するインオーダーのフロントエンドとリタイアメント・ロジックも含まれます。これらのプロセッサはスーパースケaler・ハードウェア、バッファリング、投機実行を利用す

そのため、パフォーマンス・イベントとソフトウェアから認識できるパフォーマンスのボトルネックの解釈が複雑になります。

この節では、パフォーマンス・イベントを使用して、パフォーマンスのボトルネックが存在する領域を詳細に分析する手法について説明します。パフォーマンス・イベントを絞り込むことで、インテル® VTune™ Amplifier を使用して、パフォーマンスのボトルネックとソースコード上の位置を関連付け、3 章 ~ 9 章で説明したコーディングの推奨事項を適用できます。この手法の一般原則はさまざまなマイクロアーキテクチャーに適用できますが、この節では分かりやすいように、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサで利用可能なパフォーマンス・イベントを使用します。

パフォーマンス・チューニングは通常、厳密に定義されたワークロードの完了にかかる時間を減らすことに焦点を置きます。パフォーマンス・イベントを使用して、ワークロードの開始から終了までの経過時間を測定できます。したがって、プロセッサ・サイクルの計測値が小さくなれば、ワークロード完了までの経過時間が短縮されたこととなります。

このドリルダウン手法では、4 フェーズのパフォーマンス・イベントを測定することによって、主要なパイプライン・ステージまたはマイクロアーキテクチャー・サブシステムとコードの相互作用の特性を評価します。図 B-16 は、パフォーマンス・イベントのドリルダウン手法とソフトウェア・チューニングによるフィードバック・ループの関係を示しています。

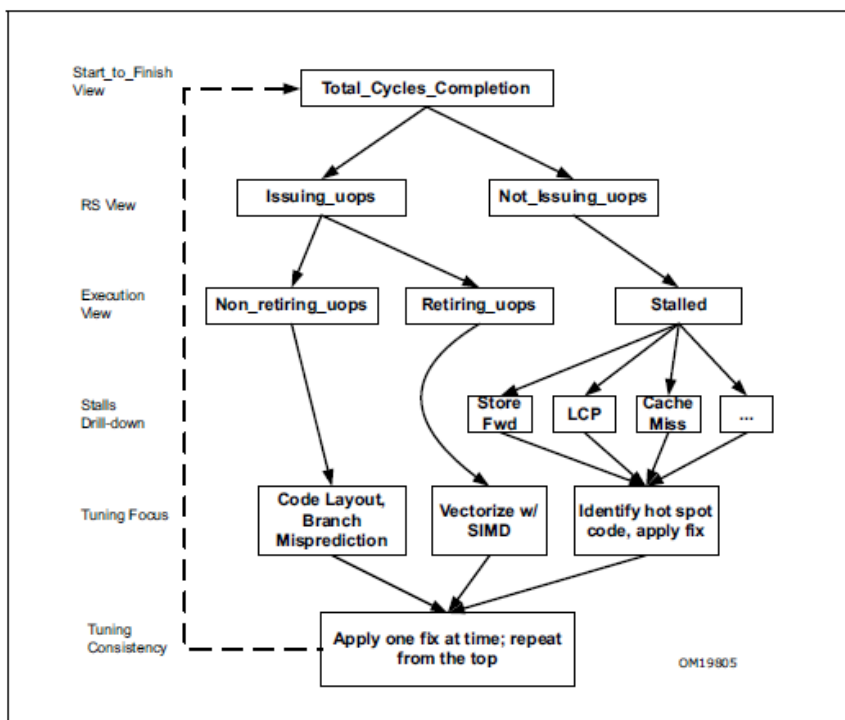


図 B-16 パフォーマンス・イベントのドリルダウンとソフトウェア・チューニングのフィードバック・ループ

通常、パフォーマンス監視ハードウェアのロジックは、サイクル、マイクロオペレーション (μop)、アドレス参照、インスタンスなどのドメインごとに異なるマイクロアーキテクチャー状態を測定します。このドリルダウン手法では、以下に説明する近似的手法によって、各フェーズで直感的に理解できるサイクルベースのビューを得られるようにします。

- Y **総サイクル数の測定** — 総サイクル数は、指定したアプリケーションの開始から終了までに費やされたサイクル数の合計です。一般的なパフォーマンス・チューニングでは、Total\_cycles (総サイクル数) メトリックは CPU\_CLK\_UNHALTED.CORE で測定できます。詳細は、『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 19 章「Performance Monitoring Events」に示されます。
- Y **発行ポートでのサイクル構成** — リザーベーション・ステーション (RS) は、プログラムが処理を進行できるように、実行されるマイクロオペレーション (μop) をディスパッチします。したがって、メトリック Total\_cycles は、2 つの排他的な構成要素である Cycles\_not\_issuing\_uops と Cycles\_issuing\_uops に分類されます。Cycles\_not\_issuing\_uops は、RS が実行されるマイクロオペレーション (μop) を発行していないサイクル数

を示し、Cycles\_issuing\_uops は、RS が実行されるマイクロオペレーション (μOP) を発行しているサイクル数を示します。Cycles\_issuing\_uops のサイクルには、アーキテクチャー上のコードパス、またはスペキュレーティブなコードパスのマイクロオペレーション (μop) が含まれます。

- Y **アウトオブオーダー(OOO)実行のサイクル構成** — アウトオブオーダー・エンジンは、複数のマイクロオペレーション (μop) を並行して実行できる複数の実行ユニットを備えています。1 つの実行ユニットがストールしても、プログラムの実行がストールするとは限りません。このドリルダウン手法では、プログラム実行の進行を近似的に示すサイクル構成ビューを作成します。関連するメトリックは、Cycles\_stalled、Cycles\_not\_retiring\_uops、Cycles\_retiring\_uops の 3 つです。
- Y **実行ストールの分析** — プログラム全体の実行サイクル構成から、パフォーマンス・イベントの選択を絞り込み、ワークロードとマイクロアーキテクチャー・サブシステム間の非生産的な相互作用をピンポイントで識別できます。

マイクロアーキテクチャー・サブシステムのストールまたは非効率なスペキュレーティブ実行によるサイクルの損失が特定できたら、インテル® VTune™ Amplifier を使用して、パフォーマンスに影響する要因とソースコードの位置を関連付けることができます。ストールや予測ミスによるパフォーマンスへの影響が小さい場合でも、インテル® VTune™ Amplifier はソース内のホットスポット関数の位置を特定できるため、プログラマーはこれらの関数のベクトル化によって得られるメリットを評価できます。

## B.7.1 発行ポートでのサイクル構成

最近のプロセッサのマイクロアーキテクチャーは、フロントエンドでプログラム命令をマイクロオペレーション (μop) にデコードしながら、マイクロオペレーション (μop) のストリームをネイティブで実行するアウトオブオーダー・エンジンを採用しています。メトリック Total\_cycles だけでは、プログラムの実行の生産的なサイクルと非生産的なサイクルの分類については不明瞭です。一貫性のあるサイクルベースの分類が行えるように、以下の 2 つのメトリックが用意されています。これらのメトリックは、インテル® Core™ マイクロアーキテクチャー・ベースのプロセッサで利用可能なパフォーマンス・イベントで測定できます。そのイベントとは次の 2 つです。

- Y **Cycles\_not\_issuing\_uops** — このメトリックは、イベント RS\_UOPS\_DISPATCHED によって測定できます。対象とするパフォーマンス・イベント選択 (IA32\_PERFVSELx) MSR 内で INV ビットをセットし、カウンターマスク (CMASK) の値を 1 に設定します (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章を参照)。インテル® VTune™ Amplifier には、CMASK と INV がそれぞれ 1 に設定された、イベント RS\_UOPS\_DISPATCHED.CYCLES\_NONE が用意されています。
- Y **Cycles\_issuing\_uops** — このメトリックは、イベント RS\_UOPS\_DISPATCHED によって測定できます。対象とするパフォーマンス・イベント選択 MSR 内で INV ビットをクリアし、カウンターマスク (CMASK) の値を 1 に設定します。

なお、ここで使用しているサイクル分類ビューは、基本的には近似的なものです。RS が一杯か空かを識別したり、RS は空になっているがパイプライン内のマイクロオペレーション (μop) の一部がリタイアの途中であるといった詳細な状況を把握することはできません。

## B.7.2 アウトオブオーダー (OOO) 実行のサイクル構成

アウトオブオーダー (OOO) エンジン内では、スペキュレーティブ・エグゼキューション (投機実行) がプログラムの処理を進行する上で重要な要素となります。しかし、予測が外れたコードパスに含まれるマイクロオペレーション (μop) のスペキュレーティブ・エグゼキューションは、実行リソースと実行帯域幅を消費する非生産的なアクティビティです。

Cycles\_not\_issuing\_uops は、定義により、アウトオブオーダー・エンジンがストールしたサイクル数 (Cycles\_stalled) を表します。このメトリックは、近似的にプログラムの処理が進行していないサイクル数として解釈できます。

発行されたマイクロオペレーション (μop) は、すべてリタイアメントで完了するとは限りません。リタイアメントに到達しないマイクロオペレーション (μop) は、プログラムの処理進行に貢献していません。したがって、さらなる近似的手法によって、Cycles\_issuing\_uops を以下の 2 つの指標に分類できます。

- Y **Cycles\_non\_retiring\_uops** — リタイアしないマイクロオペレーション (μop) のサイクル数を直接測定するイベントはありませんが、いくつかの利用可能なパフォーマンス・イベントからこのメトリックを算出できます。
  - 発行ポートを通過するマイクロオペレーション (μop) の発行速度は一定とします。したがって、 $uops\_rate = Dispatch\_uops / Cycles\_issuing\_uops$  と定義します。ここで、Dispatch\_uops はイベント RS\_UOPS\_DISPATCHED によって測定できます (INV ビットと CMASK をクリア)。
  - リタイアしない非生産的なマイクロオペレーション (μop) の数を、 $non\_productive\_uops = Dispatch\_uops - executed\_retired\_uops$  によって近似的に算出します。ここで、executed\_retired\_uops は、実行帯域幅を消費した、処理の進行に貢献する生産的なマイクロオペレーション (μop) を表します。
  - executed\_retired\_uops は、(イベント UOPS\_RETIRED.ANY によって測定) num\_retired\_uops と (イベント UOPS\_RETIRED.FUSED によって測定される) num\_fused\_uops の 2 つのメトリックの合計によって近似的に求められます。したがって、 $Cycles\_non\_retiring\_uops = non\_productive\_uops / uops\_rate$  となります。

- Y **Cycles\_retiring\_uops** — このメトリックは、 $Cycles\_retiring\_uops = num\_retired\_uops / uops\_rate$  から算出されます。

ここで使用するサイクル分類手法では、生産的なマイクロオペレーション (μop) と非生産的なマイクロオペレーション (μop) が同じサイクルでアウトオブオーダー・エンジンにディスパッチされる状況を識別できません。しかし、経験則によると、リタイアしないマイクロオペレーション (μop) が多数存在すると、アウトオブオーダー・エンジンが混雑し、プログラムがストールする可能性が高いため、この近似的手法は妥当であると言えます。

Total\_cycles に関連する、Cycles\_non\_retiring\_uops、Cycles\_stalled、Cycles\_retiring\_uops の 3 つのメトリックを評価することで、チューニング作業に役立つ以下の情報が得られます。

- Y Cycles\_non\_retiring\_uops が大きい場合、コードのレイアウトに注目して、分岐予測ミスを減らすことが重要です。
- Y Cycles\_non\_retiring\_uops と Cycles\_stalled がどちらも小さい場合、パフォーマンス・チューニングでベクトル化などの手法を重視し、ホットスポット関数のリタイアメントのスループットを向上させるべきです。
- Y Cycles\_stalled が大きい場合、マイクロアーキテクチャー・パイプラインのさらに深い場所に潜むボトルネックを発見するため、さらなるドリルダウンが必要です。

### B.7.3 パフォーマンス・ストールのドリルダウン

状況によっては、マイクロアーキテクチャー内の各種のストレスポイントを原因とするストールサイクル数を評価し、候補となるストレスポイントを合計する手法が有効です。しかし、この手法は非常に大まかな簡略化に基づくものであり、アウトオブオーダー・エンジンのスーパースケイラー構造とバッファリングを考慮することは困難です。

