# 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 Volume 2:

# 旧世代のインテル® 64 および IA-32 プロセッサー・アーキテクチャー、スループット、およびレイテンシー

参照ドキュメント番号: 356477-050 April 2024

## 注意事項:

**本ドキュメントはレイアウト調整および校閲を行っておりません。誤字脱字、製品名や用語の表記、レイアウト等の不具合が含まれる可能性があることを予めご了承ください。**

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。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

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

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。詳細については、各システムメーカーまたは販売店にお問い合わせいただくか、http://www.intel.co.jp/ を参照してください。

テストでは、特定のシステムでの個々のテストにおけるコンポーネントの性能を文書化しています。

ハードウェア、ソフトウェア、システム構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。購入を検討される場合は、ほかの情報も参考にして、パフォーマンスを総合的に評価することをお勧めします。性能やベンチマーク結果について、さらに詳しい情報をお知りになりたい場合は、http://www.intel.com/benchmarks/ (英語) を参照してください。

絶対的なセキュリティーを提供できるコンピューター・システムはありません。インテルは、データやシステムの紛失または盗難、およびそのような損失に起因するいかなる損害についても責任を負いません。

本資料に記載されているインテル製品に関する侵害行為または法的調査に関連して、本資料を使用または使用を促すことはできません。本資料を使用することにより、お客様は、インテルに対し、本資料で開示された内容を含む特許クレームで、その後に作成したものについて、非独占的かつロイヤルティー無料の実施権を許諾することに同意することになります。

本資料に含まれるコードは、0 条項 BSD オープンソース・ライセンス (OBSD) の下にライセンスが許可されることを除いて、本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。本資料に記載されているすべての情報は、予告なく変更されることがあります。インテルの最新の製品仕様およびロードマップをご希望の方は、インテルの担当者までお問い合わせください。

結果はインテル社内での分析またはアーキテクチャーのシミュレーションあるいはモデリングに基づくものであり、情報提供のみを目的としています。システム・ハードウェア、ソフトウェア、構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。

本書で紹介されている注文番号付きのドキュメントや、インテルのその他の資料を入手するには、1-800-548-4725 (アメリカ合衆国) までご連絡いただくか、http://www.intel.com/design/literature.htm (英語) を参照してください。

Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation. 無断での引用、転載を禁じます。

# 改版履歴

1. 2023 年 9 月 001JA
   最新英語版 (248966-048 August 2023) で Volume 1 と 2 に分冊されて、Volume 2 に新しい資料番号 (356477-001) が付与されました

2. 2024 年 2 月 002JA
   最新英語版 (356477-002US January 2024) の変更を適用しました。

3. 2024 年 6 月 050JA
   最新英語版 (356477-050US April 2024) の変更を適用しました。Volume 1 から第 8 章「インテル® TSX の推奨事項」が移動されました。

前バージョンの英語版 (356477-050US April 2024) から追加または変更されたテキストは紫色の文字で強調表示しています。

# 目次


## 第 1 章 Haswell† マイクロアーキテクチャー

1.1 はじめに
1.2 フロントエンド
1.3 アウトオブオーダー･エンジン
1.3.1 実行エンジン
1.4 キャッシュとメモリー･サブシステム
1.4.1 ロード操作とストア操作の拡張
1.4.2 アンラミネーション
1.5 Haswell-E† マイクロアーキテクチャー
1.6 Broadwell† マイクロアーキテクチャー

## 第 2 章 インテル® マイクロアーキテクチャー Sandy Bridge†

2.1 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
2.1.1 フロントエンド
2.1.1.1 レガシー･デコード･パイプライン
2.1.1.2 デコード済み命令キャッシュ
2.1.1.3 分岐予測
2.1.1.4 マイクロオペレーション (uop) キューおよびループストリーム検出器 (LSD)
2.1.2 アウトオブオーダー･エンジン
2.1.2.1 リネーマー
2.1.2.2 スケジューラー
2.1.3 実行コア
2.1.4 キャッシュ階層
2.1.4.1 ロード操作とストア操作の概要
2.1.5 L1D キャッシュ
2.1.6 リング･インターコネクトとラスト･レベル･キャッシュ
2.1.7 データ･プリフェッチ
2.2 システム･エージェント
2.3 Ivy Bridge† マイクロアーキテクチャー

## 第 3 章 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

3.1 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
3.1.1 フロントエンド
3.1.1.1 分岐予測ユニット
3.1.1.2 命令フェッチユニット
3.1.1.3 命令キュー (IQ)
3.1.1.4 命令デコード
3.1.1.5 スタックポインター追尾
3.1.1.6 マイクロフュージョン
3.1.2 実行コア
3.1.2.1 発行ポートと実行ユニット
3.1.3 インテル® アドバンスト･メモリー･アクセス
3.1.3.1 ロードとストア
3.1.3.2 1 次キャッシュからのデータ･プリフェッチ
3.1.3.3 データ･プリフェッチ･ロジック
3.1.3.4 ストア･フォワーディング
3.1.3.5 メモリー･ディスアンビゲーション

3.1.4 インテル® アドバンスト・スマート・キャッシュ

3.1.4.1 ロード
3.1.4.2 ストア

第 4 章 Nehalem† マイクロアーキテクチャー

4.1 マイクロアーキテクチャー・パイプライン
4.2 フロントエンドの概要
4.3 実行エンジン

4.3.1 発行ポートと実行ユニット

4.4 キャッシュとメモリー・サブシステム

4.4.1 ロード操作とストア操作の強化
4.4.2 効率的なアライメント・ハザードの処理
4.4.3 ストア・フォワーディングの強化

4.5 REP (リピート) 文字列の強化
4.6 システム・ソフトウェアの強化

4.6.1 電力消費の効率化
4.6.2 Nehalem† マイクロアーキテクチャーにおけるインテル® ハイパースレッディング・テクノロジーのサポート

第 5 章 Knights Landing† マイクロアーキテクチャーの最適化

5.1 フロントエンド

5.1.1 アウトオブオーダー・エンジン
5.1.2 アンタイル

5.2 Knights Landing† 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

5.2.1 Gather および Scatter 命令の利用
5.2.2 拡張された逆数命令の利用

5.3 インテル® AVX-512CD 命令の利用

5.3.1 インテル® ハイパースレッディング・テクノロジーの利用
5.3.2 フロントエンドに関する考察
5.3.3 命令デコーダー
5.3.4 4GB 境界を超える間接分岐

5.4 整数実行に関する考察

5.4.1 フラグの利用
5.4.2 整数除算

5.5 FP およびベクトル実行の最適化

5.5.1 命令選択の注意事項
5.5.2 前世代からの組込み関数の移植
5.5.3 ベクトル化のトレードオフを推定する

5.6 メモリー最適化

5.6.1 データ・アライメント
5.6.2 ハードウェア・プリフェッチャー
5.6.3 ソフトウェア・プリフェッチ

5.6.3.1 メモリー実行クラスター

5.6.4 ストア・フォワーディング
5.6.5 ウェイとセットの競合
5.6.6 ストリーミング・ストアと通常のストア
5.6.7 コンパイラー・オプションとディレクティブ
5.6.8 ダイレクト割り当て MCDRAM キャッシュ

第 6 章 旧世代の 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

6.1 概要

6.2 Intel Atom® マイクロアーキテクチャー
6.2.1 Intel Atom® 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・テクノロジーのサポート
6.3 Intel Atom® マイクロアーキテクチャー向けのコーディングの推奨事項
6.3.1 Intel Atom® マイクロアーキテクチャーのフロントエンドの最適化
6.3.2 実行コアの最適化
6.3.2.1 整数命令の選択
6.3.2.2 アドレス生成
6.3.2.3 整数乗算
6.3.2.4 整数シフト命令
6.3.2.5 パーシャル・レジスター・アクセス
6.3.2.6 FP/SIMD 命令の選択
6.3.3 メモリーアクセスの最適化
6.3.3.1 ストア・フォワーディング
6.3.3.2 第 1 レベルキャッシュ
6.3.3.3 セグメントベース
6.3.3.4 文字列移動
6.3.3.5 引数渡し
6.3.3.6 関数呼び出し
6.3.3.7 乗算/加算の依存関係チェーンの最適化
6.3.3.8 位置に依存しないコード
6.4 命令レイテンシー
6.5 Silvermont† マイクロアーキテクチャー
6.5.1 整数パイプライン
6.5.2 浮動小数点パイプライン
6.6 Goldmont† マイクロアーキテクチャー
6.7 Goldmont Plus† マイクロアーキテクチャー
6.8 コーディングの推奨事項
6.8.1 フロントエンドの最適化
6.8.1.1 命令デコーダー
6.8.1.2 フロントエンドの IPC が高い場合の考慮事項
6.8.1.3 4GB 境界を超える分岐
6.8.1.4 ループアンロールおよびループストリーム検出器
6.8.1.5 コードとデータの混在
6.8.2 実行コアの最適化
6.8.2.1 スケジューリング
6.8.2.2 アドレス生成
6.8.2.3 FP 乗算-加算-ストアの実行
6.8.2.4 整数乗算の実行
6.8.2.5 ゼロイディオム
6.8.2.6 慣用的な NOP
6.8.2.7 ムーブの排除 (Move Elimination) と ESP の折りたたみ (Folding)
6.8.2.8 スタック操作命令
6.8.2.9 フラグの利用
6.8.2.10 SIMD 浮動小数点と x87 命令
6.8.2.11 SIMD 整数命令
6.8.2.12 ベクトル化の注意事項
6.8.2.13 その他の SIMD 命令
6.8.2.14 命令の選択
6.8.2.15 整数除算
6.8.2.16 整数シフト
6.8.2.17 ポーズ命令
6.8.3 メモリーアクセスの最適化
6.8.3.1 PALIGNR でアライメントされていないメモリーアクセスを軽減

6.8.3.2 メモリー実行の問題を最小化する
6.8.3.3 ストア･フォワーディング
6.8.3.4 PrefetchW 命令
6.8.3.5 キャッシュラインの分割とアライメント
6.8.3.6 セグメントベース
6.8.3.7 コピーと文字列のコピー

6.9 命令レイテンシーとスループット

第 7 章 命令レイテンシーとスループット

7.1 概要
7.2 用語説明
7.3 レイテンシーとスループット

7.3.1 レジスターオペランドのレイテンシーとスループット
7.3.2 表の脚注について
7.3.3 メモリーオペランドを持つ命令

7.3.3.1 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

第 8 章 インテル® TSX の最適化

8.1 はじめに

8.1.1 インテル® TSX について
8.1.2 最適化の概略

8.2 アプリケーション･レベルのチューニングと最適化

8.2.1 既存のインテル® TSX 対応ロック･ライブラリー

8.2.1.1 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
8.2.1.2 プログラムの修正を必要とするライブラリー

8.2.2 初期のチェック
8.2.3 アプリケーションの実行とプロファイル
8.2.4 トランザクション･アボートを最小限に抑える

8.2.4.1 データ競合によるトランザクション･アボート
8.2.4.2 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
8.2.4.3 ロック省略固有のトランザクション･アボート
8.2.4.4 HLE 固有のトランザクション･アボート
8.2.4.5 その他のトランザクション･アボート

8.2.5 トランザクション実行専用のコードパスの使用
8.2.6 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

8.2.6.1 アボートしないで非省略実行へ遷移する
8.2.6.2 早期のアボート強制
8.2.6.3 選択したロックを省略しない

8.3 インテル® TSX 対応の同期ライブラリーの開発

8.3.1 HLE プリフィクスの追加
8.3.2 省略に適したクリティカル･セクションのロック
8.3.3 ロックの省略における HLE または RTM の使用
8.3.4 ロックの省略に RTM を使用するラッパーの例
8.3.5 RTM フォールバック･ハンドラーのガイドライン
8.3.6 インテル® TSX による省略に適したロックの実装

8.3.6.1 HLE を使用する単純なスピンロックの実装
8.3.6.2 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
8.3.6.3 インテル® TSX を使用するチケットロックの実装
8.3.6.4 インテル® TSX を使用するキューベースのロックの実装

8.3.7 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
8.3.8 永続的な非省略実行を回避する
8.3.9 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

8.3.10 HLE と RTM を混在させる

8.4 インテル® TSX のパフォーマンス監視サポートを利用する

8.4.1 トランザクション成功を測定する
8.4.2 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
8.4.3 トランザクション･アボートのサンプリング
8.4.4 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
8.4.5 RTM フォールバック･ハンドラー向けの XABORT 引数
8.4.6 トランザクション･アボートのコールグラフ
8.4.7 LBR とトランザクション･アボート
8.4.8 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
8.4.9 HLE 固有のパフォーマンス監視イベント
8.4.10 インテル® TSX の有用なメトリックを計算する

8.5 パフォーマンスのガイドライン
8.6 デバッグのガイドライン
8.7 インテル® TSX 用の一般的な組込み関数

8.7.1 RTM C 組込み関数

8.7.1.1 古い gcc* 互換コンパイラーによる RTM 組込み関数のエミュレート

8.7.2 gcc およびその他の Linux* 互換コンパイラーの HLE 組込み関数

8.7.2.1 gcc 4.8 による HLE 組込み関数の生成
8.7.2.2 C++11 atomic のサポート
8.7.2.3 古い gcc 互換コンパイラーによる HLE 組込み関数のエミュレート

8.7.3 Windows* C/C++ コンパイラーの HLE 組込み関数

## 1.1 はじめに

Haswell† マイクロアーキテクチャーは、Sandy Bridge† および Ivy Bridge† 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 1-1 に Haswell† 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、1.2 節から 1.4 節で示される多くの機能は、Broadwell† マイクロアーキテクチャーにも適用されます。Broadwell† マイクロアーキテクチャーの拡張は、1.6 節にまとめられています。