各種のパフォーマンス・イベントに関連するドメインは多様であるため、各ストレスポイントのパフォーマンスへの影響をサイクルベースで評価すると、影響の過大評価や過小評価によってさまざまな誤差が生じる可能性があります。

特定の原因のパフォーマンス全体に対する影響を推測する場合、インスタンスあたりのコストに対して、そのマイクロアーキテクチャー状態の発生回数を示すイベントカウントを掛けると、その影響が過大に評価される傾向があります。その結果、各種のストレスポイントが原因で失われたサイクル数の合計が、より正確な指標である Cycles\_stalled の値を上回ることがあります。

しかし、個々のストレスポイントが原因で失われたサイクル数を合計する手法は、コードをチューニングして各ストレスポイントのパフォーマンスへの影響を解決する際に、コード・チューニング・ループ作業の有効性を測定する反復

的な指標としては有益です。この節では、パフォーマンス・イベントによってカウントでき、本書で説明した以下のコーディングの推奨事項によって解決できる、パフォーマンスのボトルネックの一般的な原因について説明します。

以下の項目は、マイクロアーキテクチャーの典型的なストレスポイントを示します。

- Y **L2 ミスの影響** — L2 ロードミスがあると、メモリー・サブシステムのフルレイテンシーが明らかになります。システムメモリーのアクセス・レイテンシーは、一般にチップセットの種類によって 100 サイクル以上異なります。サーバー・チップセットはデスクトップ・チップセットより長いレイテンシーを示す傾向があります。L2 キャッシュミスの参照数は、MEM\_LOAD\_RETIRED.L2\_LINE\_MISS によって測定できます。

システムメモリーのレイテンシーに L2 ミスの数を掛けることで L2 ミスの全体の影響を推測する手法は、アウトオブオーダー・エンジンが複数の未処理のロードミス进行处理することを考慮していません。L2 ミスのレイテンシーと回数の乗算は、各 L2 ミスが逐次的に発生することを前提としています。

L2 ミスが影響する推測の精度を高めるには、CMASK の値を 1 に設定してイベント BUS\_REQUEST\_OUTSTANDING を使用する代替手法も検討する必要があります。この手法では、未処理のバス読み出し要求から得られるデータについてアウトオブオーダー・エンジンが待機しているサイクル数を測定します。これによって、メモリー・レイテンシーと L2 ミス数の乗算による過大評価の問題を解決できます。

- Y **L2 ヒットのシャドウ** — L2 からのメモリーアクセスがあると、L2 レイテンシーのコストが発生します (表 2-28 を参照)。L2 にヒットしたキャッシュライン参照の数は、MEM\_LOAD\_RETIRED.L1D\_LINE\_MISS - MEM\_LOAD\_RETIRED.L2\_LINE\_MISS の 2 つのイベントの差によって測定できます。

L2 ヒットのレイテンシーに L2 ヒット参照の数を掛けることで L2 ヒット全体の影響を推測する手法は、アウトオブオーダー・エンジンが複数の未処理のロードミス进行处理できることを考慮していません。

- Y **L1 DTLB ミスのシャドウ** — DTLB ルックアップ・ミスのコストは、約 10 サイクルです。イベント MEM\_LOAD\_RETIRED.DTLB\_MISS は、DTLB にミスしたマイクロオペレーション (μop) のロード回数を測定します。

- Y **LCP の影響** — LCP ストール全体の影響は、イベント ILD\_STALLS によって直接測定できます。イベント ILD\_STALLS は、低速デコーダーがトリガーされた回数を測定します。各インスタンスのコストは 6 サイクルです。

- Y **ストア・フォワーディングによるストールの影響** — ストア・フォワーディングが、ハードウェアが要求するアドレスやサイズ条件を満たさない場合、ストールが発生します。遅延は、ストア・フォワーディングのストール状況によって異なります。そのため、各種ストア・フォワーディングのストール状況をきめ細かく検出するパフォーマンス・イベントがいくつか用意されています。これには以下のものがあります。

- 未知のアドレスへの先行するストアによってブロックされたロード: この状況は、イベント Load\_Blocks.Sta によって測定できます。インスタンスあたりのコストはおおよそ 5 サイクルです。
- 先行するストアと部分的にオーバーラップしているか、ロードと先行するストアの間の 4K バイト・エイリアス・アドレスと部分的にオーバーラップしているロード。これらの 2 つの状況は、イベント Load\_Blocks.Overlap\_store で測定されます。
- キャッシュラインの境界にまたがるロード: これは Load\_Blocks.Until\_Retire によって測定されます。インスタンスあたりのコストはおおよそ 20 サイクルです。

## B.8 インテル® Core™ マイクロアーキテクチャーのイベント比率

付録 B.8 には、パフォーマンス・イベントを使用したパフォーマンスのボトルネックの迅速な診断例が記載されています。

この節では、パフォーマンス・イベントを使用して各種のパフォーマンス分析、ワークロードの特性評価、パフォーマンス・チューニングに役立つメトリックを評価する方法について、さらに詳しく説明します。



なお、インテル® Core™ マイクロアーキテクチャーの多くのパフォーマンス・イベント名は、XXXX.YYY の形式で表されます。この表記の一般的な規則では、XXXX は通常はパフォーマンス・イベント選択レジスター (IA32\_PERFVSELx) 内の固有のイベント選択コードに対応し、YYY は特定のマイクロアーキテクチャー状態を個別に定義する固有のサブイベント・マスクに対応します (『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 3B』の第 18 章と第 19 章を参照してください)。

## B.8.1 命令リタイアごとのクロック比率 (CPI)

1. 命令リタイアごとのクロック比率 (CPI):  

$$\text{CPU\_CLK\_UNHALTED.CORE} / \text{INST\_RETIRED.ANY}$$

インテル® Core™ マイクロアーキテクチャーは、理想的な状態では CPI が 0.25 まで低くなります。しかし、実際の多くのコードの CPI の値はそれより大きくなります。特定のワークロードで CPI の値が大きいほど、コードのチューニングによってパフォーマンスを改善できる可能性は高くなります。CPI は全体的な指標であり、どのマイクロアーキテクチャー・サブシステムが高い CPI 値に影響しているかを特定するものではありません。

以下の節では、フロントエンド、実行、メモリーとの相互作用の特性評価に役立つ一連のイベント比率を定義しています。

## B.8.2 フロントエンド比率

2. RS Full Ratio (RS フル比率):  

$$\text{RESOURCE\_STALLS.RS\_FULL} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$
3. ROB Full Ratio (ROB フル比率):  

$$\text{RESOURCE\_STALLS.ROB\_FULL} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$
4. Load or Store Buffer Full Ratio (ロードまたはストアバッファ・フル比率):  

$$\text{RESOURCE\_STALLS.LD\_ST} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

ROB Full Ratio、RS Full Ratio、Load Store Buffer Full Ratio の値が小さく、CPI 値が大きい場合は、フロントエンドがアウトオブオーダー・エンジンのバッファを一杯にするのに十分な速度で命令とマイクロオペレーション (μop) を供給できないため、アウトオブオーダー・エンジンが実行するマイクロオペレーション (μop) を待機している可能性があります。この場合は、フロントエンドにパフォーマンスの問題がないかをさらにチェックする必要があります。

### B.8.2.1 コードの局所性

5. Instruction Fetch Stall (命令フェッチストール):  

$$\text{CYCLES\_L1I\_MEM\_STALLED} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

Instruction Fetch Stall 比率は、命令フェッチユニット (IFU) がキャッシュミスと命令 TLB (ITLB) ミスが原因で、デコード用のキャッシュラインを供給できない状態のサイクル数の比率 (%) です。この比率が高い場合は、コードページのワーキングセットのサイズを縮小して、実行される命令の数を減らし、コードの局所性を改善することで、パフォーマンスが向上する可能性があります。

6. ITLB Miss Rate (ITLB ミス比率):  

$$\text{ITLB\_MISS\_RETIRED} / \text{INST\_RETIRED.ANY}$$

ITLB Miss Rate の値が大きい場合は、実行されるコードのページ数が多すぎて、多くの命令 TLB ミスが発生していることを示します。ITLB ミスを引き起こした命令がリタイアすると、パイプラインは自然に排出されます。また ITLB ミスは、多くの命令のフェッチをストールさせます。

7. L1 Instruction Cache Miss Rate (L1 命令キャッシュミス比率):  
 $L1\_MISSES / INST\_RETIRED.ANY$

L1 Instruction Cache Miss Rate の値が大きい場合、コードのワーキングセットが L1 命令キャッシュの容量を超えていることを示します。コードのワーキングセットを縮小すると、パフォーマンスは向上します。

8. L2 Instruction Cache Line Miss Rate (L2 命令キャッシュライン・ミス比率):  $L2\_IFETCH.SELF.I\_STATE / INST\_RETIRED.ANY$

L2 Instruction Cache Line Miss Rate が 0 より大きい場合は、L2 キャッシュからの命令キャッシュライン・ミスがプログラムのパフォーマンスに顕著な影響を与えていることを示します。

### B.8.2.2 分岐とフロントエンド

9. BACLEAR Performance Impact (BACLEAR のパフォーマンスへの影響):  
 $7 * BACLEARS / CPU\_CLK\_UNHALTED.CORE$

BACLEAR Performance Impact の値が大きい場合は、通常はコード内の分岐が多すぎて分岐予測ユニットが分岐を処理できていないことを示します。

10. Taken Branch Bubble (分岐した分岐のバブル):  
 $(BR\_TKN\_BUBBLE\_1 + BR\_TKN\_BUBBLE\_2) / CPU\_CLK\_UNHALTED.CORE$

Taken Branch Bubble の値が大きい場合は、連続的に分岐する分岐がコードに多数含まれ、フロントエンドにバブルを発生させていることを示します。このバブルは、後続のパイプ内の実行レイテンシーとストールによって隠蔽されない場合にのみ、パフォーマンスに影響します。

### B.8.2.3 スタックポインター追跡

11. ESP Synchronization (ESP 同期):  
 $ESP.SYNCH / ESP.ADDITIONS$

ESP Synchronization 比率は、ESP の明示的な使用 (例えば、ロード命令またはストア命令による使用) と暗黙的な使用 (例えば、PUSH 命令または POP 命令による使用) の比率を計算します。期待される比率は 0.2 以下です。この比率が 0.2 より高い場合は、コードを再編成して ESP 同期イベントを回避することを検討すべきです。

### B.8.2.4 マクロフュージョン

12. Macro-Fusion (マクロフュージョン):  
 $UOPS\_RETIRED.MACRO\_FUSSION / INST\_RETIRED.ANY$

Macro-Fusion 比率は、1 つのマクロオペレーション (μop) に何個のリタイアした命令が結合されているかを計算します。この比率は 32 ビット・バイナリー実行ファイルでは高くなり、それと等価な 64 ビット・バイナリーではかなり低い値を示します。また、64 ビット・バイナリーの実行速度は 32 ビット・バイナリーより低速です。その理由はおそらく、32 ビット・バイナリーがマクロフュージョンから大きなメリットを得ていると考えられます。

### B.8.2.5 レングス変更プリフィクス (LCP) のストール

13. LCP Delays Detected (検出された LCP 遅延):  
 $ILD\_STALL / CPU\_CLK\_UNHALTED.CORE$

LCP Delays Detected の値が大きい場合は、測定したコード内で多くのレングス変更プリフィクス (LCP) 遅延が発生しています。

## B.8.2.6 自己修正コードの検出

14. Self Modifying Code Clear Performance Impact (自己修正コードのクリアのパフォーマンスへの影響):  
 $MACHINE\_NUKES.SMC * 150 / CPU\_CLK\_UNHALTED.CORE * 100$

コードセクションへの書き込みを行い、すぐにそのコードを実行するプログラムでは、大きなペナルティーが生じます。Self Modifying Code Performance Impact は、プログラムが自己修正コードのペナルティーとして要したサイクル数の比率 (%) を推測します。

## B.8.3 分岐予測比率

付録 B.8.2.2 では、フロントエンドのパフォーマンスに影響を与える分岐について説明しました。この節では、分岐予測ミスの特性評価によく使用されるイベント比率について説明します。

### B.8.3.1 分岐予測ミス

15. Branch Misprediction Performance Impact (分岐予測ミスのパフォーマンスへの影響):  
 $RESOURCE\_STALLS.BR\_MISS\_CLEAR / CPU\_CLK\_UNHALTED.CORE * 100$

Branch Misprediction Performance Impact を使用して、プロセッサが分岐予測ミスからの回復に要したサイクル数の比率 (%) を計算できます。

16. Branch Misprediction per Micro-Op Retired (リタイアしたマイクロオペレーション (μop) あたりの分岐予測ミス):  
 $BR\_INST\_RETIRED.MISPRED / UOPS\_RETIRED.ANY$

Branch Misprediction per Micro-Op Retired 比率は、分岐の予測ミスが多すぎてコードのパフォーマンスが低下していないかを示します。低下している場合、分岐予測の精度を向上させることで、コードのパフォーマンスが著しく改善できる可能性があります。

また、個々の分岐予測ミスがパフォーマンスに大きな影響を及ぼしている場合もあります。この状況は、予測が外れた分岐に先行するコードの CPI が高く (キャッシュミスなど)、分岐予測ミスによって後続のコードがその処理を並列処理できない場合に発生します。このコードの CPI を小さくすれば、分岐予測ミスによるパフォーマンスへの影響を軽減できます。これらの状況を識別する方法については、ほかの比率を参照してください。

プリサイズイベントである BR\_INST\_RETIRED.MISPRED を使用して、予測が外れた分岐の実際のターゲットを検出できます。これにより、予測が外れた分岐を識別できます。

### B.8.3.2 仮想テーブルと間接呼び出し

17. Virtual Table Usage (仮想テーブルの利用):  
 $BR\_IND\_CALL\_EXEC / INST\_RETIRED.ANY$

Virtual Table Usage の値が大きい場合は、コードに多くの間接呼び出しが含まれていることを示します。間接呼び出しのデスティネーション・アドレスの予測は困難です。

18. Virtual Table Misuse (仮想テーブルの誤使用):  
 $BR\_CALL\_MISSP\_EXEC / BR\_INST\_RETIRED.MISPRED$

Branch Misprediction Performance Impact (比率 15) と Virtual Table Misuse の比率がともに高い場合は、間接関数呼び出しの予測ミスのために長い時間がかかっていることを示します。

C コード内での関数ポインターの明示的な使用以外に、間接呼び出しは C++ の継承、抽象型クラス、仮想メソッドの実装に使用されます。

### B.8.3.3 リターン予測ミス

19. Mispredicted Return Instruction Rate (予測ミスしたリターン命令の比率):  

$$\text{BR\_RET\_MISSP\_EXEC} / \text{BR\_RET\_EXEC}$$

プロセッサはコールとリターンのペアを追跡する特殊なメカニズムを備えています。プロセッサは、すべての CALL 命令がそれに対応する RETURN 命令を持つことを前提としています。RETURN 命令によって復帰したリターンアドレスが、それに対応する CALL 命令で格納されたアドレスと一致しない場合は、予測ミスのペナルティーが生じます。

## B.8.4 実行比率

この節では、マイクロオペレーション ( $\mu\text{op}$ ) と RS、ROB、実行ユニットなどの相互作用を解析するイベント比率について説明します。

### B.8.4.1 リソースストール

RS Full Ratio (比率 2) の値が大きい場合は、依存関係チェーンが長い場合、リザベーション・ステーション (RS) が頻繁にマイクロオペレーション ( $\mu\text{op}$ ) で一杯になっていることを示します。RS 内のマイクロオペレーション ( $\mu\text{op}$ ) は、先行するマイクロオペレーション ( $\mu\text{op}$ ) によってオペランドが計算されるのを待機しているか、実行ユニットが空くのを待機しています。これにより、複数実行ユニットを使用した並列処理が妨げられます。

ROB Full Ratio (比率 3) の値が大きい場合、リオーダーバッファ (ROB) が頻繁にマイクロオペレーション ( $\mu\text{op}$ ) で一杯になっていることを示します。これは通常、L2 キャッシュ要求ミスなどレイテンシーの長い操作を意味します。

### B.8.4.2 ROB 読み出しポートのストール

20. ROB Read Port Stall Rate (ROB 読み出しポートストールの比率):  

$$\text{RAT\_STALLS.ROB\_READ\_PORT} / \text{CPU\_CLK\_UNHALTED.CORE}$$

ROB Read Port Stall Rate は、ROB 読み出しポートストールを識別します。ただし、この比率は Resource Stall Ratio で示されるリソースのストールの回数が少ない場合にのみ参照すべきです。

### B.8.4.3 パーシャル・レジスター・ストール

21. Partial Register Stalls Ratio (パーシャル・レジスター・ストールの比率):  

$$\text{RAT\_STALLS.PARTIAL\_CYCLES} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

パーシャルストールの原因となるレジスターに頻繁にアクセスすると、アクセスレイテンシーが増加し、パフォーマンスが低下します。Partial Register Stalls Ratio は、パーシャルストールが発生しているサイクル数の比率 (%) です。

## B.8.4.4 パーシャル・フラグ・ストール

22. パーシャル・フラグ・ストール比率 (パーシャル・フラグ・ストールの比率):  

$$\text{RAT\_STALLS.FLAGS} / \text{CPU\_CLK\_UNHALTED.CORE}$$

パーシャル・フラグ・ストールによって大きなペナルティーが生じますが、このストールは簡単に回避できます。ただし、状況によっては、実際にフラグストールが発生していないにもかかわらず、Partial Flag Stalls Ratio の値が大きくなる場合があります。一部の命令は、RFLAGS レジスターを部分的に変更するため、パーシャル・フラグ・ストールを発生させる場合があります。最も典型的な例は、シフト命令 (SAR、SAL、SHR、SHL) と、INC および DEC 命令です。

## B.8.4.5 実行ドメイン間のバイパス

23. Delayed Bypass to FP Operation Rate (遅延した FP 演算バイパスの比率):  

$$\text{DELAYED\_BYPASS.FP} / \text{CPU\_CLK\_UNHALTED.CORE}$$
24. Delayed Bypass to SIMD Operation Rate (遅延した SIMD 演算バイパスの比率):  

$$\text{DELAYED\_BYPASS.SIMD} / \text{CPU\_CLK\_UNHALTED.CORE}$$
25. Delayed Bypass to Load Operation Rate (遅延したロード操作バイパスの比率):  

$$\text{DELAYED\_BYPASS.LOAD} / \text{CPU\_CLK\_UNHALTED.CORE}$$

ドメインバイパスがあると、命令レイテンシーが 1 サイクル増えます。コード内でドメインバイパスが頻繁に起こる操作を特定するには、上記の比率を使用します。

## B.8.4.6 浮動小数点演算パフォーマンス比率

26. Floating-Point Instructions Ratio (浮動小数点命令比率):  

$$\text{X87\_OPS\_RETIRED.ANY} / \text{INST\_RETIRED.ANY} * 100$$

浮動小数点演算の比率が高い場合は、浮動小数点演算アルゴリズム固有の最適化を適用します。

27. FP Assist Performance Impact (FP アシストのパフォーマンスへの影響):  

$$\text{FP\_ASSIST} * 80 / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

浮動小数点演算アシストは、デノーマルや NaN などの非正規の FP 値に対して起動されます。FP アシストは、正規の FP 演算の実行に比べて極めて低速です。生じるペナルティーはアシストによって異なります。FP Assist Performance Impact は、全体的な影響を推測します。

28. Divider Busy (除算器ビジー):  

$$\text{IDLE\_DURING\_DIV} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

Divider Busy の値が大きい場合は、除算器がビジーであり、ほかの実行ユニットやロード操作は数サイクルにわたって進行していないことを意味します。この比率は、並行して実行でき、除算器のペナルティーを隠蔽できる L1 データ・キャッシュ・ミスと L2 キャッシュ・ミスを考慮に入れていません。

29. Floating-Point Control Word Stall Ratio (浮動小数点制御ワードストールの比率):  

$$\text{ESOURCE\_STALLS.FPCW} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

浮動小数点制御ワード (FPCW) が頻繁に変更されると、パフォーマンスが大幅に低下します。FPCW 変更の主な理由は、FP から整数への変換時の丸めモードの変更です。

## B.8.5 メモリー・サブシステム - アクセス競合の比率

Load or Store Buffer Full Ratio (比率 4) の値が大きいと、ロードバッファまたはストアバッファが頻繁に一杯になるため、新しいマイクロオペレーション (μop) を実行パイプラインに送り込めなくなります。これによって実行の並列性が減少し、パフォーマンスが低下します。

30. Load Rate (ロード比率):  

$$\text{L1D\_CACHE\_LD.MESI} / \text{CPU\_CLK\_UNHALTED.CORE}$$

各コアは 1 サイクルごとに 1 つのメモリー読み出し操作を処理できます。Load Rate の値が大きい場合は、実行がメモリー読み出し操作によって制約を受けていることを示します。

31. Store Order Block (ストア・オーダー・ブロック):  

$$\text{STORE\_BLOCK.ORDER} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$

Store Order Block 比率は、L2 キャッシュをミスしたストア操作によって、その後のストア操作によるメモリー・サブシステムへのデータ移動がブロックされたサイクル数の比率 (%) です。さらに、これによってストアバッファが一杯になることがあります (比率 4 を参照)。

### B.8.5.1 L1 データキャッシュによってブロックされたロード

32. Loads Blocked by L1 Data Cache Rate (L1 データキャッシュによってブロックされたロードの比率):  

$$\text{LOAD\_BLOCK.L1D} / \text{CPU\_CLK\_UNHALTED.CORE}$$

Loads Blocked by L1 Data Cache Rate の値が大きい場合、多数の L1 データ・キャッシュ・ミスが同時に発生することで、リソースの不足のためロード操作が L1 データキャッシュによってブロックされていることを示します。

### B.8.5.2 4K エイリアシング/ストア・フォワーディング・ブロックの検出

33. Loads Blocked by Overlapping Store Rate (オーバーラップしているストアによってブロックされたロードの比率):  

$$\text{LOAD\_BLOCK.OVERLAP\_STORE} / \text{CPU\_CLK\_UNHALTED.CORE}$$

4K エイリアシングとストア・フォワーディング・ブロックは、ロードがさまざまな理由で先行するストアによってブロックされる状況を示します。いずれの場合も、同じイベント、LOAD\_BLOCK.OVERLAP\_STORE で検出できます。Loads Blocked by Overlapping Store Rate の値が大きい場合、4K エイリアシングやストア・フォワーディング・ブロックがパフォーマンスに影響を与える可能性があります。

### B.8.5.3 先行するストアによってブロックされたロード

34. Loads Blocked by Unknown Store Address Rate (未知のストアアドレスによってブロックされたロードの比率):  

$$\text{LOAD\_BLOCK.STA} / \text{CPU\_CLK\_UNHALTED.CORE}$$

Loads Blocked by Unknown Store Address Rate の値が大きい場合は、未知のアドレスを持つ先行するストアによってロードが頻繁にブロックされるため、パフォーマンスのペナルティーが生じていることを示します。

35. Loads Blocked by Unknown Store Data Rate (未知のストアデータによってブロックされたロードの比率):  

$$\text{LOAD\_BLOCK.STD} / \text{CPU\_CLK\_UNHALTED.CORE}$$

Loads Blocked by Unknown Store Data Rate の値が大きい場合、未知のデータを持つ先行するストアによってロードが頻繁にブロックされるため、パフォーマンスのペナルティーが生じていることを示します。

## B.8.5.4 メモリー・ディスアンビゲーション

インテル® Core™ マイクロアーキテクチャーのメモリー・ディスアンビゲーション機能により、未知のアドレスを持つストアによるロードのブロックを軽減することができます。(該当しないロード/ストアのディスアンビゲーションにより) この機能が失敗した場合は、イベント `LOAD_BLOCK.STA` と `MEMORY_DISAMBIGUATION.RESET` がカウントされます。

## B.8.5.5 ロード操作のアドレス変換

36. L0 DTLB Miss due to Loads - Performance Impact (ロードによる L0 DTLB ミス - パフォーマンスへの影響):  
`DTLB_MISSES.L0_MISS_LD * 2 / CPU_CLK_UNHALTED.CORE`

DTLB0 ミスの数が多い場合、ワークロードが使用するデータセットが多数のページにわたり DTLB0 の容量を超えていることを示します。高いミスの数は、CPI (比率 1) が低い (約 0.8) 場合にのみ、ワークロードのパフォーマンスに影響を与えると予想されます。CPI の値が大きい場合は、DTLB0 ミスサイクルはほかのレイテンシーによって隠蔽される可能性があります。