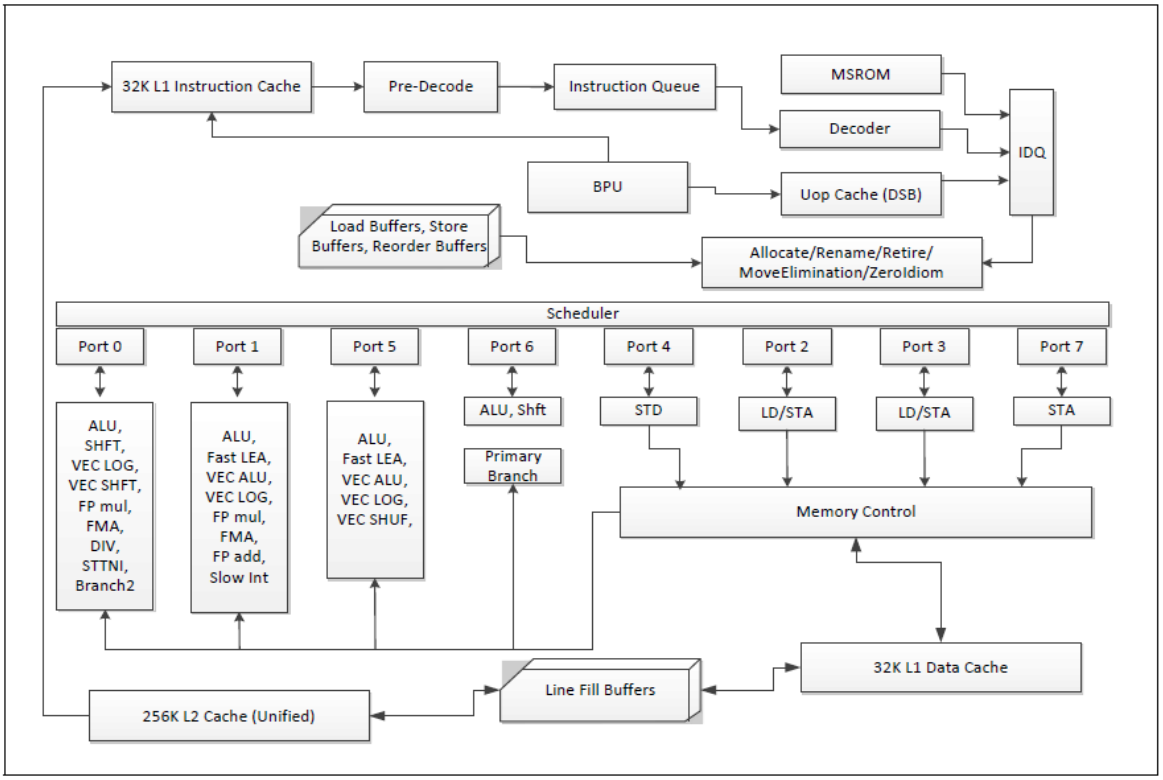


図 1-1 インテル® マイクロアーキテクチャー Haswell† の CPU コア・パイプライン

Haswell† マイクロアーキテクチャーの基本パイプラインは、以下の革新的な機能を提供します。

- インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2)、FMA のサポート
- 整数値演算と暗号化を高速化する新しい汎用命令
- 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
- 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 (uop) をディスパッチ可能
- 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
- L1 データキャッシュと L2 キャッシュの帯域幅が増加
- 2 つの FMA 実行パイプライン
- 4 つの数値演算ユニット (ALU)
- 3 つのストア・アドレス・ポート
- 2 つの分岐実行ユニット
- IA プロセッサー・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
- 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† 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、プロセッサー・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、複数のプロセッサー・コアとの柔軟な統合をサポートしています。図 1-2 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。

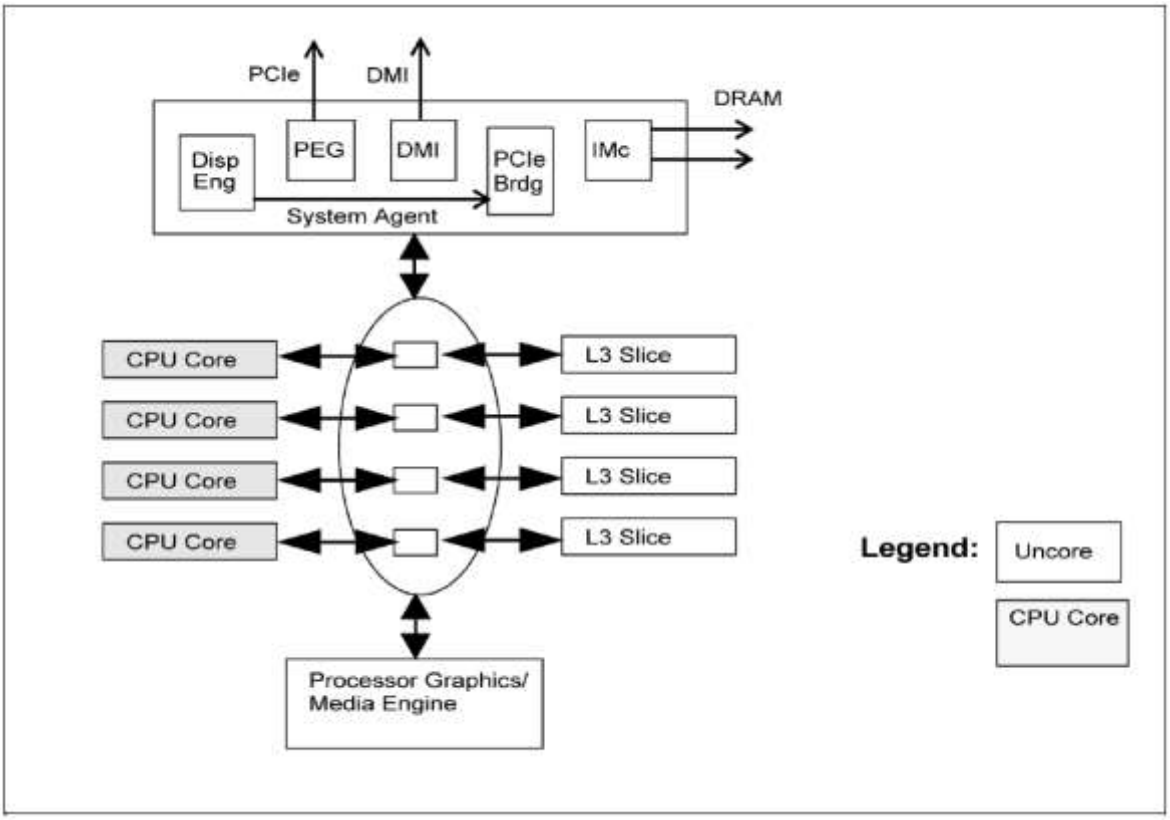


図 1-2 インテル® マイクロアーキテクチャー Haswell† の 4 コアのシステム統合

## 1.2 フロントエンド

インテル® マイクロアーキテクチャー Haswell† 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† (2.1.1 節) とインテル® マイクロアーキテクチャー Ivy Bridge† (2.3 節) をベースに開発され、次の点が拡張されています。

- マイクロオペレーション (uop) 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ー間で均等に分割されます。
- 命令デコーダーは、アクティブな論理プロセッサー間で交互に使用される。1 つの論理プロセッサー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ーがデコーダーを続けて使用されます。
- ループストリーム検出器 (LSD)/マイクロオペレーション (uop) は、56 マイクロオペレーション (uop) 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 (uop) キューは、インテル® 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ー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† では、各コアに 28 エントリーのマイクロオペレーション (uop) キューの複製が提供されます)。

## 1.3 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 (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop)、倍精度では 16 マイクロオペレーション (uop) であり、インテル® マイクロアーキテクチャー Sandy Bridge† 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 (uop) 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† では 168 マイクロオペレーション (uop) です)。

### 1.3.1 実行エンジン

表 1-1 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 (uop) を示します。

表 1-1 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、<br>Fast LEA、 | Load_Addr、<br>Store_addr | Store_data | ALU、Fast LEA | ALU、<br>Shift、<br>JEU | Store_addr、<br>Simple_AGU |
| SIMD_Log、<br>STTNI、<br>SIMD_Shifts | SIMD_ALU、<br>SIMD_Log | | | SIMD_ALU、<br>SIMD_Log | | |
| FMA/FP_mul、<br>Div | FMA/FP_mul、<br>FP_add | | | FP/Int Shuffle | | |
| 2nd_Jeu | slow_int | | | | | |

表 1-2 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 1-2 はまた、Broadwell † マイクロアーキテクチャー・ベースのプロセッサーのみで利用可能な命令をいくつか含んでいます。

表 1-2 Haswell † 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)[1] etc. |
| Slow Int | 1 | mul, imul, bsr, rcl, shld, mulx, pdep, etc. |
| BM | 2 | andn, bextr, blsi, blsmsk, bzhi, etc |
| SIMD Log | 3 | (v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)blendp*, vpblendd |
| 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)psrldq, (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 † 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ーのみで利用可能。

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 † では 54 エントリー)、マイクロオペレーション (uop) 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 (uop) をディスパッチできます。RS でマイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop) 実行にバイパスする方法とその遅延サイクル数を表 2-25 に示します。

表 1-3 マイクロオペレーション (uop) 間のバイパスによる遅延 (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 | | • uOP (ポート 5)<br>• uOP (ポート 6) + 1 サイクル遅延 | • uOP (ポート 5)<br>• uOP (ポート 6) + 1 サイクル遅延 | 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 |

## 1.4 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・インターコネクト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ー・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 1-4 にキャッシュ階層の詳細を示します。

表 1-4 インテル® マイクロアーキテクチャー Haswell† 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[1] | 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[2] | 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 つ処理できます。

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 1-5 インテル® マイクロアーキテクチャー Haswell† 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 |

## 1.4.1 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 (uop) の同時実行をサポートするため、72 のロードバッファーと 42 のストアバッファーが装備されています。

## 1.4.2 アンラミネーション

いくつかのマイクロフューズされた命令は、単一の uop として割り当てできないため、マイクロ・オペレーション・キューでは 2 つの uop に分解されます。フューズ (融合) された命令を uop に分解する処理は、アンラミネーションと呼ばれます。

フューズされた命令のソースが 3 つ以上である場合、アンラミネーションが行われます。

アンラミネーションされるコンテキストの命令ソースは、メモリー・アドレス・ベース、メモリー・アドレス・インデックス、ソースレジスター、デスティネーション・レジスター (フラグを含む)、またはソースとデスティネーション・レジスターのいずれかであると考えられます。

アンラミネーション・コンテキストのメモリーオペランドは、最大 2 つのソースを持ちます。x86 命令セットのメモリーアドレスは、base + index * scale + displacement で構成されます。

ベース (base) とインデックス (index) のみが命令ソースとしてカウントされます。インデックスが使用される場合、ベースが指定されていなくともベースとしてカウントされることに注意してください。

さらに、ソースとデスティネーション・レジスターは 2 つのソースとしてカウントされます。これもまた、ソースとデスティネーション・レジスターが同じである場合にも当てはまります。

次の表は、マイクロフューズ命令の例とアンラミネーションの詳細を示します。

表 1-6 フロントエンドの構成要素

| 命令の例 | ソース | ディスティネーション | ソースとディスティネーション | インデックス | ベース | ソース[1]の数 | アンラミネーション |
|---|---|---|---|---|---|---|---|
| mulss xmm1, [4*rax+100] | - | - | xmm1 | rax | 0 | 3 | なし |
| vmulss xmm1, xmm1, [rax +100] | xmm1 | xmm1 | - | - | rax | 3 | なし |
| vmulss xmm1, xmm1, [4*rax+100] | xmm1 | xmm1 | - | rax | 0 | 4 | あり |
| cmp rax, [rbx+4*rax+4] | rax | フラグ | - | rax | rbx | 4 | あり |
| cmp rax, [rbx+4] | rax | フラグ | - | - | rbx | 3 | なし |

注意:

1. 推奨事項: アンラミネーションを避けるには、マイクロフューズ命令のソースを 4 以下に保ちます。

## 1.5 Haswell-E† マイクロアーキテクチャー

Haswell-E† 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で説明した同じプロセッサー・コアを包括するインテル® プロセッサー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† マイクロアーキテクチャー・ベースのプロセッサーは、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† マイクロアーキテクチャーは、スケーラビリティーと高いパフォーマンス向けに多面的なプロセッサー・アーキテクチャーをサポートします。Haswell-E† 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。

- 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。
- 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。
- 物理プロセッサーごとに最大 40 レーンの PCI Express* 3.0 リンクを提供します。
- それぞれの物理プロセッサーで、最大 18 個のプロセッサー・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 1-3 に、Haswell-E† マイクロアーキテクチャーを使用する 12 コアのプロセッサー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† マイクロアーキテクチャーを実装するプロセッサー・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® プロセッサー E5 v3 ファミリーのデータシートを参照してください。