## B.8.6 メモリー・サブシステム - キャッシュミスの比率

### B.8.6.1 コード内のキャッシュミスの検出

インテル® Core™ マイクロアーキテクチャーは、L1 データ・キャッシュ・ミスまたは L2 キャッシュミスを引き起こしたリタイアしたロード命令を正確にカウントするイベントを備えています。これらはプリサイズイベントであり、イベントを発生させた命令に続く命令の命令ポインターを提供します。そのため、ポインターが指す命令の直前の命令が、キャッシュミスを発生させた命令となります。パフォーマンスの問題を解決する際に注目すべきロード命令の迅速な特定に役立つ以下のイベントがあります。

`MEM_LOAD_RETIRE.L1D_MISS`  
`MEM_LOAD_RETIRE.L1D_LINE_MISS`  
`MEM_LOAD_RETIRE.L2_MISS`  
`MEM_LOAD_RETIRE.L2_LINE_MISS`

### B.8.6.2 L1 データ・キャッシュ・ミス

37. L1 Data Cache Miss Rate (L1 データ・キャッシュ・ミス比率):  
`L1D_REPL / INST_RETIRED.ANY`

L1 Data Cache Miss Rate の値が大きい場合、コードの L1 データ・キャッシュ・ミスの頻度が高いため、L2 キャッシュアクセスのペナルティーが生じていることを示します。Loads Blocked by L1 Data Cache Rate (比率 32) を参照してください。

ロードが原因であるキャッシュミス、ストアが原因であるキャッシュミス、ロックされた操作が原因であるキャッシュミスは、それぞれイベント `L1D_CACHE_LD.I_STATE`、`L1D_CACHE_ST.I_STATE`、`L1D_CACHE_LOCK.I_STATE` によって別々にカウントできます。

### B.8.6.3 L2 キャッシュミス

38. L2 Cache Miss Rate (L2 キャッシュミス比率):  
`L2_LINES_IN.SELF.ANY / INST_RETIRED.ANY`

L2 Cache Miss Rate の値が大きい場合、実行中のワークロード内のデータセットが L2 キャッシュの容量を超えているため、データの一部が使用される前に排出されていることを示します。ハードウェア・プリフェッチまたはソフトウェア・プリフェッチ命令によって、必要なデータがすべて事前に取り込まれていない限り、メモリーからデータが取り込まれるとパフォーマンスに大きな影響を与えます。

39. L2 Cache Demand Miss Rate (L2 キャッシュ要求ミス比率):  

$$\text{L2\_LINES\_IN.SELF.DEMAND} / \text{INST\_RETIRED.ANY}$$

L2 Cache Demand Miss Rate の値が大きい場合、このワークロードが必要とするデータの取り込みにハードウェア・プリフェッチが利用されていません。データを使用するときに、その都度メモリーから取り込まれ、メモリーアクセスごとにメモリー・レイテンシーが発生しています。

## B.8.7 メモリー・サブシステム - プリフェッチ

### B.8.7.1 L1 データ・プリフェッチ

イベント L1D\_PREFETCH.REQUESTS は、DCU が L2 (またはメモリー) から DCU にキャッシュラインをプリフェッチしようとするたびにカウントされます。DCU プリフェッチャーが動作しており、このイベントがカウントされることが予想されるにもかかわらず、イベント MEM\_LOAD\_RETIRE.L1D\_MISS が検出された場合は、複数のロードによるロード命令アドレスの衝突が IP プリフェッチャーの動作を妨げている可能性があります。

### B.8.7.2 L2 ハードウェア・プリフェッチ

イベント L2\_LD.SELF.PREFETCH.MESI を使用して、L2 ハードウェア・プリフェッチャーが L2 に発行したプリフェッチ要求の数をカウントできます。L2 にプリフェッチされたキャッシュラインの数は、イベント L2\_LD.SELF.PREFETCH.I\_STATE によってカウントできます。

### B.8.7.3 ソフトウェア・プリフェッチ

ソフトウェア・プリフェッチに関するイベントは、それぞれのプリフェッチ・レベルを対象とします。

40. Useful PrefetchT0 Ratio (有効な PrefetchT0 の比率):  

$$\text{SSE\_PRE\_MISS.L1} / \text{SSE\_PRE\_EXEC.L1} * 100$$
41. Useful PrefetchT1 and PrefetchT2 Ratio (有効な PrefetchT1 と PrefetchT2 の比率):  

$$\text{SSE\_PRE\_MISS.L2} / \text{SSE\_PRE\_EXEC.L2} * 100$$

いずれかの有効なプリフェッチの比率が低い場合は、インテル® SSE プリフェッチ命令の一部が、すでにキャッシュ内にあるデータをプリフェッチしています。

42. Late PrefetchT0 Ratio (遅れた PrefetchT0 の比率):  

$$\text{LOAD\_HIT\_PRE} / \text{SSE\_PRE\_EXEC.L1}$$
43. Late PrefetchT1 and PrefetchT2 Ratio (遅れた PrefetchT1 および PrefetchT2 の比率):  

$$\text{LOAD\_HIT\_PRE} / \text{SSE\_PRE\_EXEC.L2}$$

遅れたプリフェッチの比率が高い場合、ソフトウェア・プリフェッチ命令の発行が遅すぎたため、プリフェッチされるデータを使用するロード操作がキャッシュラインの到着を待機していることを示しています。

## B.8.8 メモリー・サブシステム - TLB ミス比率

44. TLB miss penalty (TLB ミスのペナルティ):  

$$\text{PAGE\_WALKS.CYCLES} / \text{CPU\_CLK\_UNHALTED.CORE} * 100$$



TLB miss penalty の比率が高い場合、TLB ミスの処理に多くのサイクルを要していることを示します。TLB ミスの数を減らすことで、パフォーマンスが向上します。この比率には、DTLB0 ミスのペナルティーは含まれません (比率 37 を参照)。

以下の比率は、頻繁に TLB ミスを発生するメモリアクセスに焦点を絞るのに役立ちます。命令フェッチが原因である TLB ミスについては、「ITLB ミス比率 (比率 6) を参照してください。

45. DTLB Miss Rate (DTLB ミス比率):  

$$\text{DTLB\_MISSES.ANY} / \text{INST\_RETIRED.ANY}$$

DTLB Miss Rate の値が大きい場合、コードが短時間で多くのデータページをアクセスしているため、多数のデータ TLB ミスが発生していることを示します。

46. DTLB Miss Rate due to Loads (ロードによる DTLB ミスの比率):  

$$\text{DTLB\_MISSES.MISS\_LD} / \text{INST\_RETIRED.ANY}$$

DTLB Miss Rate due to Loads の値が大きい場合、コードが短時間で多くのページからデータをロードするため、多数のデータ TLB ミスが発生していることを示します。ロード操作による DTLB ミスは、ロード操作のレイテンシーを増加させるため、パフォーマンスに大きな影響を与えます。この比率には、DTLB0 ミスのペナルティーは含まれません (比率 37 を参照)。

DTLB ミスを発生するロード命令を正確に検出するには、プリサイスイベント MEM\_LOAD\_RETIRE.DTLB\_MISS を使用します。

47. DTLB Miss Rate due to Stores (ストアによる DTLB ミスの比率):  

$$\text{DTLB\_MISSES.MISS\_ST} / \text{INST\_RETIRED.ANY}$$

DTLB Miss Rate due to Stores の値が大きい場合、コードが短時間で多くのデータページをアクセスしているため、ストア操作による多数のデータ TLB ミスが発生していることを示します。これらのミスは、ほかの命令と並行して行われていない場合、パフォーマンスに影響します。また、多くのストア操作が連続している場合、その一部で DTLB ミスが発生すると、フルストア・バッファによるストールが発生することがあります。

## B.8.9 メモリー・サブシステム - コアとの相互作用

### B.8.9.1 変更されたデータの共有

48. Modified Data Sharing Ratio (変更されたデータの共有の比率):  

$$\text{EXT\_SNOOP.ALL\_AGENTS.HITM} / \text{INST\_RETIRED.ANY}$$

変更されたデータの共有が頻繁に発生するのは、2 つのスレッドが同一キャッシュラインにあるデータを変更するためです。変更されたデータの共有は、L2 キャッシュミスの原因となります。変更されたデータの共有が意図せずに発生すると (フォルス・シェアリングとも呼ばれる)、通常は要求ミスが発生し大きなペナルティーが生じます。フォルス・シェアリングを排除すると、コードのパフォーマンスは飛躍的に向上します。

49. Local Modified Data Sharing Ratio (ローカルな変更されたデータの共有の比率):  

$$\text{EXT\_SNOOP.THIS\_AGENT.HITM} / \text{INST\_RETIRED.ANY}$$

Modified Data Sharing Ratio は、システム内で測定された変更されたデータの共有の総量を示します。複数のプロセッサを搭載したシステムでは、Local Modified Data Sharing Ratio を使用して、同じプロセッサ内の 2 つのコアで共有される、変更されたデータの共有の量を示すことができます (1 つのプロセッサを搭載するシステムでは、2 つの比率はほぼ同じになります)。

## B.8.9.2 高速同期のペナルティー

### 50. Locked Operations Impact (ロックされた操作の影響):

$$(L1D\_CACHE\_LOCK\_DURATION + 20 * L1D\_CACHE\_LOCK.MESI) / CPU\_CLK\_UNHALTED.CORE * 100$$

高速同期は、通常ロックされたメモリアクセスを使用して実装されます。Locked Operations Impact の値が大きい場合、ワークロードで使用されるロックされた操作によって、大きなペナルティーが生じていることを示します。ロックされた操作のレイテンシーは、データの位置 (L1 データキャッシュ、L2 キャッシュ、その他のコアのキャッシュ、またはメモリー) によって異なります。

## B.8.9.3 同時に多発するストアミスとロードミス

### 51. Store Block by Snoop Ratio (スヌープによるストアブロックの比率):

$$(STORE\_BLOCK.SNOOP / CPU\_CLK\_UNHALTED.CORE) * 100$$

Store Block by Snoop Ratio の値が大きいと、ストア操作が頻繁にブロックされ、パフォーマンスが低下していることを示します。これは、プロセッサ内の 1 つのコアが連続するストアストリームを実行し、もう 1 つのコアが、L1 データキャッシュ内に見つからないキャッシュラインを検索してそのストリームを頻繁にスヌープする場合に発生します。

## B.8.10 メモリー・サブシステム - バスの特性

### B.8.10.1 バス利用率

#### 52. Bus Utilization (バス利用率):

$$BUS\_TRANS\_ANY.ALL\_AGENTS * 2 / CPU\_CLK\_UNHALTED.BUS * 100$$

Bus Utilization は、任意のタイプのバス・トランザクションの転送に使用されるバスサイクル数の比率 (%) です。シングルプロセッサ・システムでは、バス・トランザクションの大部分はデータを転送します。マルチプロセッサ・システムでは、バス・トランザクションの一部は、キャッシュの状態を調整してデータの coherence を維持するために使用されます。

#### 53. Data Bus Utilization (データバス利用率):

$$BUS\_DRDY\_CLOCKS.ALL\_AGENTS / CPU\_CLK\_UNHALTED.BUS * 100$$

Data Bus Utilization は、プロセッサとメモリーを含むシステム内のすべてのバス・エージェント間のデータ転送に使用されるバスサイクル数の比率 (%) です。バス利用率が高い場合、プロセッサとメモリーの間に大量のトラフィックが発生しています。メモリー・サブシステムのレイテンシーは、プログラムのパフォーマンスに影響することがあります。計算集約型のアプリケーションでバス利用率が高い場合、データとコードの局所性を改善する可能性を探すべきです。ほかのタイプのアプリケーション (例えば、あるメモリー領域から別のメモリー領域へ大量のデータをコピーする) では、バス利用率を最大化します。

#### 54. Bus Not Ready Ratio (バスの準備ができていない比率):

$$BUS\_BNR\_DRV.ALL\_AGENTS * 2 / CPU\_CLK\_UNHALTED.BUS * 100$$

Bus Not Ready Ratio は、新しいバス・トランザクションを開始できないバスサイクル数の比率 (%) を推測します。Bus Not Ready Ratio の値が大きい場合、バスに大きな負荷がかかっています。Bus Not Ready (BNR) 信号により、新しいバス・トランザクションが遅延し、そのレイテンシーがプログラムのパフォーマンスに大きな影響を与えることがあります。