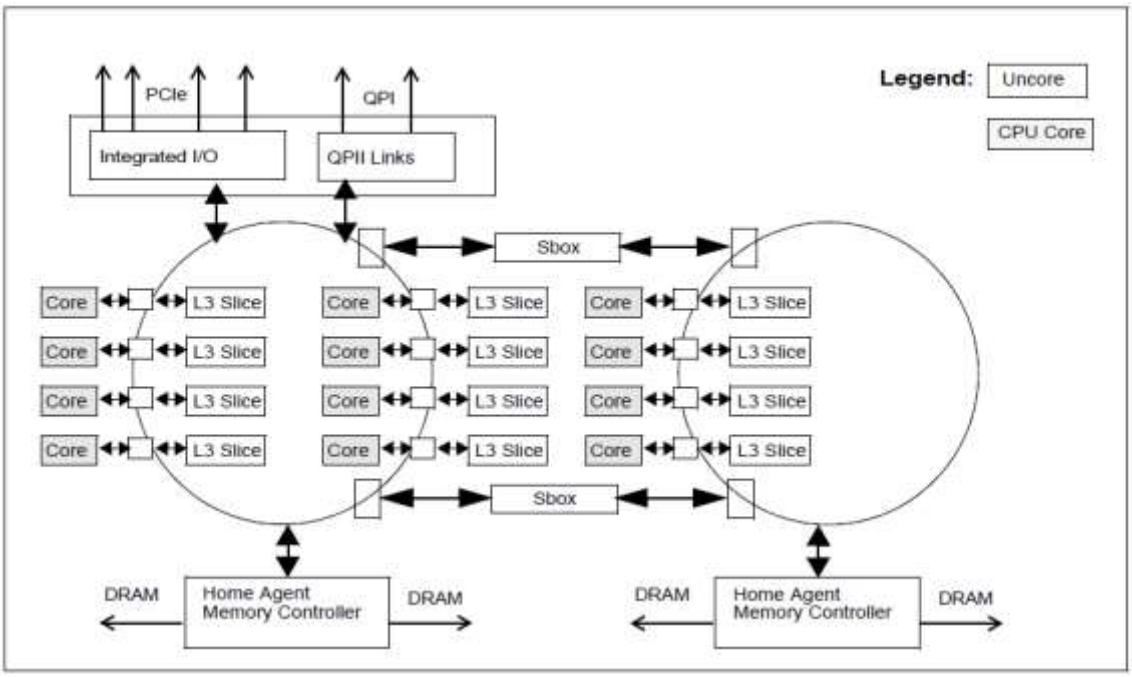


図 1-3 12 個のプロセッサー･コアをサポートする Haswell-E† マイクロアーキテクチャーの例

## 1.6 Broadwell† マイクロアーキテクチャー

インテル® Core™ M プロセッサーは、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。

- 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。
- ギャザー命令のスループットが大幅に向上しました。
- Broadwell† マイクロアーキテクチャーでは、PCLMULQDQ 命令が単一 uop 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 1-7 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 |

# 第 2 章
# 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 Sandy Bridge† 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† 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。

- 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)
  - 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

- 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

- 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
  - バッファー増加により、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

- システムオンチップのサポート
  - 第 2 世代インテル® Core™ プロセッサーでグラフィックス・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

- 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

## 2.1 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

図 2-4 に、インテル® マイクロアーキテクチャー Sandy Bridge † をベースとするプロセッサー・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。

- 命令をフェッチし、マイクロオペレーション (uop) 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 (uop) 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。
- サイクルあたり最大 6 つのマイクロオペレーション (uop) をディスパッチするアウトオブオーダー・スーパースケーラー実行エンジン。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 (uop) を「データフロー」の順序に並べ替えます。
- 検出された例外など、マイクロオペレーション (uop) 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。プロセッサーは、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

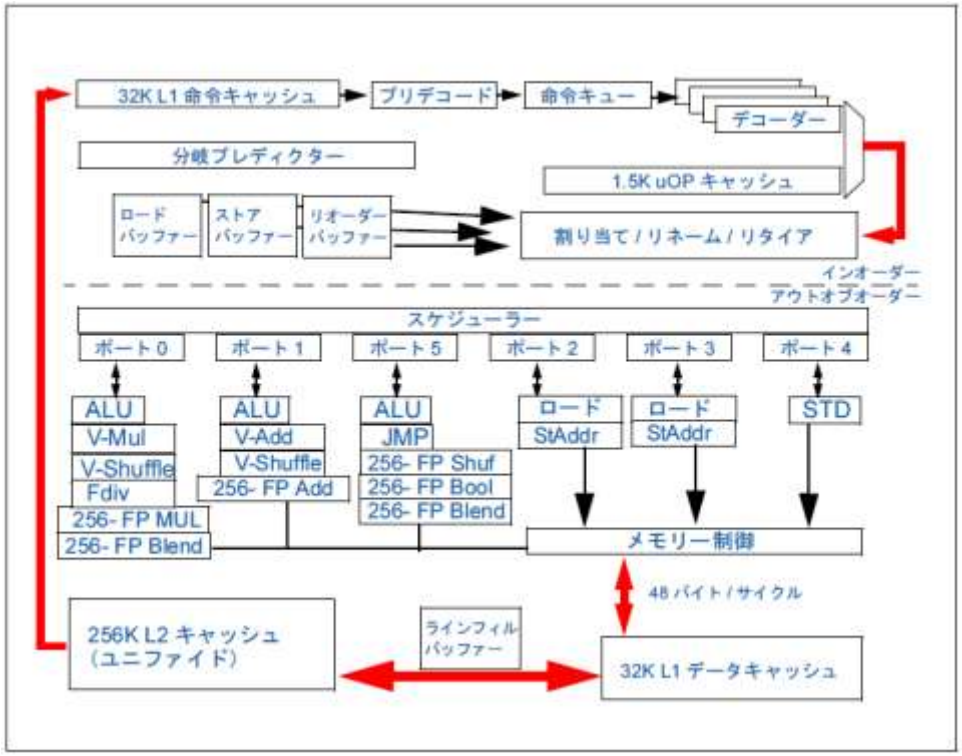


図 2-1 インテル® マイクロアーキテクチャー Sandy Bridge † のパイプラインの構造

2. コードに対応するマイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) を供給するようにフロントエンドに指示されます。プロセッサーは、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 (フォールト、トラップ) は、フォー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† をベースとする各プロセッサー・コアは、2 つの論理プロセッサーをサポートできます。

## 2.1.1 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 2-1 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 2-1 インテル® マイクロアーキテクチャー Sandy Bridge† 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® プロセッサーと同じデコード・レイテンシーおよび帯域幅を提供する。<br>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。 |

### 2.1.1.1 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、命令トランスレーション・ルックアサイド・バッファー (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 2-2 に、命令キャッシュと ITLB の前世代との比較を示します。

表 2-2 インテル® マイクロアーキテクチャー Sandy Bridge† 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ーは 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。以前のプロセッサーでは、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† では、以前のプロセッサーと比べペナルティーは改善されています。

- 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。
- 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。
- 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 (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop) は、マイクロオペレーション (uop) 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 (uop) よりも長い命令は、MSROM からマイクロオペレーション (uop) 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 (uop) です。MSROM からマイクロオペレーション (uop) 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 (uop) が単一の複雑なマイクロオペレーション (uop) に融合されます。複雑なマイクロオペレーション (uop) 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 (uop) 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 (uop) の例を以下に示します。

- 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。
- 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, OWORD PTR [RSP+40]
  – FADD DOUBLE PTR [RDI+RSI*8]
  – XOR RAX, QWORD PTR [RBP+32]

- 「ロードおよびジャンプ」形式のすべての命令。
  例:
  – JMP [RDI+200]
  – RET

- 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。

- 即値の追加が必要な場合。
  例:
  – CMP [RIP+400], 27
  – MOV [RIP+3000], 142

- 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 (uop) 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 (uop) 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。

- 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†: CMP, TEST
  – インテル® マイクロアーキテクチャー Sandy Bridge†: 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。

  - 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。
  - 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。

- 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。

### 2.1.1.2 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。

- 分岐予測ミスでのレイテンシーの減少
- 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 (uop) 供給帯域幅の拡大
- 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 (uop) 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 (uop) が取り出されます。これにより、マイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop) が格納される際のルールの一部です。

- ウェイのすべてのマイクロオペレーション (uop) 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。
- 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 (uop) をキャッシュに格納できます。
- 複数のマイクロオペレーション (uop) をウェイに分割することはできません。
- 1 ウェイあたり、最大 2 つの分岐が許容されます。

- MSROM を使用する命令があると、ウェイ全体が消費されます。
- 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 (uop) です。
- マイクロフュージョンされたマイクロオペレーション (uop) (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 (uop) として保持されます。
- 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 (uop) として保持されます。
- 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 (uop) 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 (uop) 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 (uop) のフェッチは次の分岐のマイクロオペレーション (uop) 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 (uop) 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。

### 2.1.1.3 分岐予測

分岐予測は分岐ターゲットを予測し、実際の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。

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

### 2.1.1.4 マイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop) を確実に供給します。

マイクロオペレーション (uop) 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 (uop) として示されます。マイクロオペレーション (uop) 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 (uop) (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 (uop) 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できることがあります。

**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 (uop) 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 (uop) 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 (uop) 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 (uop) キューの再実行の対象となります。

- 32 命令バイトの最大 8 つのチャンクフェッチ
- 最大 28 のマイクロオペレーション (uop) (28 以下の命令)
- すべてのマイクロオペレーション (uop) がデコード済み命令キャッシュにも存在する
- 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
- 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。

## 2.1.2 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 (uop) 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 2-1 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 (uop) 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 (uop) 間の不正な依存関係を排除し、マイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) を順番にリタイアさせ、フォールトおよび例外を処理します。

### 2.1.2.1 リネーマー

リネーマーは、図 E-4 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 (uop) をマイクロオペレーション (uop) 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 (uop) (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 (uop) 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。

- マイクロオペレーション (uop) 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。
- マイクロオペレーション (uop) にリソースを割り当てます。例えば、ロードバッファーやストアバッファーなど。
- マイクロオペレーション (uop) 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 (uop) 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。

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

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

- 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 (uop) 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 (uop) 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。

### 2.1.2.2 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 (uop) 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 (uop) 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 (uop) 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 (uop) をディスパッチするのかを選択します。

## 2.1.3 実行コア

実行コアはスーパースケーラーであり、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。

- 読み出しポートのストールの軽減
- ライトバックの競合および遅延の軽減
- 消費電力の軽減
- 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† 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。

- 汎用整数
- SIMD 整数および浮動小数点
- 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 (uop) をディスパッチできます。次の表は、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 2-3 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 |
|---|---|---|---|---|---|---|
| **整数** | ALU、Shift | ALU、Fast LEA、Slow LEA、MUL | Load_Addr、Store_addr | Load_Addr Store_addr | Store_data | ALU、Shift、Branch、Fast LEA |
| **SSE-Int、AVX-Int、MMX** | Mul、Shift、STTNI、Int-Div、128b-Mov | ALU、Shuf、Blend、128b-Mov | | | Store_data | ALU、Shuf、Shift、Blend、128b-Mov |
| **SSE-FP、AVX-FP_low** | Mul、Div、Blend、256b-Mov | Add、CVT | | | Store_data | Shuf、Blend、256b-Mov |
| **X87、AVX-FP_High** | 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 (uop) 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 (uop) 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。Sandy Bridge† マイクロアーキテクチャーでは、マイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) 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 (uop) 実行にバイパスできるかを示します。

表 2-4 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 |

## 2.1.4 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ーで共有されます。L2 キャッシュは命令とデータで共有されます。物理プロセッサー・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、命令トランスレーション・ルックアサイド・バッファー (ITLB)、データ・トランスレーション・ルックアサイド・バッファー (DTLB)、共有トランスレーション・ルックアサイド・バッファー (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 2-5 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 |

### 2.1.4.1 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、プロセッサーはキャッシュおよびメモリー内を検索します。表 E-13 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 2-6 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[1] | 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<br>60- ダーティーヒット | |

注意:

1. 表 2-4 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™ プロセッサー・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・エンプティーのいずれかの場合) 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、プロセッサーはまず 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、プロセッサー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。プロセッサーは、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、ストアバッファー内のエントリーが消費されます。ストアバッファーがいっぱいになると、新しいマイクロオペレーション (uop) 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。

## 2.1.5 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:

- 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 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ーの順序付け規則に従うことを確実にします。

表 2-7 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 |
| ロードバッファー | 64 エントリー | 48 エントリー |
| ストアバッファー | 36 エントリー | 32 エントリー |
| ライン・フィル・バッファー（LFB） | 10 エントリー | 10 エントリー |

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されていないアクセス、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 (uop) 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･バッファー (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を処理し、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･ミスを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、より下位のメモリー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。

- 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。
- 先行する分岐が解決される前に、投機実行できます。
- 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。

- あらゆるフォールトやトラップに対し投機的に対応すること。
- 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、2.1.3 節を参照してください。

次の表は、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 2-8 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;<br>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。

- 実行フェーズ: ストアバッファーをリニアアドレス、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。
- 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ーから 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、プロセッサー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。以下の表に、DTLB および STLB のエントリーの数を示します。

表 2-9 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。

- 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。
- 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。
- 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。

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

表 2-10 から表 2-13 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 2-10 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)

| | | 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 |
| 1 | 1 | F | | | | | | | | | | | | | | | |
| 2 | 1 | F | F | | | | | | | | | | | | | | |
| | 2 | F | N | | | | | | | | | | | | | | |

表 2-11 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 2-12 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 2-13 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アドレスが不明な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、先行するストアと同じアドレスからデータを取得します。ロードが実行された時点でアドレスが不明なストアに実際に依存する場合、競合が検出され、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ビ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。

- 16 バイト境界にまたがるロード。
- 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･ディスアンビ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す。

## 2.1.6 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 世代インテル® Core™ プロセッサー 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、メモリーアクセス順序、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。

プロセッサー・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リング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® プロセッサー 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ーをサポー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。

## 2.1.7 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、Sandy Bridge † マイクロアーキテクチャー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。

- 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。
- 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。
- 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。

- ライトバック・メモリーからのロードである。
- 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。
- パイプライン内でフェンスが発生していない。
- その他のロードミスがあまり発生していない。
- 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。

- **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。
- **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。

- 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。
- 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。

- 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。
- 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。

## 2.2 システム・エージェント

第 2 世代インテル® Core™ プロセッサー・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。

- 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。
- 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グ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)。
- DMI コントローラー: PCH チップセットに接続します。
- 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。
- 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™ プロセッサー 2xxx 番台のメモリー・コントローラーは、2 つの DDR チャネルをサポートし、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・レインテンシーに多少影響が生じます。

## 2.3 Ivy Bridge† マイクロアーキテクチャー

Ivy Bridge† マイクロアーキテクチャーをベースとした第 3 世代インテル® Core™ プロセッサーです。2.1 節から 2.2 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。

- 半精度浮動小数値との間の数値変換
- NIST SP 800-90A 準拠のハードウェア･ベースの乱数生成器
- 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参照してください:https://software.intel.com/en-us/articles/inteldigital-random-number-generator-drng-software-implementation-guide/ (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。