#### 55. Burst Read in Bus Utilization (バス利用率中のバースト読み出し):

$$BUS\_TRANS\_BRD.SELF * 2 / CPU\_CLK\_UNHALTED.BUS * 100$$

Burst Read in Bus Utilization の値が大きい場合、バースト読み出し操作によるバスとメモリーのレイテンシーがプログラムのパフォーマンスに影響を与えていることを示しています。

56. RFO in Bus Utilization (バス利用率中の RFO):

$$\text{BUS\_TRANS\_RFO.SELF} * 2 / \text{CPU\_CLK\_UNHALTED.BUS} * 100$$

RFO in Bus Utilization の値が大きい場合、所有権読み出し (RFO) トランザクションのレイテンシーがプログラムのパフォーマンスに影響を与えていることを示します。RFO トランザクションは、ほかのバースト読み出し操作 (例えば、ロードが L2 にミスした場合など) に比べて、プログラムのパフォーマンスに大きな影響を与えます。比率 31 も参照してください。

### B.8.10.2 変更されたキャッシュラインの排出

57. L2 Modified Lines Eviction Rate (変更された L2 ラインの排出比率):

$$\text{L2\_M\_LINES\_OUT.SELF.ANY} / \text{INST\_RETIRED.ANY}$$

新しいキャッシュラインがメモリーから取り込まれると、新しいラインを格納するため、既存のキャッシュライン (おそらく変更済み) が L2 キャッシュから排出される可能性があります。変更されたラインが L2 キャッシュから頻繁に排出されると、L2 キャッシュミスのレイテンシーが増加し、バス帯域幅を消費します。

58. Explicit WB in Bus Utilization (バス利用率中の明示的ライトバック):

$$\text{BUS\_TRANS\_WB.SELF} * 2 / \text{CPU\_CLK\_UNHALTED.BUS} * 100$$

Explicit Write-back in Bus Utilization は、変更されたキャッシュラインの L2 キャッシュからの排出だけでなく、L1 データキャッシュからの排出も考慮します。この比率は、プロセッサからメモリーへの明示的ライトバックに使用されるバスサイクル数の比率 (%) を表します。

この付録には、よく使用される命令に関連したレイテンシーとスループットを記載した表を収録しています。<sup>15</sup> 命令タイミングデータは、プロセッサのファミリー/モデルによって異なります。付録は、以下の節で構成されています。

- Y **C.1 節「概要」** — 命令の選択とスケジューリングに関する問題の概要を説明しています。
- Y **C.2 節「用語説明」** — 本付録で使用する用語の定義を示します。
- Y **C.3 節「レイテンシーとスループット」** — よく使用される命令のスループット、レイテンシーを各表にまとめています。

## C.1 概要

この付録には、アセンブリ言語のプログラマーやコンパイラー開発者向けの情報が含まれています。ここで示す情報を参照することにより、依存関係チェーンに起因するレイテンシーが最も小さくなる命令シーケンスを選ぶ作業が容易になります。パフォーマンスに関する以下の要素にアプリケーションが影響されていないければ、ここで示す情報を活用すると、パフォーマンス面で数パーセントの効果が現れることが分かっています。

- Y キャッシュ・ミス・レイテンシー
- Y バス帯域幅
- Y I/O 帯域幅

2 章では以下のパフォーマンス問題について議論しましたが、プログラマーがこれらの問題を解決した後に問題となるのが、命令の選択とスケジューリングです。

- Y ストア・フォワードイングの制約事項を守る。
- Y キャッシュラインの分割やメモリー・オーダー・バッファの分割を避ける。
- Y 分岐予測を妨げない。
- Y メモリー・ロケーションへの `xchg` 命令の使用回数を最小限にする。

上記の項目は、適切な命令を選択することに関連しますが、本付録では次の問題点に焦点を当てています。重要な順番に記載していますが、どの項目がパフォーマンスに大きく影響するかはアプリケーションによって異なります。

- Y 実行コアへのマイクロオペレーション ( $\mu\text{op}$ ) のフローを最大化します。5 つ以上のマイクロオペレーション ( $\mu\text{op}$ ) で構成される命令は、マイクロコード ROM (MSROM) から供給されるため追加のステップが必要となります。長いフローのマイクロオペレーション ( $\mu\text{op}$ ) の命令では、フロントエンドが遅延し、実行コアへのマイクロオペレーション ( $\mu\text{op}$ ) 供給が停止します。

インテル® Pentium® 4 プロセッサ、インテル® Xeon® プロセッサでは、マイクロコード ROM からマイクロオペレーションを実行すると、多くの場合、複数のマイクロオペレーション ( $\mu\text{op}$ ) をトレースキャッシュにパックする効率が低下します。可能な場合は、4 つ以下のマイクロオペレーション ( $\mu\text{op}$ ) で構成される命令を選択します。例えば、単一のメモリーオペランドの 32 ビットの整数乗算を実行する場合、マイクロコード ROM を参照せずにトレースキャッシュに収まりますが、メモリーオペランドを持つ 16 ビットの整数乗算は収まりません。

<sup>15</sup> 命令レイテンシーはいくつかの限定された状況 (命令レイテンシーが顕著に現れる依存性チェーンがあるタイトなループなど) では役立つかもしれませんが、スーパースカラー、アウトオブオーダー・マイクロアーキテクチャー上のソフトウェア最適化では、一般に大きなスケールのコードパスの有効スループットをより高める方が有益です。命令のスケジューリングに影響する命令レイテンシーのみに注目するコーディング・テクニックは、アウトオブオーダー・マシンに悪影響を与えたり、命令レベルの並列性を損ねる可能性があるため、最適とはならないことがあります。

- Y リソースの競合を避けます。同じポートや同じ実行ユニットの奪い合いが起きないように複数の命令をインターリーブすると、スルーブットが改善できます。例えば、PADDQ 命令と PMULUDQ 命令を交互に配置した場合、どちらの命令も 2 クロックサイクルにつき発行は 1 回というスルーブットになります。インターリーブを適用すると、1 クロックサイクルにつき命令 1 つという高いスルーブットが得られます。これは、ポートが同じでも異なる実行ユニットが使用されるためです。発行ポートの帯域幅が狭くならないようにしたり、レイテンシーを隠蔽したり、ソフトウェアのパフォーマンスをより向上するには、スルーブットの高い命令を選択するのも有効です。
- Y クリティカル・パスで生じる依存関係チェーンのレイテンシーを最小にします。例えば、2 ビット左へシフトする操作では、2 つの加算命令としてエンコードする方が、1 つのシフト命令としてエンコードするよりも実行速度が速くなります。レイテンシーが問題でない場合、シフト命令を実行するとバイト・エンコーディングの密度が高まります。

本書には、一般的な規則、特殊な規則、コーディングに関するガイドライン、命令に関するデータを収録しましたが、そのほかにも、<http://developer.intel.com/software/products/index.htm> (英語) で入手できるソフトウェア解析およびチューニング・ツール群が利用できます。このツール群には、さまざまなパフォーマンス監視機能を備えたインテル® VTune™ Amplifier が含まれます。

## C.2 用語説明

データをいくつかの表にまとめています。表には次の情報が含まれます。

- Y **命令名** — 各命令のアセンブリー・ニーモニック。
- Y **レイテンシー** — 命令を構成しているすべてのマイクロオペレーション (μop) の実行が実行コアで完了するのに要するクロックサイクル数。
- Y **スルーブット** — 発行ポートが同じ種類の命令を再度受け入れられるようになるまで待たなければならないクロックサイクル数。多くの命令は、命令のスルーブットの方がレイテンシーよりもかなり小さくなります。
- Y RDRAND 命令のレイテンシーとスルーブットは上記の定義の例外です。RDRAND 命令を実行するハードウェア機能はアンコアにあるため、物理パッケージ内のすべてのプロセッサ・コアと論理プロセッサに依存します。シングルスレッドで実行される “rdrand に続く jnc” シーケンスを使用するソフトウェアのレイテンシーとスルーブットは、100 サイクル程度に抑えることができます。インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge ベースの第 3 世代インテル® Core™ プロセッサでは、アンコアの RDRAND による乱数送出力の総帯域幅は、およそ 500MB/秒です。同一プロセッサ・コアのマイクロアーキテクチャーと異なるアンコア実装では、RDRAND のレイテンシーとスルーブットは、インテル® Core™ プロセッサとインテル® Xeon® プロセッサでは異なります。

## C.3 レイテンシーとスルーブット

この節には、使用頻度の高い命令のレイテンシーとスルーブットに関する情報が記載されています。ここで扱われる命令には、インテル® MMX® テクノロジー命令、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE)、および以降の世代のインテル® SSE 命令が含まれるほか、整数命令や x87 浮動小数点命令についても、よく使用されるほとんどの命令が含まれます。

ダイナミック・エグゼキューションは複雑であり、また命令シーケンスの実行順序にかかわらず実行するという特性 (アウトオブオーダー) が実行コアに備わっているため、ここに示したレイテンシー・データを用いるだけでは実際のコードシーケンスのパフォーマンスを正確に予測するには十分ではありません。

- Y 命令のレイテンシー・データは、依存関係チェーンをチューニングする際に役立ちます。ただし、依存関係チェーンは、アウトオブオーダー・コアがマイクロオペレーション (μop) を並列実行する能力を制限します。命令のスルーブット・データは、依存関係チェーンによって妨げられない並列コードをチューニングする際に役立ちます。
- Y 数値はすべて概算値です。
  - マイクロアーキテクチャー (μop) の実装方式が将来変われば、この数値も変わる可能性があります。

- この数値は、命令レベルでのパフォーマンス評価基準として使用するものではありません。異なるマイクロアーキテクチャーベースのマイクロプロセッサで実行される命令レベルのパフォーマンスの比較は、複雑な話題であり、本書以外の情報を必要とします。

異なるマイクロアーキテクチャー間でレイテンシーとスループットのデータを比較すると、誤解を招きます。

C.3.1 節では、レジスター-レジスター方式の命令タイプのレイテンシーとスループットデータを示します。C.3.3 節では、レジスター-メモリー方式の命令タイプと、メモリー-レジスター方式の命令タイプのレイテンシーとスループットの仕様を調整する方法について説明します。

レイテンシーやスループットの数値が半クロックとなることがあります。これは倍速 ALU にのみ適用されます。

### C.3.1 レジスターオペランドのレイテンシーとスループット

命令のレイテンシーとスループットのデータを表 C-4 から表 C-15 に示します。インテル® AES-NI、インテル® SSE4.2、インテル® SSE4.1、インテル® SSSE3、インテル® SSE3、インテル® SSE2、インテル® SSE、インテル® MMX® テクノロジー命令を収録し、頻繁に使用されるインテル® 64 命令および IA-32 命令のほとんどを収録しています。異なるプロセッサ・マイクロアーキテクチャーの命令レイテンシーとスループットは、別々の列に示されます。

プロセッサの命令タイミングデータはプロセッサ固有であり、同じファミリーのエンコーディングでモデル・エンコーディング値が 3 の場合もあれば、2 未満の場合もあります。CPUID シグネチャーが 0xF2n の列と 0xF3n の列には、命令レイテンシーとスループットのデータセットが別々に示されます。0xF3n の列のデータは、CPUID シグネチャーが 0xF4n と 0xF6n のインテル® プロセッサにも適用されます。0xF2n の表記は、入力値が EAX = 1 の CPUID 命令によって報告される EAX レジスターの下位 12 ビットの 16 進値を表しています。「F」はファミリー・エンコーディングが 15、「2」はモデル・エンコーディングが 2、「n」は任意のステップング・エンコーディングを示します。

インテル® Core™ Solo プロセッサとインテル® Core™ Duo プロセッサは、06\_0EH として示されます。65nm プロセスを採用したインテル® Core™ マイクロアーキテクチャーベースのプロセッサは、06\_0FH です。

拡張版インテル® Core™ マイクロアーキテクチャーベースのプロセッサは、06\_17H と 06\_1DH で示されます。インテル® マイクロアーキテクチャー開発コード名 Nehalem ベースのプロセッサの CPUID ファミリー/モデル・シグネチャーは、06\_1AH、06\_1EH、06\_1FH、および 06\_2EH です。

インテル® マイクロアーキテクチャー開発コード名 Westmere ベースのプロセッサは、06\_25H、06\_2CH、および 06\_2FH です。

インテル® マイクロアーキテクチャー開発コード名 Sandy Bridge ベースのプロセッサは、06\_2AH および 06\_2DH です。

インテル® マイクロアーキテクチャー開発コード名 Ivy Bridge ベースのプロセッサは、06\_3AH と 06\_3EH です。

インテル® マイクロアーキテクチャー開発コード名 Haswell ベースのプロセッサは、06\_3CH、06\_45H、および 06\_46H です。

表 C-1 近年のマイクロアーキテクチャー (CPUIDシグネチャー) による SIMD 拡張命令のサポート

DisplayFamily_DisplayModel	近年のマイクロアーキテクチャー
06_4EH, 06_5EH	Skylake <sup>+</sup> マイクロアーキテクチャー
06_3DH, 06_47H, 06_56H	Broadwell <sup>+</sup> マイクロアーキテクチャー
06_3CH, 06_45H, 06_46H, 06_3FH	Haswell <sup>+</sup> マイクロアーキテクチャー
06_3AH, 06_3EH	Ivy Bridge <sup>+</sup> マイクロアーキテクチャー
06_2AH, 06_2DH	Sandy Bridge <sup>+</sup> マイクロアーキテクチャー
06_25H, 06_2CH, 06_2FH	Westmere <sup>+</sup> マイクロアーキテクチャー
06_1AH, 06_1EH, 06_1FH, 06_2EH	Nehalem <sup>+</sup> マイクロアーキテクチャー
06_17H, 06_1DH	拡張版インテル® Core™ マイクロアーキテクチャー
06_0FH	インテル® Core™ マイクロアーキテクチャー

命令レイテンシーはマイクロアーキテクチャーによって異なります。表 C-1 は、近年のマイクロアーキテクチャーにおけるインテル® ストリーミング SIMD 拡張命令を示しています。各マイクロアーキテクチャーは、CPUID 命令の「ファミリー」と「モデル」によって示される複数のシグネチャーに関連付けられている場合があります。特定のファミリー/モデルに関連付けられた、すべてのプロセッサのすべての命令セット拡張が有効となるわけではありません。特定の命令セット拡張がサポートされているかどうかを知るには、ソフトウェアは『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発者マニュアル、ボリューム 2A』の説明に従って適切な CPUID 機能フラグを使用する必要があります。

表 C-2 マイクロアーキテクチャー (CPUID シグネチャー) で挿入された命令拡張

SIMD 命令 拡張	DisplayFamily_DisplayModel								
	06_4EH, 06_5EH	06_3DH, 06_47H, 06_56H	06_3CH, 06_45H, 06_46H, 06_3FH	06_3AH, 06_3EH	06_2AH, 06_2DH	06_25H, 06_2CH, 06_2FH	06_1AH, 06_1EH, 06_1FH, 06_2EH	06_17H, 06_1DH	
CLFLUSHOPT	はい	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ
ADX, RDSEED	はい	はい	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ
AVX2, FMA, BMI1, BMI2	はい	はい	はい	いいえ	いいえ	いいえ	いいえ	いいえ	いいえ
F16C, RDRAND, RWFSGSBASE	はい	はい	はい	はい	いいえ	いいえ	いいえ	いいえ	いいえ
AVX	はい	はい	はい	はい	はい	いいえ	いいえ	いいえ	いいえ
AESNI, PCLMULQDQ	はい	はい	はい	はい	はい	はい	いいえ	いいえ	いいえ
SSE4.2, POPCNT	はい	はい	はい	はい	はい	はい	はい	はい	いいえ
SSE4.1	はい	はい	はい	はい	はい	はい	はい	はい	はい
SSSE3	はい	はい	はい	はい	はい	はい	はい	はい	はい
SSE3	はい	はい	はい	はい	はい	はい	はい	はい	はい
SSE2	はい	はい	はい	はい	はい	はい	はい	はい	はい
SSE	はい	はい	はい	はい	はい	はい	はい	はい	はい
MMX	はい	はい	はい	はい	はい	はい	はい	はい	はい

表 C-3 BMI1、BMI2 および汎用命令

命令	レイテンシー <sup>1</sup>		スループット	
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_4E, 06_5E	06_3D, 06_47, 06_56
ADCX	1	1	1	1
ADOX	1	1	1	1
RESEED	RDRAND と同様	RDRAND と同様	RDRAND と同様	RDRAND と同様

表 C-4 256 ビットのインテル® AVX 命令

命令	レイテンシー 1			スループット		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
VEXTRACTI128 xmm1, ymm2, imm	1	1	1	1	1	1
VMPSADBW	4	6	6	2	2	2
VPACKUSDW/SSWB	1	1	1	1	1	1
VPADDB/D/W/Q	1	1	1	0.33	0.5	0.5
VPADDSB	1	1	1	0.5	0.5	0.5
VPADDUSB	1	1	1	0.5	0.5	0.5
VPALIGNR	1	1	1	1	1	1
VPAVGB	1	1	1	0.5	0.5	0.5
VPBLEND	1	1	1	0.33	0.33	0.33
VPBLENDW	1	1	1	1	1	1
VPBLENDVB	1	2	2	1	2	2
VPBROADCASTB/D/SS/SD	3	3	3	1	1	1
VPCMPEQB/W/D	1	1	1	0.5	0.5	0.5
VPCMPEQQ	1	1	1	0.5	0.5	0.5
VPCMPGTQ	3	5	5	2	2	2
VPHADDW/D/SW	3	3	3	2	2	2
VINSERTI128 ymm1, ymm2, xmm, imm	3	3	3	1	1	1
VPMADDWD	5 <sup>b</sup>	5	5	0.5	1	1
VPMADDUBSW	5 <sup>b</sup>	5	5	0.5	1	1
VPMAXSD	1	1	1	0.5	0.5	0.5
VPMAXUD	1	1	1	0.5	0.5	0.5
VPMOVSX	3	3	3	1	1	1
VPMOVZX	3	3	3	1	1	1
VPMULDQ/UDQ	5 <sup>b</sup>	5	5	0.5	1	1
VPMULHSW	5 <sup>b</sup>	5	5	0.5	1	1
VPMULHW/LW	5 <sup>b</sup>	5	5	0.5	1	1
VPMULLD	10 <sup>b</sup>	10	10	1	2	2
VPOR/VPXOR	1	1	1	0.33	0.33	0.33
VPSADBW	3	5	5	1	1	1
VPSHUFB	1	1	1	1	1	1
VPSHUFD	1	1	1	1	1	1
VPSHUFLW/HW	1	1	1	1	1	1
VPSIGNB/D/W/Q	1	1	1	0.5	0.5	0.5



命令	レイテンシー <sup>1</sup>			スループット		
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
VPERMD/PS	3	3	3	1	1	1
VPSLLVD/Q	2	2	2	0.5	2	2
VPSRAVD	2	2	2	0.5	2	2
VPSRAD/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLDQ ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm, ymm, ymm	4	4	4	1	1	1
VPUNPCKHBW/WD/DQ/QDQ	1	1	1	1	1	1
VPUNPCKLBW/WD/DQ/QDQ	1	1	1	1	1	1
ALL VFMA	4	5	5	0.5	0.5	0.5
VPMASKMOVD/Q mem, ymm <sup>d</sup> , ymm				1	2	2
VPMASKMOVD/Q NUL, msk_0, ymm				>200 <sup>e</sup>	2	2
VPMASKMOVD/Q ymm, ymm <sup>d</sup> , mem	11	8	8	1	2	2
VPMASKMOVD/Q ymm, msk_0, [base+index] <sup>f</sup>	>200	200	200	>200	200	200

b: バイパスによる 1 サイクルのバブルを含みます

c: バイパスによる 2 つの 1 サイクルバブルを含みます

d: L1 参照で測定された MASKMOV 命令のタイミングと、少なくとも 1 つ以上の要素を選択するマスクレジスター。

e: 0 要素を選択するマスク値の MASKMOV ストア命令と、アシストによる遅延を被る不正アドレス (NULL または非 NULL)

f: 0 要素を選択するマスク値の MASKMOV ロード命令と、アシストによる遅延を被る正規のアドレス

表 C-5 L1D\* からタイミングデータを収集

命令	レイテンシー <sup>1</sup>			スループット		
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
VPGATHERDD/PS xmm, [vi128], xmm	20	17	14	4	5	7
VPGATHERQQ/PD xmm, [vi128], xmm	18	15	12	3	4	5
VPGATHERDD/PS ymm, [vi256], ymm	22	19	20	5	6	10
VPGATHERQQ/PD ymm, [vi256], ymm	20	16	15	4	5	7

\* ギャザー命令はメモリーを参照してデータ要素をフェッチ。タイミングデータは、L1 データキャッシュに存在するメモリー参照、および選択されたすべてのマスク要素に適用されます。

表 C-6 BMI1、BMI2 および汎用命令

命令	レイテンシー <sup>1</sup>			スループット		
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F
ANDN	1	1	1	0.5	0.5	0.5
BEXTR	2	2	2	0.5	0.5	0.5
BLSI/BLSMSK/BLSR	1	1	1	0.5	0.5	0.5
BZHI	1	1	1	0.5	0.5	0.5
MULX r64, r64, r64	4	4	4	1	1	1
PDEP/PEXT r64, r64, r64	3	3	3	1	1	1
RORX r64, r64, r64	1	1	1	0.5	0.5	0.5
SALX/SARX/SHLX r64, r64, r64	1	1	1	0.5	0.5	0.5
LZCNT/TZCNT	3	3	3	1	1	1

表 C-7 F16C、RDRAND 命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
RDRAND* r64	それぞれ異なる	それぞれ異なる	それぞれ異なる	<200	<300	250	250	<200
VCVTPH2PS ymm1, xmm2	7	6	6	7	1	1	1	1
VCVTPH2PS xmm1, xmm2	5	4	4	6	1	1	1	1
VCVTPS2PH ymm1, xmm2, imm	7	6	6	10	1	1	1	1
VCVTPS2PH xmm1, xmm2, imm	5	4	4	9	1	1	1	1

\* C.2 節を参照してください。

表 C-8 256 ビットのインテル® AVX 命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
VADDPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VADDSUBPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VANDNPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.5	1	1	1
VANDPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VBLENDPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	0.33	0.33	0.33	0.5
VBLENDVPD/PS ymm1, ymm2, ymm3,	1	2	2	1	1	2	2	1

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
DisplayFamily_DisplayModel								
ymm								
VCMPPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VCVTDQ2PD ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTDQ2PS ymm1, ymm2	4	3	3	3	0.5	1	1	1
VCVT(T)PD2DQ ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTPD2PS ymm1, ymm2	7	6	6	4	1	1	1	1
VCVT(T)PS2DQ ymm1, ymm2	4	3	3	3	1	1	1	1
VCVTPS2PD ymm1, xmm2	7	4	4	2	1	1	1	1
VDIVPD ymm1, ymm2, ymm3	14	16-23	25-35	27-35	8	16	27	28
VDIVPS ymm1, ymm2, ymm3	11	13-17	17-21	18-21	5	10	13	14
VDPPS ymm1, ymm2, ymm3	13	12	14	12	1.5	2	2	2
VEXTRACTF128 xmm1, ymm2, imm	3	3	3	3	1	1	1	1
VINSERTF128 ymm1, xmm2, imm	3	3	3	3	1	1	1	1
VMAXPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMINPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMOVAPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMOVDDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVDQA/U ymm1, ymm2	1	1	1	1	0.25	0.25	0.25	0.5
VMOVMSKPD/PS ymm1, ymm2	2	2	2	1	1	1	1	1
VMOVQ xmm1, xmm2	1	1	1	1	0.33	0.33	0.33	0.33
VMOVD/Q xmm1, r32/r64	2	1	1	1	1	1	1	1
VMOVD/Q r32/r64, xmm	2	1	1	1	1	1	1	1
VMOVNTDQ/PS/PD					1	1	1	1
VMOVSHDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVSLDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVUPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMULPD/PS ymm1, ymm2, ymm3	4	3	5	5	0.5	0.5	0.5	1
VORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
DisplayFamily_DisplayModel								
VPERM2F128 ymm1, ymm2, ymm3, imm	3	3	3	2	1	1	1	1
VPERMILPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VRCPPS ymm1, ymm2	4	7	7	7	1	2	2	2
VROUNDPD/PS ymm1, ymm2, imm	8	6	6	3	1	2	2	1
VRSQRTPS ymm1, ymm2	4	7	7	7	1	2	2	2
VSHUFDP/PS ymm1, ymm2, ymm3, imm	1	1	1	1	1	1	1	1
VSQRTPD ymm1, ymm2	<18	19-35	19-35	19-35	<12	16-27	16-27	28
VSQRTPS ymm1, ymm2	12	18-21	18-21	18-21	<6	13	13	14
VSUBPD/PS ymm1, ymm2, imm	4	3	3	3	0.5	1	1	1
VTESTPS ymm1, ymm2	3	2	2	2	1	1	1	1
VUNPCKHPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VUNPCKLPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VXORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VZEROUPPER	0	0	0	0	1	1	1	1
VZEROALL					12	8	8	9
VEXTRACTPS reg, xmm2, imm	3	2	2	2	1	1	1	1
VINSERTPS xmm1, xmm2, reg, imm	1	1	1	1	1	1	1	1
VMASKMOVPD/PS mem <sup>a</sup> , ymm, ymm					1	2	2	2
VMASKMOVPD/PS NUL, msk_0, ymm					>200 <sup>b</sup>	2	2	2
VMASKMOVPD/PS ymm, ymm <sup>a</sup> , mem	11	8	8	9	1	2	2	2
VMASKMOVPD/PS ymm, msk_0, [base+index] <sup>c</sup>	>200	200	200	200	>200	200	200	200