- ハードウェア･プリフェッチャーの拡張: Ivy Bridge† マイクロアーキテクチャーでは、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。
- 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。
- フロントエンドの拡張: Sandy Bridge† マイクロアーキテクチャーでは、マイクロオペレーション (uop) 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りなく、それぞれの論理プロセッサーへ 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。Ivy Bridge† マイクロアーキテクチャーでは、一方の論理プロセッサーがアクティブでない場合、プロセッサー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。
- いくつかの命令のレイテンシーとスループットが、Sandy Bridge† マイクロアーキテクチャー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。

# 第 3 章
# 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。

- **インテル® ワイド・ダイナミック・エグゼキューション:** 各プロセッサー・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 (uop) 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 (uop) のリタイアメント帯域幅
  - 高度な分岐予測
  - 関数やプロシージャーの開始/終了を効率化するスタックポインター追跡機能

- **インテル® アドバンスト・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

- **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

- **インテル® アドバンスト・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。

- **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。

- **インテル® アドバンスト・スマート・キャッシュ:** 2 つのプロセッサー・コア間で共有される最大 6MB の 2 次キャッシュ (クアッドコア・プロセッサーは最大 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。
- **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・フォワーディングなどのメモリー操作の処理を効率化します。

- **インテル® アドバンスト・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。

## 3.1 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。

- 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: uop) をアウトオブオーダー実行コアに供給します。
- アウトオブオーダー・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 (uop) を発行できます (表 3-2 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 (uop) をリオーダーします。
- 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 (uop) 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 プロセッサー X6800、インテル® Core™2 Duo プロセッサー、インテル® Xeon® プロセッサー 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 つのプロセッサー・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・プロセッサー、インテル® Core™2 Quad プロセッサー、インテル® Xeon® プロセッサー 3200/5300 番台には、4 つのプロセッサー・コアが実装されています。これらのクアッドコア・プロセッサーの各物理パッケージには 2 つのプロセッサー・ダイが搭載され、各ダイには 2 つのプロセッサー・コアが搭載されています。図 3-1 に、各コア内のサブシステムの機能を示します。

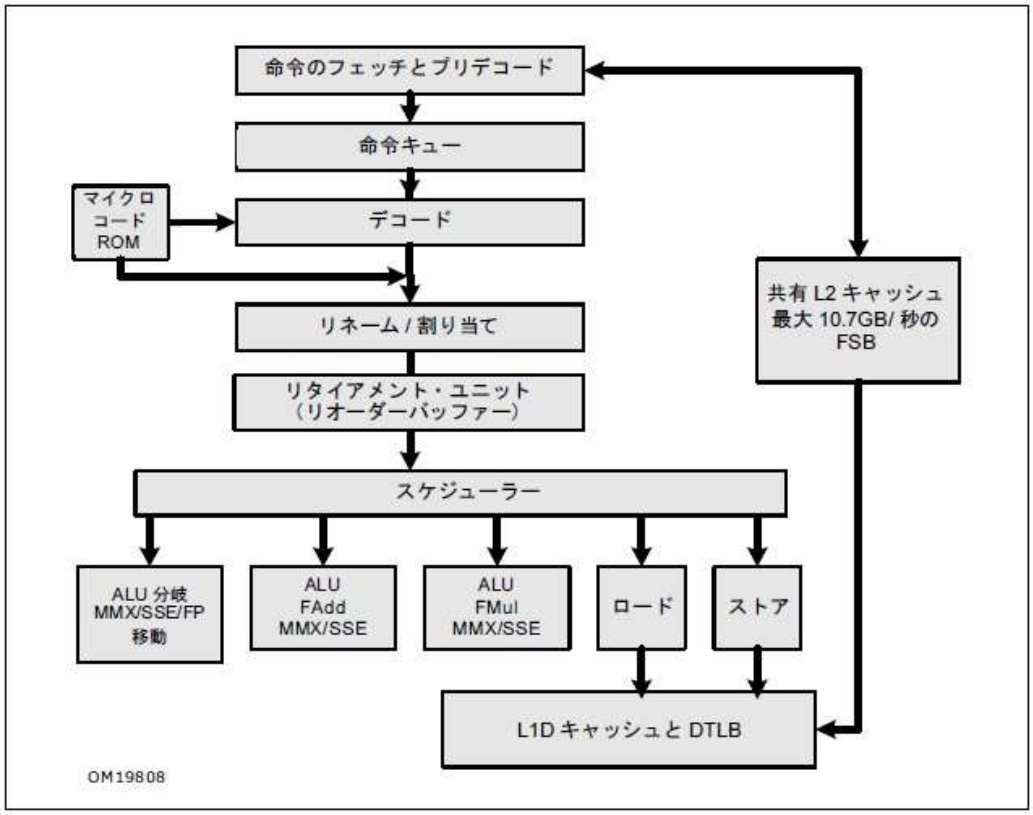


図 3-1 インテル® Core™ マイクロアーキテクチャーのパイプラインの構造

### 3.1.1 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:uop) ストリームの供給を、6 マイクロオペレーション (uop) の発行が可能なアウトオブオーダー・エンジンへ維持する必要があります。表 E-21 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 E-21 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) | • 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。 | • 投機実行を有効にします。<br>• 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」[1] 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。 |
| 命令フェッチユニット | • 実行される可能性の高い命令をプリフェッチします。<br>• 実行頻度の高い命令をキャッシュします。<br>• 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。 | • 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。<br>• 実行される分岐とアライメントされていない分岐先は、フェッチユニットの帯域幅全体に悪影響を及ぼします。 |
| 命令キューとデコードユニット | • 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。<br>• 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。<br>• 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。<br>• 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。 | • 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 (uop) への展開が必要となります。<br>• プリフィクス付きの命令はデコードが複雑になります。<br>• 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。 |

注意:

1. 実行されるとプロセッサーが判断しましたが、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。

#### 3.1.1.1 分岐予測ユニット

分岐予測によって、プロセッサーは、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。

- 16 エントリーのリターン・スタック・バッファー (RSB): BPU は RET 命令を正確に予測できます。
- 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。
- 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。

- 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。
- 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。
- 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。

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、『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 Volume 1』の 3.4 節「フロントエンドの最適化」を参照してください。

### 3.1.1.2 命令フェッチユニット

命令フェッチユニットは、命令トランスレーション・ルックアサイド・バッファー (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・バッファーへの 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すが、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。

分岐先がアライメントされていないと、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になります。

通常、パイプラインのほかの部分が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・バッファーから 16 バイトを受け入れて、以下のタスクを実行します。

- 命令長を判断します。
- 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。
- 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になります。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。

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

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 のペナルティーは発生しないので、LCP とはみなされません。

### 3.1.1.3 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ると、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。

- 実行される分岐による帯域幅の損失がない。
- アライメントされていない命令による帯域幅の損失がない。
- プリデコード・ステージを通過済みなので LCP のペナルティーがない。
- 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。

### 3.1.1.4 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 (uop) 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 (uop) 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 (uop) に対応可能であり、5 マイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) 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。

### 3.1.1.5 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ジャーの開始/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・ポインター・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 (uop) 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。

- 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 (uop) の命令なので、デコード帯域幅が向上します。
- 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。
- マイクロオペレーション (uop) 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。
- 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。

### 3.1.1.6 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 (uop) が単一の複雑なマイクロオペレーション (uop) に融合されます。複雑なマイクロオペレーション (uop) 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。

- 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。
- 複雑なマイクロオペレーション (uop) 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 (uop) 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。

## 3.1.2 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、スーパースケーラーであり、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ーがリソース (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。

- **リネーマー:** マイクロオペレーション (uop) 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。

- **リオーダーバッファー (ROB):** 各ステージで処理されたマイクロオペレーション (uop) を保持し、完了したマイクロオペレーション (uop) 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。
- **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 (uop) 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 (uop) 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 (uop) 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。

- リソース (例えば、ロードバッファーやストアバッファー) をマイクロオペレーション (uop) に割り当てます。
- マイクロオペレーション (uop) を適切な発行ポートにバインドします。
- マイクロオペレーション (uop) 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。
- 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 (uop) 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。

- **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(uop):** 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 (uop) 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。
- **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(uop):** この種のマイクロオペレーション (uop) 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 (uop) を同時に実行できます。
- **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。
- **オペランドが特定の制約に一致した場合の、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。
- **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・バッファー (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、3.1.3 節を参照してください。

### 3.1.2.1 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 (uop) を各ポートにディスパッチできます。表 E-22 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 CPUID 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 3-2 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 [1] |
|---|---|---|---|
| | シグネチャー = 06_0FH | シグネチャー = 06_17H | |
| 整数 ALU<br>整数 SIMD ALU<br>FP/SIMD/インテル® SSE2 ムーブと論理演算 | 1, 1<br>1, 1<br>1, 1 | 1, 1<br>1, 1<br>1, 1 | 64 ビット・モードの整数 MUL を含む。<br>発行ポート 0、ライトバック・ポート 0 |
| 単精度 FP MUL<br>倍精度 FP MUL | 4, 1<br>5, 1 | 4, 1<br>5, 1 | 発行ポート 0、ライトバック・ポート 0 |
| FP MUL (X87)<br>FP シャッフル<br>DIV/SQRT | 5, 2<br>1, 1 | 5, 2<br>1, 1 | 発行ポート 0、ライトバック・ポート 0<br>FP シャッフルは QW シャッフルを処理しない。 |
| 整数 ALU<br>整数 SIMD ALU<br>FP/SIMD/インテル® SSE2 ムーブと論理演算 | 1, 1<br>1, 1<br>1, 1 | 1, 1<br>1, 1<br>1, 1 | 64 ビット・モードの整数 MUL を除く。<br>発行ポート 1、ライトバック・ポート 1 |
| FP ADD<br>QW シャッフル | 3, 1<br>1, 1[2] | 3, 1<br>1, 1[3] | 発行ポート 1、ライトバック・ポート 1 |
| 整数ロード<br>FP ロード | 3, 1<br>4, 1 | 3, 1<br>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 [4] | 3, 1 | 3, 1 | 発行ポート 3 |
| ストアデータ [5] | | | 発行ポート 4 |
| 整数 ALU<br>整数 SIMD ALU<br>FP/SIMD/インテル® SSE2 ムーブと論理演算 | 1, 1<br>1, 1<br>1, 1 | 1, 1<br>1, 1<br>1, 1 | 発行ポート 5、ライトバック・ポート 5 |
| QW シャッフル<br>128 ビットのシャッフル/パック/アンパック | 1, 1[2]<br>2-4, 2-4[6] | 1, 1[3]<br>1-3, 1[7] | 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 (uop) 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 (uop) 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。

- SIMD 整数
- 通常の整数
- 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 E-6 を参照してください。

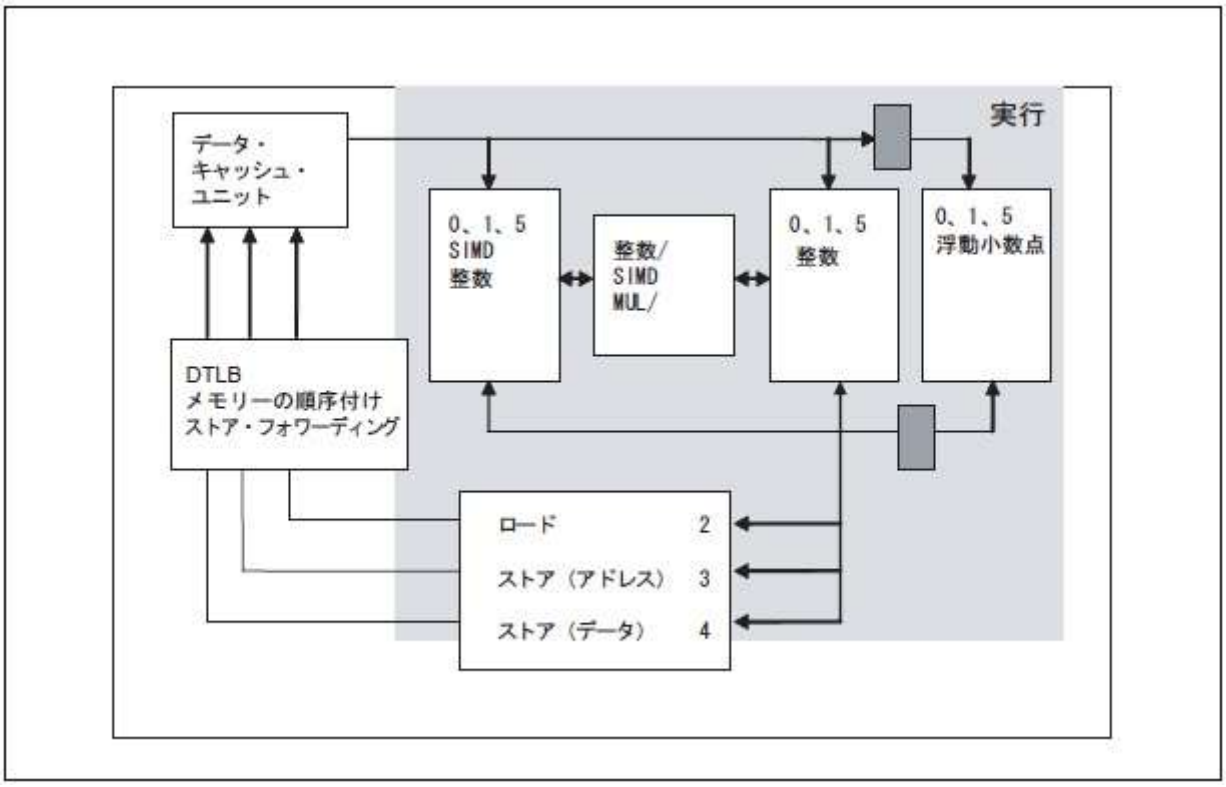


図 3-2 インテル® Core™ マイクロアーキテクチャーの実行コア

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 3-2 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。

## 3.1.3 インテル® アドバンスト・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。

- **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を処理して、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