CPUID シグネチャー 06\_3AH のレイテンシーとスループットは、一般に 06\_2AH と同じであり、06\_2AH と異なるもののみが 06\_3AH のコラムに示されます。

- a: L1 参照で測定された MASKMOV 命令のタイミングと、少なくとも 1 つ以上の要素を選択するマスクレジスター。
- b: 0 要素を選択するマスク値の MASKMOV ストア命令と、アシストによる遅延を被る不正アドレス (NULL または非 NULL)
- c: 0 要素を選択するマスク値の MASKMOV ロード命令と、アシストによる遅延を被る正規のアドレス

VEX.128 エンコードされたインテル® AVX 命令のレイテンシーは、対応するレガシー 128 ビット命令に相当します。

表 C-9 インテル® AES-NI と PCLMULQDQ 命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
DisplayFamily_DisplayModel								
AESDEC/AESDECLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESENC/AESENCLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESIMC xmm1, xmm2	8	14	14	14	2	2	2	2
AESKEYGENASSIST xmm1, xmm2, imm	12	10	10	10	12	8	8	8
PCLMULQDQ xmm1, xmm2, imm	7 <sup>b</sup>	5	7	14	1	1	2	8

b: バイパスによる 1 サイクルのバブルが含まれます。

表 C-10 インテル® SSE4.2 命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
DisplayFamily_DisplayModel								
CRC32 r32, r32	3	3	3	3	1	1	1	1
PCMPSTRM xmm1, xmm2, imm	15	10	10	11	5	4	4	4
PCMPSTRM xmm1, xmm2, imm	10	10	10	11	6	5	5	4
PCMPSTRM xmm1, xmm2, imm	15	10	10	11	3	3	3	3
PCMPSTRM xmm1, xmm2, imm	15	11	11	11	3	3	3	3
PCMPGTQ xmm1, xmm2	3	5	5	5	0.33	1	1	1
POPCNT r32, r32	3	3	3	3	1	1	1	1
POPCNT r64, r64	3	3	3	3	1	1	1	1

表 C-11 インテル® SSE4.1 命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
DisplayFamily_DisplayModel								
BLENDD/S xmm1, xmm2, imm	1	1	1	1	0.33	0.33	0.33	0.5
BLENDVP/S xmm1, xmm2	1	2	2	2	1	2	2	1
DPPD xmm1, xmm2	9	7	9	9	1	1	1	1
DPPS xmm1, xmm2	13	12	14	13	2	2	2	2
EXTRACTPS xmm1, xmm2, imm	3	2	2	2	1	1	1	1
INSERTPS xmm1, xmm2, imm	1	1	1	1	1	1	1	1
MPSADBW xmm1, xmm2, imm	4	6	6	6	2	2	2	2
PACKUSDW xmm1, xmm2	1	1	1	1	1	1	1	0.5
PBLENVB xmm1, xmm2	2	2	2	2	2	2	2	1
PBLENDW xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PCMPEQQ xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRB/W/D reg, xmm1, imm	3	3	3	3	1	1	1	1
PHMINPOSUW xmm1, xmm2	4	5	5	5	1	1	1	1
PINSRB/W/D xmm1, reg, imm	2	2	2	2	1	1	1	1
PMAXSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMAXUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMOVSXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVSXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMULDQ xmm1, xmm2	5 <sup>b</sup>	5	5	5	0.5	1	1	1
PMULLD xmm1, xmm2	10 <sup>c</sup>	10	10	5	2	2	2	1
PTEST xmm1, xmm2	3	2	2	2	1	1	1	1
ROUNDPD/PS xmm1, xmm2, imm	6	6	6	3	2	2	2	1
ROUNDSD/SS xmm1, xmm2, imm	6	6	6	3	2	2	2	1

b: バイパスによる 1 サイクルのバブルを含みます

c: バイパスによる 2 つの 1 サイクルバブルを含みます

表 C-12 インテル® SSSE3 命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
PALIGNR xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PHADDD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PMADDUBSW xmm1, xmm2	5 <sup>b</sup>	5	5	5	0.5	1	1	1
PMULHRSW xmm1, xmm2	5 <sup>b</sup>	5	5	5	0.5	1	1	1
PSHUFB xmm1, xmm2	1	1	1	1	1	1	1	0.5
PSIGNB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PASB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5

b: バイパスによる 1 サイクルのパブルを含みます

表 C-13 インテル® SSE3 浮動小数点命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
ADDSUBPD/ADDSUBPS	4	3	3	3	0.5	1	1	1
HADDPD xmm1, xmm2	6	5	5	5	2	2	2	2
HADDPS xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPD xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPS xmm1, xmm2	6	5	5	5	2	2	2	2
MOVDDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSHDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSLDUP xmm1, xmm2	1	1	1	1	1	1	1	1

表 C-14 インテル® SSE2 の 128 ビット整数命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
CVTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
CVTTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
MASKMOVDQU xmm, xmm					7	6	6	6
MOVD xmm, r64/r32	2	1	1	1	1	1	1	1
MOVD r64/r32, xmm	2	1	1	1	1	1	1	1
MOVDQA xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVDQU xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVQ xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PACKSSWB/PACKSSDW/ PACKUSWB xmm, xmm	1	1	1	1	1	1	1	0.5
PADDB/PADDW/PADDD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PADDSB/PADDSW/PADDUS B/PADDUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PADDQ/ PSUBQ <sup>3</sup> xmm, xmm1	1	1	1	1	0.33	0.5	0.5	0.5
PAND xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PANDN xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PAVGB/PAVGW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPEQB/PCMPEQD/PCMP EQW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPGTB/PCMPGTD/PCMP GTW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRW r32, xmm, imm8	3	3	3	3	1	1	1	1
PINSRW xmm, r32, imm8	2	2	2	2	2	2	2	1
PMADDWD xmm, xmm	5 <sup>b</sup>	5	5	5	0.5	1	1	1
PMAX xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMIN xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMOVMASK <sup>3</sup> r32, xmm	2	2	2	2	1	1	1	1
PMULHUW/PMULHW/PMUL LW xmm, xmm	5 <sup>b</sup>	5	5	5	0.5	1	1	1
PMULUDQ xmm, xmm	5 <sup>b</sup>	5	5	5	0.5	1	1	1
POR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PSADBW xmm, xmm	3	5	5	5	1	1	1	1
PSHUFD xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFHW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFLW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLW/PSLLD/PSLLQ xmm, imm8	1	1	1	1	1	1	1	1
PSLL/PSRL xmm, xmm	2	2	2	2	1	1	1	1
PSRAW/PSRAD xmm, imm8	1	1	1	1	1	1	1	1
PSRAW/PSRAD xmm, xmm	2	2	2	2	1	1	1	1



命令	レイテンシー <sup>1</sup>				スルーブット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
CPUID								
PSRLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSRLW/PSRLD/PSRLQ xmm, imm8	1	1	1	1	1	1	1	1
PSUBB/PSUBW/PSUBD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PSUBSB/PSUBSW/PSUBUSB/PSUBUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKHQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PXOR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33

b: バイパスによる 1 サイクルのバブルを含みます

表 C-15 インテル® SSE2 の浮動小数点命令

命令	レイテンシー <sup>1</sup>				スルーブット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
CPUID								
ADDPD xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSD xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPD xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPD xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
CMPSD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
COMISD xmm, xmm	2	2	2	2	1	1	1	1
CVTDQ2PD xmm, xmm	5	4	4	4	1	1	1	1
CVTDQ2PS xmm, xmm	4	3	3	3	1	1	1	1
CVTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTPD2PS xmm, xmm	5	4	4	4	1	1	1	1
CVT[T]PS2DQ xmm, xmm	4	3	3	3	1	1	1	1
CVTPS2PD xmm, xmm	5	2	2	2	1	1	1	1
CVT[T]SD2SI r64/r32, xmm	6	4	4	5	1	1	1	1
CVTSD2SS xmm, xmm	5	4	4	4	1	1	1	1
CVTSI2 SD xmm, r64/r32	5	3	3	4	1	1	1	1
CVTSS2SD xmm, xmm	5	2	2	2	1	1	1	1
CVTTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTTSD2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPD xmm, xmm <sup>1</sup>	14	<14	14-20	16-22 (15-20)	4	8	13	22 (14)

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
DIVSD xmm, xmm	14	<14	14-20	16-22 (15-20)	4	5	13	22 (14)
MAXPD xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSD xmm, xmm	4	3	3	3	0.5	1	1	1
MINPD x mm, xmm	4	3	3	3	0.5	1	1	1
MINSD xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MOVMSKPD r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSD xmm, xmm	1	1	1	1	1	1	1	1
MOVUPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MULPD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
MULSD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
ORPD xmm, xmm	1	1	1	1	0.33	1	1	1
SHUFPD xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPD xmm, xmm <sup>2</sup>	18	20	20	22 (21)	6	13	13	22 (14)
SQRTSD xmm, xmm	18	20	20	22 (21)	6	7	13	22 (14)
SUBPD xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSD xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISD xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPD xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPD xmm, xmm	1	1	1	1	1	1	1	1
XORPD <sup>3</sup> xmm, xmm	1	1	1	1	0.33	1	1	1

注意:

1. DIVPD/DIVSD のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは 6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。
2. SQRTPD/SQRTSD のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは 6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。

表 C-16 インテル® SSE の浮動小数点命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
CPUID								
ADDPS xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSS xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPS xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPS xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPS xmm, xmm	4	3	3	3	0.5	1	1	1
CMPSS xmm, xmm	4	3	3	3	0.5	1	1	1
COMISS xmm, xmm	2	2	2	2	1	1	1	1
CVTSI2SS xmm, r32	6	4	4	5	1	1	1	1
CVTSS2SI r32, xmm	6	4	4	5	1	1	1	1
CVT[T]SS2SI r64, xmm	6	4	4	5	1	1	1	1
CVTTSS2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPS xmm, xmm <sup>1</sup>	11	<11	<13	10-14	3	4	6	14 (6)
DIVSS xmm, xmm	11	<11	<13	10-14	3	2.5	6	14 (6)
MAXPS xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSS xmm, xmm	4	3	3	3	0.5	1	1	1
MINPS xmm, xmm	4	3	3	3	0.5	1	1	1
MINSS xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MOVHLPs xmm, xmm	1	1	1	1	1	1	1	1
MOVLHPS xmm, xmm	1	1	1	1	1	1	1	1
MOVMSKPS r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSS xmm, xmm	1	1	1	1	1	1	1	1
MOVUPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MULPS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
MULSS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
ORPS xmm, xmm	1	1	1	1	0.33	1	1	1
RCPPS xmm, xmm	4	5	5	5	1	1	1	1
RCPSS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTPS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTSS xmm, xmm	4	5	5	5	1	1	1	1
SHUFPS xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPS xmm, xmm <sup>2</sup>	13	13	13	14	3	7	7	14 (7)
SQRTSS xmm, xmm	13	13	13	14	3	4	7	14 (7)
SUBPS xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSS xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISS xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPS xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPS xmm, xmm	1	1	1	1	1	1	1	1
XORPS xmm, xmm	1	1	1	1	1	1	1	1
LFENCE <sup>3</sup>					6	5	5	4
MFENCE <sup>3</sup>					40	35	35	35

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
CPUID								
SFENCE <sup>3</sup>					7	6	6	5
STMXCSR <sup>3</sup>					1	1	1	1
FXSAVE3					90	71	75	78

注意:

1. DIVPS/DIVSS のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは 6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。
2. SQRTPS/SQRTSS のレイテンシーとスループットは、入力値によって異なります。特定の値ではハードウェアは即座に完了でき、スループットは 6 と低いかもしれません。同様に、ある値の入力のレイテンシーは 10 サイクル未満であるかもしれません。
3. FXSAVE/LFENCE/MFENCE/SFENCE/STMXCSR のスループットは、デスティネーションが L1 データキャッシュにある状態で計測されます。

表 C-17 汎用命令

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
ADC/SBB reg, reg	1	2	2	2	0.5	1	1	1
ADC/SBB reg, imm	1	2	2	2	0.5	1	1	1
ADD/SUB	1	1	1	1	0.25	0.25	0.25	0.33
AND/OR/XOR	1	1	1	1	0.25	0.25	0.25	0.33
BSF/BSR	3	3	3	3	1	1	1	1
BSWAP	2	2	2	2	0.5	0.5	0.5	1
BT	1	1	1	1	0.5	0.5	0.5	0.5
BTC/BTR/BTS	1	1	1	1	0.5	0.5	0.5	0.5
CBW/CWDE/CDQE	1	1	1	1	1	1	1	1
CDQ	1	1	1	1	1	1	1	1
CQO	1	1	1	1	0.5	0.5	0.5	0.5
CLC					0.5	0.5	0.5	0.5
CMC					0.25	0.33	0.33	0.33
STC					0.25	0.33	0.33	0.33
CLFLUSH <sup>13</sup>					2-50	3-50	3-50	5-50
CLFLUSHOPT <sup>14</sup>					2-10	NA	NA	NA
CMOVE/CMOVcc	1	1	2	2	0.5	0.5	0.5	0.5
CMOVBE/NBE/A/NA	2	2	3	3	1	1	1	1
CMP/TEST	1	1	1	1	0.25	0.25	0.25	0.33
CPUID (EAX = 0)					100	100	100	95
CPUID (EAX != 0)					>200	>200	>200	>200
CMPXCHG r64, r64	5	5	5	5	5	5	5	5
CMPXCHG8B m64	15	8	8	8	15	8	8	8

命令	レイテンシー <sup>1</sup>				スループット			
	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C, 06_45, 06_46, 06_3F	06_3A 06_3E
CPUID								
CMPXCHG16B m128	19	10	10	10	19	10	10	10
Lock CMPXCHG8B m64	22	19	19	24	22	19	19	24
Lock CMPXCHG16B m128	32	28	28	29	32	28	28	29
DEC/INC	1	2	2	2	0.25	0.25	0.25	0.33
IMUL r64, r64	3	3	3	3	1	1	1	1
IMUL r64 <sup>11</sup>	4,5	3,4	3,4	3,4	1	1	1	1
IMUL r32	5	4	4	4	1	1	1	1
IDIV r64 (RDX!= 0) <sup>9</sup>					85-100	85-100	85-100	85-100
IDIV r32 <sup>10</sup>					20-26	20-26	20-26	19-25
LEA	1	1	1	1	0.5	0.5	0.5	0.5
LEA [base+index]disp	3	3	3	3	1	1	1	1
MOVSB/MOVSW	1	1	1	1	0.25	0.25	0.25	0.33
MOVZB/MOVZW	1	1	1	1	0.25	0.25	0.25	0.33
DIV r64 (RDX!= 0) <sup>9</sup>					85-95	85-95	85-95	85-95
DIV r32 <sup>10</sup>					20-26	20-26	20-26	19-25
MUL r64 <sup>11</sup>	4,5	3,4	3,4	3,4	1	1	1	1
NEG/NOT	1	2	2	2	0.25	0.25	0.25	0.33
PAUSE					140	10	10	10
RCL/RCR reg, 1	2	2	2	2	2	1.5	1.5	1.5
RCL/RCR	6	6	6	6	6	6	6	6
RDTSC					13	10	10	20
RDTSCP					20	30	30	30
ROL/ROR reg 1	1(2flg)	1(2flg)	1(2flg)	1(2flg)	1	1	1	1
ROL/ROR reg imm1	1	1	1	1	0.5	0.5	0.5	0.5
ROL/ROR reg, cl	2	2	2	2	1.5	1.5	1.5	1.5
LAHF/SAHF	3	2	2	2				
SAL/SAR/SHL/SHR reg, imm	1	1	1	1	0.5	0.5	0.5	0.5
SAL/SAR/SHL/SHR reg, cl	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
SETBE	2	2	2	2	1	1	1	1
SETE	1	1	1	1	0.5	0.5	0.5	0.5
SHLD/RD reg, reg, cl	6	4	4	2(4flg)	1.5	1	1	1.5
SHLD/RD reg, reg, imm	3	3	3	1	0.5	0.5	0.5	0.5
XSAVE <sup>12</sup>					98	100	100	100
XSAVEOPT <sup>12</sup>					86	90	90	90
XADD	2	2	2	2	1	1	1	1
XCHG reg, reg	1	1	1	2	1	1	1	1
XCHG reg, mem	22	19	19	19	22	19	19	19

### C.3.2 表の脚注について

この付録に収録されている表を参照する際は、以下の点に注意してください。

1. 5 つ以上のマイクロオペレーション (μop) から構成される複合命令の多くは、(最悪の場合を考慮して) 控えめに算出した推測値を示しています。そのような命令をアウトオブオーダー・コア実行ユニットで実行したときの実際のパフォーマンスは、各表に示したレイテンシー・データよりも若干高速なものから、著しく高速なものまで幅があります。
2. Intel NetBurst® マイクロアーキテクチャーに適用される実行ユニットの名称は、CPUID シグネチャーのファミリー・エンコーディング 15、モデル・エンコーディング 0、1、2 のプロセッサに該当します。これには、ALU、FP\_EXECUTE、FP\_MOVE、MEM\_LOAD、MEM\_STORE が含まれます。次の点に注意してください。
  - FP\_EXECUTE ユニットは実際には複数の実行ユニットからなる 1 つのクラスターです。およそ 7 つの独立した実行ユニットで構成されます。
  - FP\_ADD ユニットでは、x87 と SIMD 浮動小数点の両方の加算と減算が実行されます。
  - FP\_MUL ユニットでは、x87 と SIMD 浮動小数点の両方の乗算が実行されます。
  - FP\_DIV ユニットでは、x87 と SIMD 浮動小数点の両方の除算と平方根が実行されます。
  - MMX\_SHFT ユニットでは、シフトとローテートが実行されます。
  - MMX\_ALU ユニットでは、SIMD 整数の ALU 演算が実行されます。
  - MMX\_MISC ユニットでは、逆数 MMX 計算と一部の整数演算が実行されます。
  - FP\_MISC では、ポート 1 に含まれている他の実行ユニットのうち、上に示した 6 つの実行ユニットから分離されている実行ユニットが指定されます。
3. コードシーケンスでインテル® 64 命令と IA-32 命令をいくつか繰り返して呼び出すようにすると、この表に列挙した数値よりも 1 ~ 2 サイクルだけレイテンシーが短くなることがあります。
4. 超越関数命令のレイテンシーとスループットは、ダイナミック・エグゼキューション環境では大きく変わることがあります。したがって、超越関数命令については、概算値か、ある程度幅を持った数値のいずれを示しています。
5. コードシーケンスにおける FXCH 命令のレイテンシーはゼロです。ただし、1 クロックサイクルあたり 1 命令しか発行できません。
6. ロード定数命令 FINCSTP と FDECSTP は、コードシーケンスではレイテンシーが 0 です。
7. 分岐の予測的中率を高くするため、条件分岐命令を選択するときは、3.4.1 節「分岐予測の最適化」の推奨事項に従ってください。分岐予測に成功すると、jcc のレイテンシーは事実上ゼロになります。
8. シフト回数が 1 回の RCL/RCR は最適化されています。1 以外のシフト回数の RCL/RCR を使用すると、実行速度が遅くなります。これは、インテル® Pentium® 4 プロセッサとインテル® Xeon® プロセッサに適用されます。
9. “DIV/IDIV r64” のレイテンシーとスループットは、RDX:RAX の入力値の有効桁数によって異なります。RDX の入力が 0 の場合、スループットはかなり高くなります。これは、“DIV/IDIV r32” と同様です。RDX が 0 でない場合は、示される範囲よりもかなり低くなります。入力 RDX:RAX (除数の有効ビット数に比例して) または出力商の有効ビット数の増加に従って、スループットは減少 (サイクル数の増加) します。“DIV/IDIV r64” のレイテンシーは、入力値の有効ビットによって異なります。特定セットの入力値では、レイテンシーはサイクルのスループットとほぼ同等です。
10. “DIV/IDIV r32” のスループットは、入力 EDX:EAX および/または、除数 r32 の有効ビットサイズに対する除算の商の有効数値数によって異なります。入力 EDX:EAX または出力商の有効ビット数の増加に従って、スループットは減少 (サイクル数の増加) します。“DIV/IDIV r32” のレイテンシーは、入力値の有効ビットによって異なります。特定セットの入力値では、レイテンシーはサイクルのスループットとほぼ同等です。
11. 128 ビットの結果を生成する MUL r64 のレイテンシーには 2 組の数字がありますが、下位 64 ビットの結果 (RAX) の読み取りから利用までのレイテンシーはより小さくなります。128 ビットの結果の上位 64 ビットのレイテンシー (RDX) の方が大きくなります。
12. XSAVE と XSAVEOPT のスループットは、デスティネーションが L1 データキャッシュにある状態で測定され、YMM ステートを含みます。
13. CLFLUSH のスループットは、バッファサイズ範囲に対するクリーンなキャッシュラインである場合を示します。CLFLUSH のスループットは、次の要因によって劇的に減少します: (a) 連続して実行される CLDLUSH の数、(b) 変更されたキャッシュラインの排出による、他のコヒーレント状態のキャッシュラインに関連する追加のコスト。7.4.6 節を参照してください。

14. CLFLUSHOPT のスループットは、バッファサイズ範囲に対するクリーンなキャッシュラインである場合を示します。CLFLUSHOPT のスループットは、次の要因によって劇的に減少します。(a) 変更されたキャッシュラインの排出による、他のコヒーレント状態のキャッシュラインに関連する追加のコスト、(b) 連続するキャッシュラインの数。7.4.7 節を参照してください。

### C.3.3 メモリーオペランドを持つ命令

メモリーオペランドを持つ命令のレイテンシーは、メモリー/キャッシュ階層内のデータの局所性や各マイクロアーキテクチャーに固有の特性などの多くの要因によって異なります。一般に、ソフトウェアにおける局所性のチューニングと命令の選択は、互いに独立したアプローチが可能です。このため、表 C-4 ~ 表 C-18 は命令を選択する目的に使用できます。メモリー/キャッシュ階層内のデータ移動レイテンシーとスループットは、命令のレイテンシーとスループットとは独立して扱うことができます。キャッシュ階層のレイテンシーのデータは、第 2 章に記載されています。

#### C.3.3.1 ソフトウェアから観察できるメモリー参照のレイテンシー

個別の命令のメモリー参照レイテンシーを測定する場合、観察されるレイテンシーは多くの要因に影響されます。アクセスパターン、キャッシュの局所性、ハードウェア・プリフェッチの効果を除き、異なるマイクロアーキテクチャーでは、命令エンコードに関するデスティネーションやメモリーアドレス形式のレジスタードメインなどの違いが表れる可能性があります。

次の表は、近年のインテル® マイクロアーキテクチャーにおけるメモリー参照のエンコードによる、ポインター追尾構文を使用するソフトウェアの L1D キャッシュヒットのレイテンシーのいくつかを示しています。

表 C-18 ソフトウェアが計測可能なポインター追尾における L1 データ・キャッシュ・レイテンシーの変化

ポインター追尾構文	観察された L1D レイテンシー
MOV rax, [rax]	4
MOV rax, disp32[rax], disp32 < 2048	4
MOV rax, [rcx+rax]	5
MOV rax, disp32[rcx+rax], disp32 < 2048	5