- **データ・トランスレーション・ルックアサイド・バッファー(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。

- **ページ・ミス・ハンドラー(PMH)**
- **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。

- 128 ビットのロード操作とストア操作
- 1 次キャッシュからのデータ・プリフェッチ
- 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
- ストア・フォワーディング
- メモリー・ディスアンビゲーション (明確化)
- 8 個のフィル・バッファー・エントリー
- 20 個のストア・バッファー・エントリー
- メモリー操作のアウトオブオーダー実行
- 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、『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 Volume 1』の 3.6 節「メモリーアクセスの最適化」を参照してください。

### 3.1.3.1 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。

- 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。
- 先行する分岐が解決される前に、投機実行できます。
- 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。
- 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。

- あらゆるフォールトやトラップに対し投機的に対応すること。
- 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。

フォー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。

- **実行フェーズ** — ストア・フォワーディ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ーを準備します。ディスパッチ・ポート (ポート 3 および 4) が使用されます。
- **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・バッファー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ーから 1 次キャッシュに移動されます。

### 3.1.3.2 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。

- **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。プロセッサーは、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。
- **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。

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

- 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。
- 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。
- 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。

### 3.1.3.3 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 プロセッサーで導入されました。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。

- 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。
- 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。
- 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。

マルチコア・プロセッサーのそれぞれのコアのエントリーは、別々に処理されます。

### 3.1.3.4 ストア・フォワーディング

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-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。

- 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。
- 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。
- ロードは、キャッシュライン境界をまたぐことはできません。
- ロードは、8 バイト境界をまたぐことはできません。ただし、16 バイト・ロードは例外です。
- 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・フォワーディ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・フォワーディ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 (背景がグラデーションのもの) を図 3-3 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・フォワーディングです。

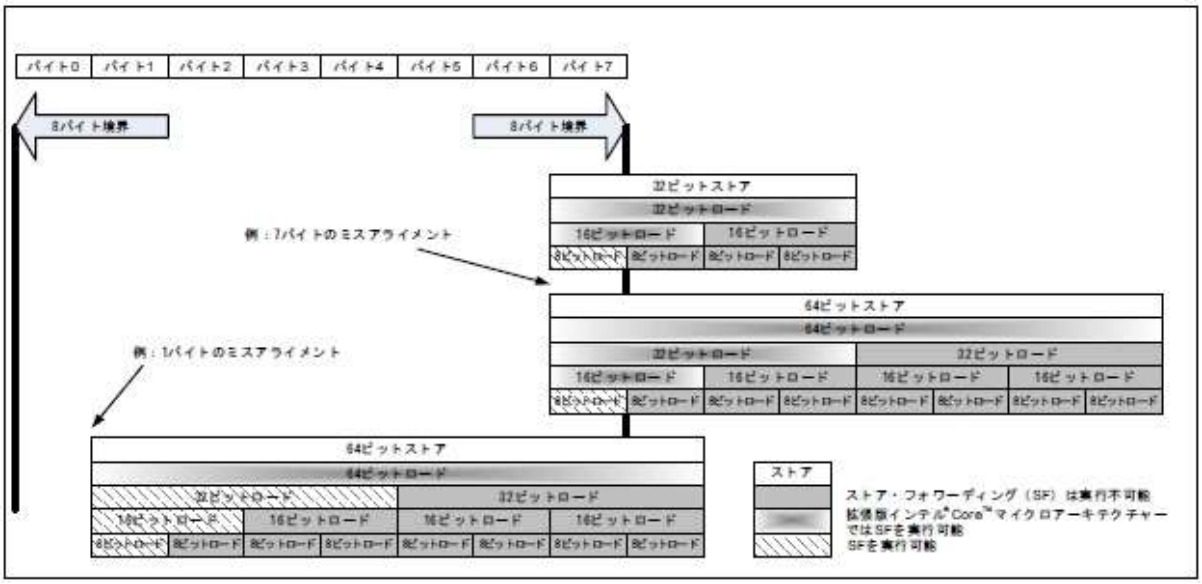


図 3-3 拡張版インテル® Core™ マイクロアーキテクチャーにおけるストア･フォワーディングの強化

### 3.1.3.5 メモリー･ディスアンビゲーション

2.1.5 節「L1D キャッシュ」の「メモリー･ディスアンビゲーション」を参照してください。

## 3.1.4 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 つのプロセッサー･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 3-4 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。

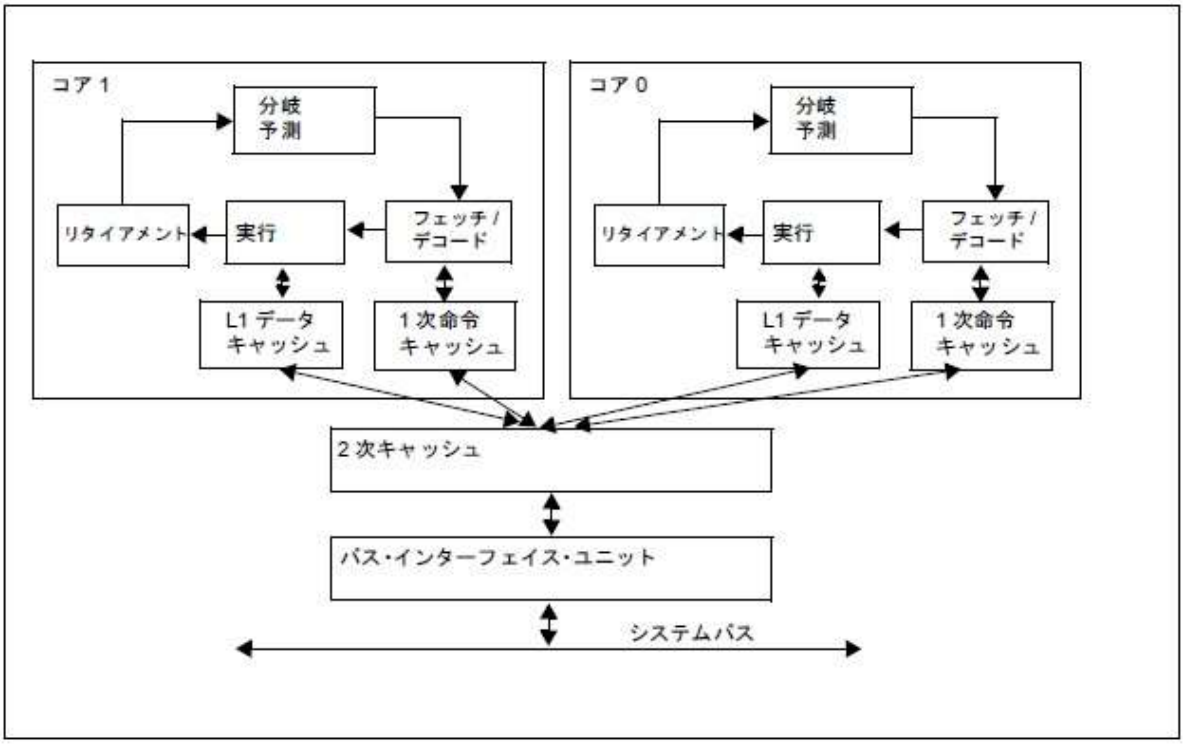


図 3-4 インテル® アドバンスト・スマート・キャッシュのアーキテクチャー

表 3-3 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 3-3 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ーのキャッシュ・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)[1] | 2、4 MB | 8 または 16 | 64 | 14[2] | 2 | ライトバック |
| 2 次 (共有 L2)[3] | 3、6MB | 12 または 24 | 64 | 15[2] | 2 | ライトバック |
| 3 次 [4] | 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 (CPUID 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 (CPUID シグネチャー DisplayFamily = 06H、DisplayModel = 1DH)。

### 3.1.4.1 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、プロセッサー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 3-4 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 3-4 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。

### 3.1.4.2 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、プロセッサーはまず、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。プロセッサーは、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 3-4 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。

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ることを強制することで、この問題を回避できます。

Knights Landing† 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。

# 第 4 章
# Nehalem† マイクロアーキテクチャー

Nehalem† マイクロアーキテクチャーは、インテル® Core™ i7 プロセッサーおよびインテル® Xeon® プロセッサー 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。

- **強化されたプロセッサー・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

- **インテル® ハイパースレッディング・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 (論理プロセッサー) を提供
  - 4 マイクロオペレーション (uop) 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

- **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・アクセスとスケー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

- **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

Westmere† マイクロアーキテクチャーは、Nehalem† マイクロアーキテクチャーの 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。

## 4.1 マイクロアーキテクチャー・パイプライン

Nehalem† マイクロアーキテクチャーは、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 (uop) 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 4-1 に、インテル® Core™ i7 プロセッサーで実装されている Nehalem† マイクロアーキテクチャーのパイプラインの基本構成要素を示します。ただし、図 4-1 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。

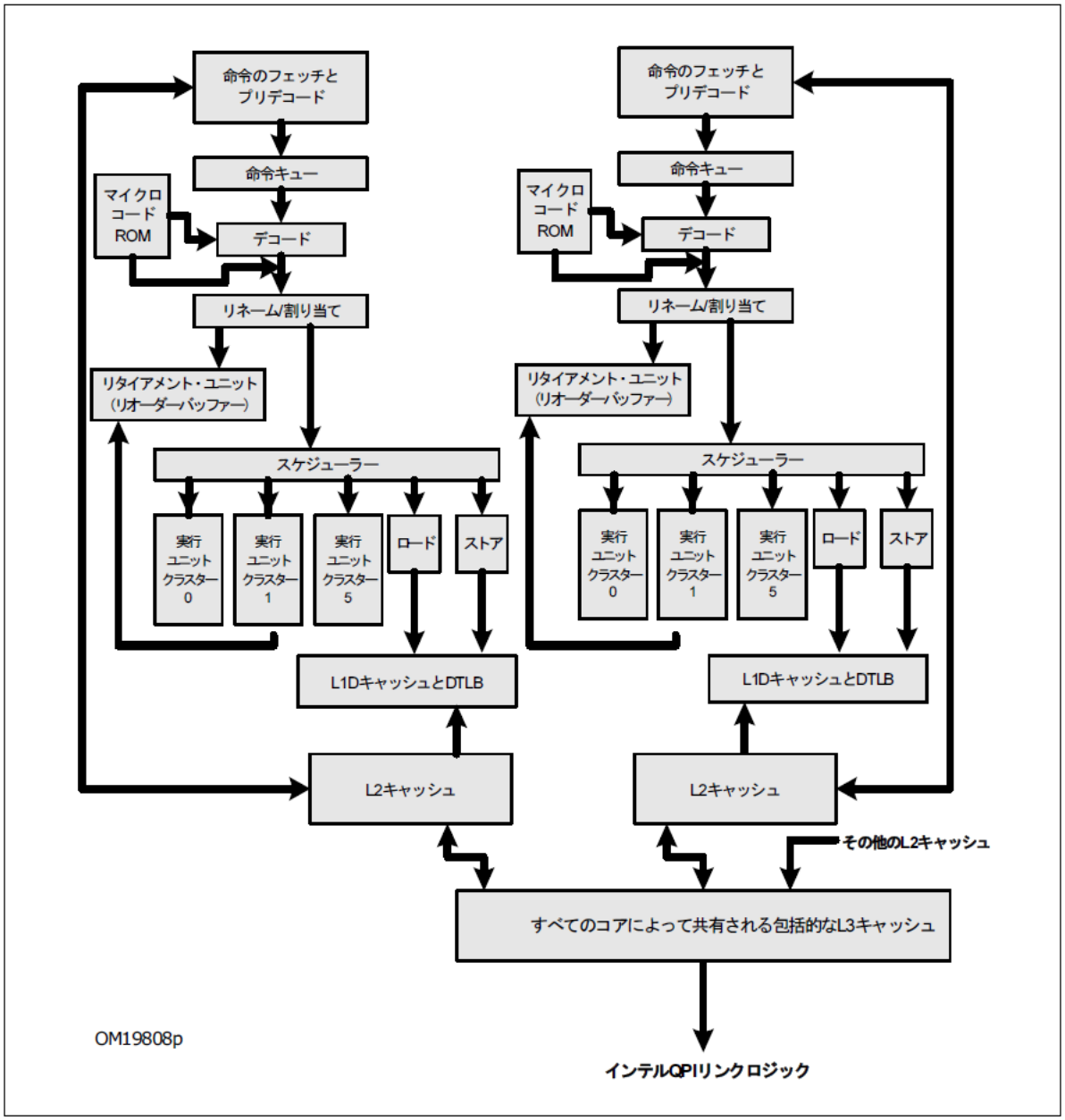


図 4-1 Nehalem† マイクロアーキテクチャーのパイプラインの構造

Nehalem† マイクロアーキテクチャー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 プロセッサー・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ー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 スループッ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 (uop) を 6 つの発行ポートにディスパッチできます (図 4-1 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 4-1 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 (uop) をリタイアできます。

## 4.2 フロントエンドの概要

図 4-2 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。

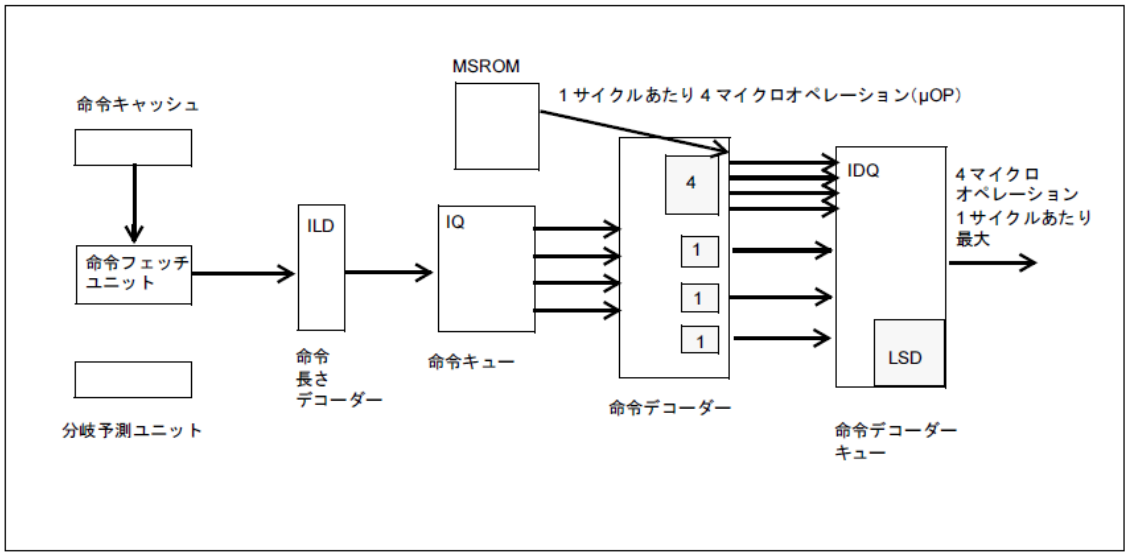


図 4-2 Nehalem† マイクロアーキテクチャーのフロントエンド

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 (uop) 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 (uop) 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 (uop) 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ド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、フロンド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 (uop) 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。

Nehalem† マイクロアーキテクチャーでは、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。

- 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

- 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)。

- 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

- Nehalem† マイクロアーキテクチャーでは、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ーが拡大し、分岐予測の精度が向上しました。リターン・スタック・バッファー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 (uop) ストリームが順方向で処理を開始できます。

## 4.3 実行エンジン

IDQ (図 4-2) は、マイクロオペレーション (uop) 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 4-1) 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 (パイプライ中で同時に進行する) のマイクロオペレーション (uop) を最大 128 個サポートします。各マイクロオペレーション (uop) は、リオーダーバッファー (ROB) 内のエントリー、リザベーション・ステーション (RS) 内のエントリー、およびロード/ストアバッファー (メモリー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。

アロケーターは、インフライトの各マイクロオペレーション (uop) のレジスター・ファイル・エントリーをリネームします。マイクロオペレーション (uop) 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 エントリーに拡大されました (前世代では 32 エントリー)。マイクロオペレーション (uop) 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 uop 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 (uop) 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 (uop) 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 uop にバイパス・ネットワーク経由で転送されます。Nehalem† マイクロアーキテクチャーは、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 (uop) から消費 (consumer) する uop 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 4-1 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 (uop) と消費 uop の間のバイパス遅延を示します。

表 4-1 生産側と消費側の uop 間のバイパス遅延 (サイクル)

| | FP | 整数 | SIMD |
|---|---|---|---|
| FP | 0 | 2 | 2 |
| 整数 | 2 | 0 | 1 |
| SIMD | 2 | 1 | 0 |

## 4.3.1 発行ポートと実行ユニット

表 4-2 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 4-2 Nehalem† マイクロアーキテクチャーの発行ポート

| 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<br>整数シフト | 1<br>1 | 1<br>1 | 整数 | |
| ポート 0 | 整数 SIMD ALU<br>整数 SIMD シャッフル | 1<br>1 | 1<br>1 | SIMD | |
| ポート 0 | 単精度（SP）FP MUL<br>倍精度 FP MUL<br>FP MUL（X87）<br>FP/SIMD/SSE2 ムーブと論理演算<br>FP シャッフル<br>DIV/SQRT | 4<br>5<br><br>5<br>1<br><br>1 | 1<br>1<br><br>1<br>1<br><br>1 | FP | |
| ポート 1 | 整数 ALU<br>整数 LEA<br>整数 MUL | 1<br>1<br>3 | 1<br>1<br>1 | 整数 | |
| ポート 1 | 整数 SIMD MUL<br>整数 SIMD シフト<br>PSAD<br>文字列比較 | 1<br>1<br>3 | 1<br>1<br>1 | SIMD | |
| 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<br>整数シフト<br>ジャンプ | 1<br>1<br>1 | 1<br>1<br>1 | 整数 | |
| ポート 5 | 整数 SIMD ALU<br>整数 SIMD シャッフル | 1<br>1 | 1<br>1 | SIMD | |
| ポート 5 | FP/SIMD/SSE2 ムーブと論理演算 | 1 | 1 | FP | |

## 4.4 キャッシュとメモリー・サブシステム

Nehalem† マイクロアーキテクチャーでは、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 4-1 を参照)。それぞれの物理プロセッサーは、複数のプロセッサー・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 プロセッサーのアンコアは、物理プロセッサー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、プロセッサー・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 4-3 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、プロセッサーとアンコア・サブシステムとの周波数比に応じて異なります。

表 4-3 インテル® Core™ i7 プロセッサーのキャッシュ・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[1] | 状況により異なる | ライトバック |
| L3 (共有)[2] | 8MB | 16 | 64 | 35-40+[2] | 状況により異なる | ライトバック |

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。

Nehalem† マイクロアーキテクチャーは、2 レベルのトランスレーション・ルックアサイド・バッファー (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。

- 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)
- ラージページの DTLB0:32 エントリー
- 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。

## 4.4.1 ロード操作とストア操作の強化

Nehalem† マイクロアーキテクチャーのメモリークラスターは、以下の拡張機能によってメモリー操作の高速化を図ります。

- 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
- ロード操作とストア操作向けの深いバッファー:48 個のロードバッファー、32 個のストアバッファー、10 個のフィルバッファー
- アライメントされていないメモリーの高速アクセスと、堅牢なメモリー・アライメント・ハザード処理
- 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
- 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

## 4.4.2 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 4-4 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 4-4 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 プロセッサー | 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 4-4 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。

Nehalem† マイクロアーキテクチャーでは、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。

## 4.4.3 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。

- 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
- 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
- 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。Nehalem† マイクロアーキテクチャー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 4-3 に、16 バイト・ストア操作おけるストア・フォワーディング・シナリオを示します。

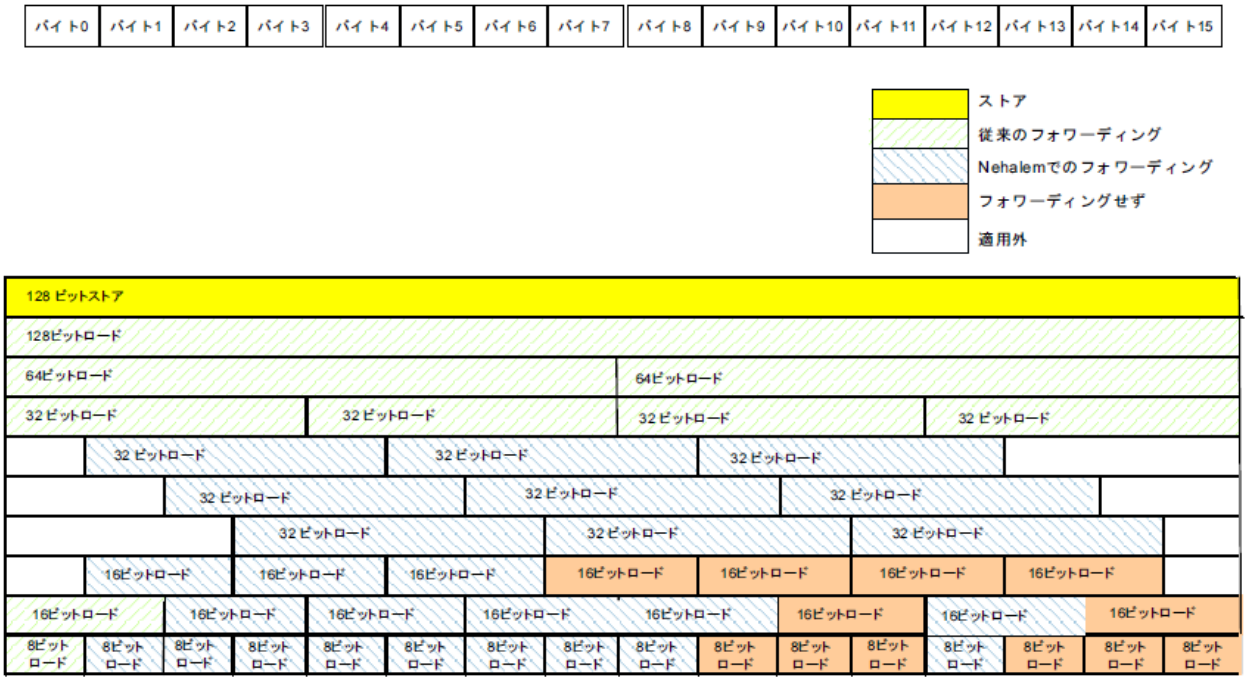


図 4-3 16 バイト・ストア操作のストア・フォワーディング・シナリオ

Nehalem† マイクロアーキテクチャーでは、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 4-4 に、8 バイト以下のストア操作のシナリオを示します。

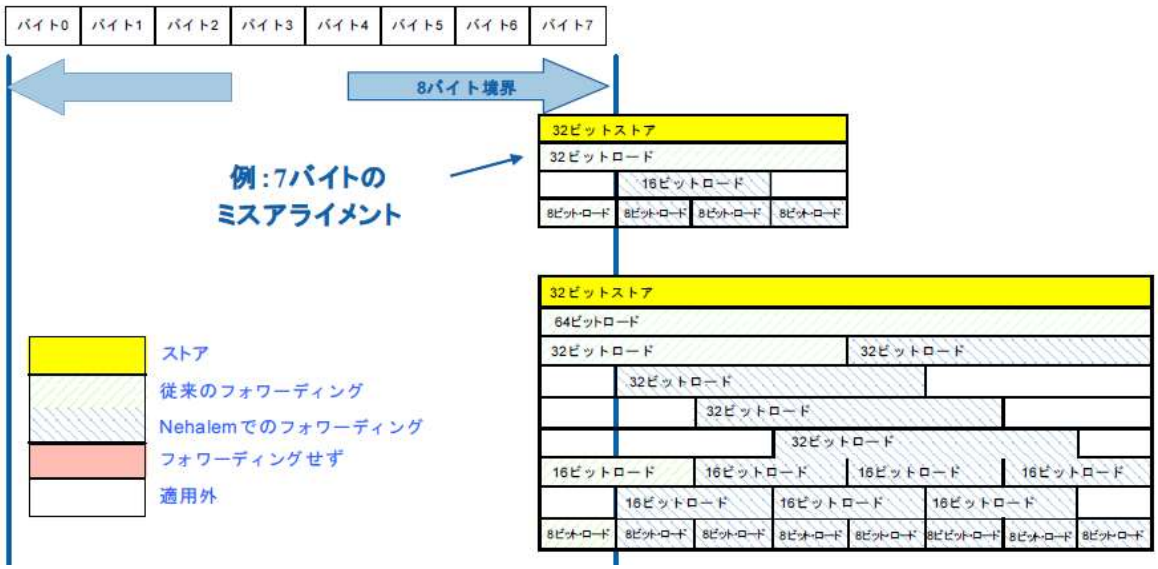


図 4-4 Nehalem† マイクロアーキテクチャーにおけるストア･フォワーディングの強化

## 4.5 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 MOVS/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ーは、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。

- 短い文字列 (ECX <= 12): REP MOVSW/MOVSD/MOVSQ のレイテンシーは約 20 サイクルです。
- 高速文字列 (ECX >= 76: REP MOVSB を除く): プロセッサーは、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。

- 中間の文字列長：REP MOVSW/MOVSD/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。

Nehalem† マイクロアーキテクチャーでは、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。

- 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
- データ転送スループットが前世代よりも向上しました

- 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、Nehalem† マイクロアーキテクチャーでは、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

## 4.6 システム・ソフトウェアの強化

Nehalem† マイクロアーキテクチャーでは、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。Nehalem† マイクロアーキテクチャー・ベース・プロセッサーでは、VMX 遷移のレイテンシーが軽減されています。

### 4.6.1 電力消費の効率化

Nehalem† マイクロアーキテクチャーは、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。Nehalem† マイクロアーキテクチャーではプロセッサー固有の 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。

### 4.6.2 Nehalem† マイクロアーキテクチャーにおけるインテル® ハイパースレッディング・テクノロジーのサポート

Nehalem† マイクロアーキテクチャーは、ハイパースレッディング (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ーが各コアの実行/キャッシュリソースの大半を共有します。Nehalem† マイクロアーキテクチャーのインテル® HT テクノロジーは、Intel NetBurst® マイクロアーキテクチャー・ベースの旧世代のインテル® HT テクノロジーと以下の点が異なります。

- Nehalem† マイクロアーキテクチャーでは、4 マイクロオペレーション (uop) 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。
- Nehalem† マイクロアーキテクチャー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 プロセッサーでは最大 25.6GB/秒のピークメモリー帯域幅を提供します。
- 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。
  - インテル® 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・バッファー、ラージページ ITLB
  - インテル® HT テクノロジー操作向けに分割されるリソース: ロードバッファー、ストアバッファー、リオーダーバッファー、スモールページ ITLB は、2 つの論理プロセッサー間で静的に割り当てられます
  - インテル® 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、フィルバッファー、DTLB0 と STLB の両方
  - インテル® 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ー間で交互に実行されます
  - インテル® HT テクノロジーで管理されないリソース: 実行ユニット

# 第 5 章
# Knights Landing† マイクロアーキテクチャーの最適化

インテル® Xeon Phi™ プロセッサー 7200/5200/3200 製品ファミリーは、Knights Landing† 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† 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† マイクロアーキテクチャー・ベースのプロセッサーは、『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。

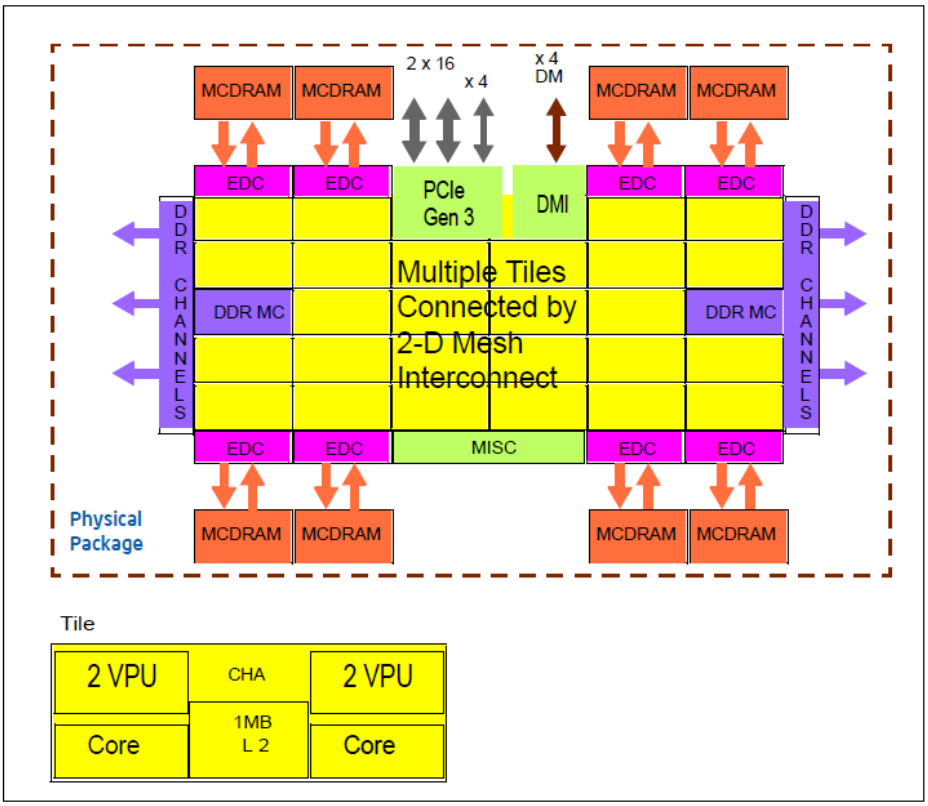


図 5-1 Knights Landing† マイクロアーキテクチャーのタイルメッシュ・トポロジー

Knights Landing† 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ーおよびコプロセッサー製品ファミリーとして設計されました。Knights Landing† マイクロアーキテクチャー・ベースのインテル® Xeon Phi™ プロセッサーは以下を含みます。

- 多数のタイル。
- 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。
- IA 互換プロセッサー・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 5-1 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 (プロセッサー・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 メモリーチャネルを示しています。

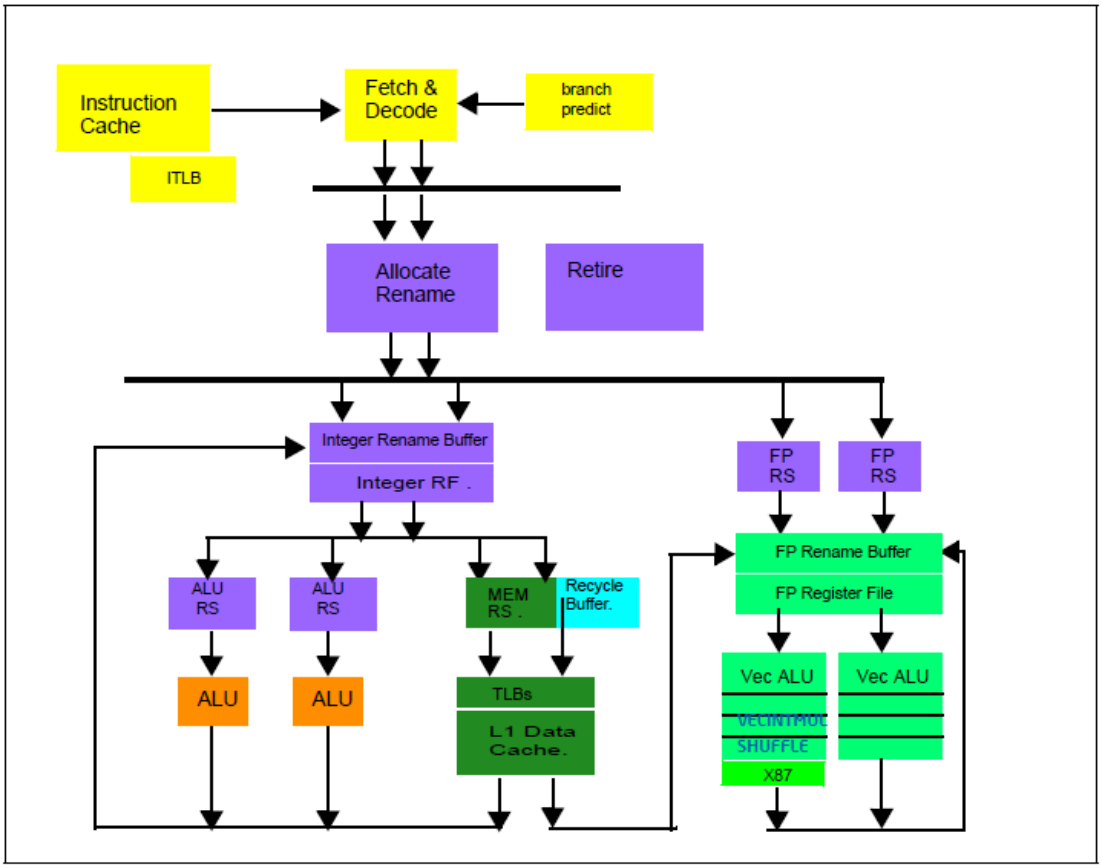


図 5-2 Knights Landing† マイクロアーキテクチャーのプロセッサー･コア･パイプラインの機能

図 5-1 はまた、各タイルの構成を示します。

- コアあたり 4 つの論理プロセッサーのインテル® ハイパースレッディング･テクノロジーをサポートする 2 つのアウトオブオーダー IA プロセッサー･コア。
- タイル内の 2 つのプロセッサー･コアで共有される 1M バイトの L2 キャッシュ。
- 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)。
- 各プロセッサー･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 5-2 は、プロセッサー･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† マイクロアーキテクチャーのプロセッサー･コアは、次の機能を持ちます。

- 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 uop (uop) 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ベーション･ステーション (72 エントリー) で分配されます。

- 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。
- 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。

- 各プロセッサー・コアは、インテル® ハイパースレッディング・テクノロジーにより 4 つの論理プロセッサーをサポートします。
- 2 つのプロセッサー・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。

## 5.1 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 uop を供給できます。命令が複数の uop 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 uop 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。uop 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リザベーション・ステーションによって扱われます。

### 5.1.1 アウトオブオーダー・エンジン

リオーダーバッファー (ROB) は、72 uop の深さを持ちます。16 個のストアバッファー (アドレスとデータの両方向け) を備えています。uop の分散スケジューリングは以下を含みます (図 5-2 を参照)。

- 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。
- 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 uop をディスパッチします。
- 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・バッファーは、論理プロセッサーごとにハードウェアでパーティション化されています (プロセッサー・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ー数に依存します)。リソースのハードウェア・パーティションは、論理プロセッサーがウェイクアップおよびスリープするたびに変化します。ストア・アドレス・バッファーは、論理プロセッサーごとに 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ー間で共有されます。

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 uop 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 uop 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 uop 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† 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 5-1 を参照)。

表 5-1 Knights Landing† 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 5-2 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。

- すべての x87 数値操作。
- FP 除算または平方根。
- インテル® AVX-512ER。
- ベクトル permute/shuffle 操作。
- ベクトルから整数への移動。
- インテル® AVX-512CD 競合命令。
- インテル® AES-NI。
- 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 5-2 Knights Landing† 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[1] | 25 | ~20 |
| DIVSD/SQRTSD[1] | 40 | ~33 |
| DIVP*/SQRTP*[1] | 38 | ~10 |
| Shuffle/Permute (1 source operand)[1] | 2 | 1 |
| Shuffle/Permute (2 source operands)[1] | 3 | 2 |
| Convert (from/to same width)[1] | 2 | 1 |
| Convert (from/to different width)[1] | 6 | 5 |
| Common x87/MMX Instructions[1] | 6 | 1 |

注意:

1. 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† 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 uop 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 uop 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。

- POP: 整数ロードデータ + ESP 更新
- PUSH: 整数ストアデータ + ESP 更新
- INC: レジスターへの加算 + パーシャルフラグの更新
- Gather: 2 つの VPU uop
- RET: JMP + ESP 更新
- 3 ソースの CALL、DEC、LEA

表 5-3 に Knights Landing† マイクロアーキテクチャーのキャシュリソースの特性リストを示します。

表 5-3 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)[1] |
| 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. µTLB 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、バッファー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。

## 5.1.2 アンタイル

Knights Landing† マイクロアーキテクチャーでは、多くのタイルがメッシュ・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 5-1 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「アンタイル」と呼ばれます。各メッシュストップは、タイ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† 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† 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。

- キャシュモード: 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。
- 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。
- 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。

- 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、メッシュのどこででも行うことができます。
- 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。
- サブ NUMA クラスター (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、Knighjts Landing† 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† 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、ほか (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。

## 5.2 Knights Landing† 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』を参照してください。Knights Landing† マイクロアーキテクチャーをベースとするインテル® Xeon Phi™ プロセッサー 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† マイクロアーキテクチャーをベースのプロセッサーでは、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™ コプロセッサー 7100/5100/3100 製品ファミリーでは、インテル® AVX-512、インテル® AVX2 またインテル® AVX 命令はサポートされていません。

### 5.2.1 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 5-1 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[]<br>kxnor k1,k0, k0; prepare mask<br>vpgatherdd zmm2{k1}, [rax+zmm0*4]<br>; compute sequence using vector register | vmovdqu ymm0, [rsp+0x1000] ; load half of index vector<br>vmovdqu ymm3, [rsp+0x1020] ; 2nd half of indirect[]<br>vpcmpeqdd ymm4, ymm4, ymm4 ; prepare mask<br>vmovdqa ymm1, ymm4<br>vpgatherdd ymm2, [rax+ymm0*4], ymm1<br>vpgatherdd ymm5, [rax+ymm3*4], ymm4<br>; 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 へルー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™ コプロセッサー (例 5-2 では「Previous Generation (以前の世代)」と表現されています) とは異なります。

例 5-2 インテル® AVX-512F と KNC† における Gather の比較

| AVX-512F | Previous Generation Equivalent Sequence |
|---|---|
| vmovdqu zmm0, [rsp+0x1000] ; load indirect[]<br>kxnor k1,k0, k0; prepare mask<br>vpgatherdd zmm2{k1}, [rax+zmm0*4]<br>; compute sequence using vector register | vmovdqu zmm0, [rsp+0x1000] ; load indirect[]<br>kxnor k1,k1 ; prepare mask<br>g_loop: ; verify gathered elements are complete<br>vpgatherdd zmm2{k1}, [rax+zmm0*4]<br>jknzd k1, g_loop ; gather latency exposure<br>; compute sequence using vector register |

### 5.2.2 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。例 5-3 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 5-3 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
vrcp28ss       xmm19, xmm18, xmm18
vgetexpss      xmm16, xmm16, [rsp+0x8]
vgetexpss      xmm17, xmm17, [rsp+0x10]
vsubss         xmm22, xmm16, xmm17
vmulss         xmm21{rne-sae}, xmm19, xmm20
vfnmadd231ss   xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss    xmm21, xmm19, xmm20
vscalefss      xmm0, xmm21, xmm22
```

## 5.3 インテル® AVX-512CD 命令の利用

インテル® AVX-512 競合検出命令の詳細については、『インテル® 64 および IA-32 アーキテクチャー最適化リファレンス・マニュアル Volume 1』の 18.16 節「競合検出」を参照してください。

### 5.3.1 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† マイクロアーキテクチャーは、それぞれのプロセッサー・コアで 4 つの論理プロセッサーをサポートします。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。

- 論理プロセッサー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。
- プロセッサー・コアで複数の論理プロセッサー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リングの効率が制限される可能性があります。

Knights Landing† 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ー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。プロセッサー・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。

## 5.3.2 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。

- 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、レジスターへロードし、レジスターバージョンの PUSH と CALL を実行します。表 5-4 にいくつかの例を示します。
- 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・ディスプレース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。
- 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。Knights Landing† 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。
- 各サイクルでデコード可能な分岐の最大数は 1 です。

## 5.3.3 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 (uop) フローにデコードするため、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 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 5-4 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 5-4 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 |

## 5.3.4 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の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。

## 5.4 整数実行に関する考察

### 5.4.1 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 (uop) 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† 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。

### 5.4.2 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。

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べきです。

## 5.5 FP およびベクトル実行の最適化

### 5.5.1 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† 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 5-4 VCOMIS* を VCMPSS/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 uop 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† 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 のスループ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 uop 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 uop 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 5-5)。

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 5-6 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 5-6 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 のスループットで、6 サイクルのレイテンシーを達成します。例 5-6 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 5-5 水平リダクションのソフトウェア･シーケンスを使用

| | |
|---|---|
| vextractf64x4 ymm1, zmm6, 1; reduction of 16 elements<br>vaddps ymm1, ymm6, ymm1<br>vpermpd ymm4, ymm1,0xff<br>vpermpd ymm5, ymm1,0xaa<br>vpermpd ymm3, ymm1,0x44<br>vaddps xmm1, xmm1, xmm4<br>vaddps xmm3, xmm5, xmm3<br>vaddps xmm3, xmm1, xmm3<br>vpsrlq xmm1, xmm3, 32<br>vaddss xmm3, xmm1, xmm3 | vextractf64x4 ymm1, zmm6, 1; reduction of 8 elements<br>vaddps ymm1, ymm6, ymm1<br>valignq ymm4, ymm1,0x3<br>valignq ymm5, ymm1,0x2<br>valignq ymm3, ymm1,0x1<br>vaddsd ymm1, ymm1, ymm4<br>vaddsd ymm3, ymm5, ymm3<br>vaddsd ymm3, ymm1, ymm3 |

例 5-6 Knights Landing† 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}
```

## 5.5.2 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 以前の世代の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、以前の世代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ーアクセス、並べ替えのコストの違い、以前の世代の制限などにより) Knights Landing† 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。

以前の世代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。

### 5.5.3 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る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 5-5 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 5-5 Knights Landing† 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。

- 反復ごとに 2 つのロード (a[i]、b[i])。
- 反復ごとに 1 つの FMA。
- 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 * 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。

- 反復ごとに 2 つのロード (indir[i]、b[i])。
- 反復ごとに 1 つの FMA。
- 反復ごとに 1 つのストア。
- 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* 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]]; }
```

- 反復ごとに 1 つのロード (indi[i])。
- 反復ごとに 1 つの FMA。
- 反復ごとに 1 つのストア。
- 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*2 + 3)* Ceiling(N/8) = 33* 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*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。

除算を伴うスキャ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)*N です。ベクトル化のコストは、(15+20+3)*Ceiling(N/8) です。N > 2 の場合ベクトル化の利点があります。

スキャ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*N で、ベクトルコードのコストは (15+20+1)*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 * numneigh + 23 * numneigh * percent_rsq_less_than_cutforcesq です。そしてベクトルのコストは、(52 + 23) * ceiling(numneigh / 8) + 3 * 30 です。(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 floor(N/8) 回実行され、リマインダー・ループが (N mod 8) 回実行されることになります。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、ceiling の変わりに 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 5-5 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。

## 5.6 メモリー最適化

### 5.6.1 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 へロードする場合、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プットは、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。

ギャザーとスキャ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ャッターする方が高速です。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 ビットの値をインサートする方が高速です。

### 5.6.2 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) はプロセッサー・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ーを使用します。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ーにかかわりなく、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ぐ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。

## 5.6.3 ソフトウェア・プリフェッチ

Knights Landing† 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 以前の世代のインオーダー・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。

5.6.2 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 (フィルバッファー) 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。

### 5.6.3.1 メモリー実行クラスター

MEC が uop 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 uop 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 5-7 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。例 5-7 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 5-7 MEC 向けメモリー命令の順番

| | | | |
|---|---|---|---|
| movq | r15, [rsp+0x40] ; cycle N (load &a[0]) | movq | r15, [rsp+0x40] ; cycle N (load &a[0]) |
| movq | r14, [rsp+0x48] ; cycle N+1 (load &b[0]) | vmovups | zmm1, [r15+rax*8] ; executes in cycle N+4 |
| vmovups | zmm1, [r15+rax*8] ; executes in cycle N+4 | movq | r15, [rsp+0x48] ; cycle N+4 (load &b[0]) |
| vmovups | zmm2, [r14+rax*8] ; cycle N+5 | vmovups | zmm2, [r15+rax*8] ; cycle N+8 |

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。

### 5.6.4 ストア・フォワーディング

Knights Landing† 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† 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。

### 5.6.5 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。

- 反復 N の a[i] と b[i] が衝突
- 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。

- 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。
- カスタム化された malloc() ルーチンを使用します。
- 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。

### 5.6.6 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† 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。5.1.2 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。

### 5.6.7 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。そうでない場合、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† マイクロアーキテクチャーをターゲットとします。

### 5.6.8 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† 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ることを強制することで、この問題を回避できます。

Knights Landing† 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。

# 第 6 章
# 旧世代の 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

## 6.1 概要

45nm プロセスの Intel Atom® プロセッサーは、Intel Atom® マイクロアーキテクチャーをベースにしています。同じマイクロアーキテクチャーは、32nm プロセスの Intel Atom® プロセッサーでも採用されています。この章では、Intel Atom® マイクロアーキテクチャーの概要を説明し、このマイクロアーキテクチャー・ベースのプロセッサーをターゲットとするソフトウェア向けの固有のコーディング手法を示します。Intel Atom® プロセッサーの主な機能は、低消費電力と以下を含む効率良いパフォーマンスをサポートすることです。

- 拡張版 Intel SpeedStep® テクノロジー：ワークロード実行中にプロセッサーが周波数/電圧レベルを低い状態に遷移できるようにオペレーティング・システム (OS) をプログラムできます。
- プロセッサー内のキャッシュと他のサブシステムへの電力を切断することによって、スタティック消費電力を削減するディープ・パワー・ダウンをサポートします。
- インテル® ハイパースレッディング・テクノロジーは、マルチタスクとマルチスレッド・ワークロード向けに 2 つの論理プロセッサーを提供します。
- 1 つの命令で複数のデータ用を処理する拡張 (インテル® SSE3 とインテル® SSSE3) をサポートします。
- インテル® 64 および IA-32 アーキテクチャーをサポートします。

Intel Atom® マイクロアーキテクチャーは、小型フォームファクターや熱的に制約された環境における消費電力の許容範囲内で、最新のワークロードのパフォーマンス要件を満たすように設計されています。

## 6.2 Intel Atom® マイクロアーキテクチャー

Intel Atom® マイクロアーキテクチャーは、2 マイクロオペレーション (uop) の発行が可能なインテル® ハイパースレッディング・テクノロジー対応のインオーダー・パイプラインによって、効率良いパフォーマンスと低消費電力の動作を実現しています。インオーダー・パイプラインは、メモリーオペランドを持つ IA-32 命令を複数のマイクロオペレーション (uop) ではなく単一のパイプライン操作として扱う点が、アウトオブオーダー・パイプラインとは異なります。

図 6-1 に、Intel Atom® マイクロアーキテクチャーのパイプラインの基本ブロック図を示します。

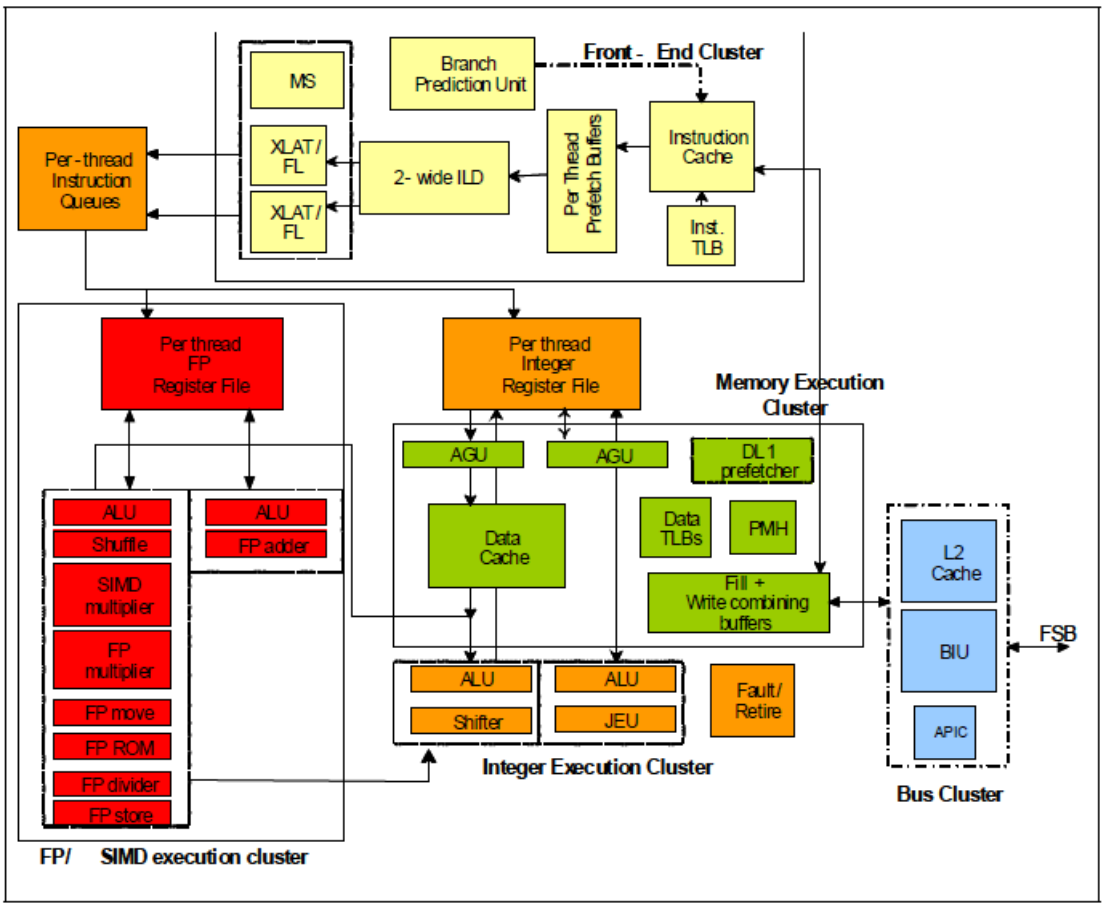


図 6-1 Intel Atom® マイクロアーキテクチャーのパイプライン

フロントエンドは、以下で構成される省電力パイプラインを備えています。

- 32KB の 8 ウェイ･セット･アソシアティブ L1 命令キャッシュ
- 分岐予測ユニットと ITLB
- 2 つの命令デコーダー (それぞれ 1 サイクルあたり最大 1 命令をデコード可能)

フロントエンドでは、1 サイクルあたり最大 2 命令を命令キューに格納して、スケジューリングすることができます。スケジューラーは、2 つの発行ポートを通じて 1 サイクルあたり最大 2 命令を整数実行クラスターまたは SIMD/FP 実行クラスターに発行できます。

2 つの発行ポートはそれぞれ、1 サイクルあたり 1 命令を、実行する整数クラスターまたは SIMD/FP クラスターにディスパッチできます。整数クラスターと SIMD/FP クラスターのポート･バインディングは、以下の機能を備えています。

- 整数実行クラスター:
  - ポート 0: ALU0、シフト/ローテートユニット、ロード/ストア
  - ポート 1: ALU1、ビット処理ユニット、分岐ユニットと LEA
  - 0 サイクルの実効「ロード後利用」レイテンシー

- SIMD/FP 実行クラスター:
  - ポート 0: SIMD ALU、シャッフルユニット、SIMD/FP 乗算ユニット、除算ユニット (IMUL、IDIV をサポート)
  - ポート 1: SIMD ALU、FP 加算器
  - SIMD/FP クラスター内の 2 つの SIMD ALU とシャッフルユニットは 128 ビット幅ですが、64 ビット SIMD 整数演算はポート 0 のみで実行されます
  - FP 加算器は、128 ビット･データパスで ADDPS/SUBPS を実行できます。ほかの FP 加算のデータパスは 64 ビット幅です

  - FP/SIMD 実行の安全な命令認識アルゴリズムにより、例外を生じる可能性がある古い FP/SIMD 命令によってブロックされることなく、レイテンシーの短い新しい整数命令を実行できます
  - FP 乗算パイプはメモリーロードもサポートしています
  - メモリーロードを伴う FP ADD 命令は、両方のポートをディスパッチに使用できます

インテル® 64 アーキテクチャーでは、メモリー実行サブシステム (MEU) は、32 ビットまたは 36 ビットの物理アドレスモードで 48 ビットのリニアアドレスをサポートします。MEU は以下を備えています。

- 24KB 第 1 レベルキャッシュ
- L1 データキャッシュへのハードウェア・プリフェッチ
- 4KB およびそれよりも大きなページ構造に対応した 2 レベルの DTLB
- DTLB と ITLB のミスを処理するハードウェア・ページウォーカー
- 2 つのアドレス生成ユニット (ポート 0 はロードとストアをサポート。ポート 1 は LEA とスタック操作をサポート)
- 整数演算でのストア・フォワーディングのサポート
- 8 つのライト・コンバイン・バッファー

バス論理サブシステムは以下の機能を備えています。

- 512KB の 8 ウェイ・セット・アソシアティブ・ユニファイド L2 キャッシュ
- L2 へのハードウェア・プリフェッチとフロントサイド・バスへのインターフェイス・ロジック

### 6.2.1 Intel Atom® 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・テクノロジーのサポート

命令キューは、2 つのスレッドからの命令実行をスケジューリングできるように静的に分割されています。スケジューラーは、いずれかのスレッドから命令を 1 つ選択し、実行に向けてポート 0 またはポート 1 にディスパッチすることができます。ハードウェアは、公平な基準と、各スレッドでの順方向進行の対応状況に基づいて、2 つのスレッド間で命令のフェッチ/デコード/ディスパッチを選択します。

## 6.3 Intel Atom® マイクロアーキテクチャー向けのコーディングの推奨事項

アウトオブオーダー・マイクロアーキテクチャーに適用される命令スケジューリングとコーディング手法が、インオーダー・マイクロアーキテクチャーで最適なパフォーマンスを実現できるとは限りません。同様に、Intel Atom® マイクロアーキテクチャーのようなインオーダー・パイプライン向けの命令スケジューリングとコーディング手法が、アウトオブオーダー・マイクロアーキテクチャーで最適なパフォーマンスを実現できるとは限りません。ここでは、主な導入対象が Intel Atom® マイクロアーキテクチャー・ベースのプロセッサーであるソフトウェアに固有のコーディング推奨事項について説明します。

### 6.3.1 Intel Atom® マイクロアーキテクチャーのフロントエンドの最適化

Intel Atom® マイクロアーキテクチャーのフロントエンドにある 2 つのデコーダーは、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命令の大半を処理できます。複雑な演算を処理する一部の命令では、フロントエンドの MSROM を使用する必要があります。2 つのデコーダーを通過する命令はほとんどの場合、フロントエンドのデコーダーユニットのいずれかによってデコードされます。MSROM が必要な命令や、フロントエンドでデコーダー割り当ての再調整が行われる場合、フロントエンドで遅延が発生します。

ソフトウェアは、固有のパフォーマンス監視イベントを使用して、フロントエンドでデコーダー割り当ての再調整が行われる命令シーケンスや条件を検出できます。

**アセンブリー/コンパイラー・コーディング規則 1 (影響 MH、一般性 ML)**: Intel Atom® プロセッサーでは、MSROM を必要とする複雑な命令を最小限に抑えることで、2 つのデコードユニットの最適なデコード帯域幅を活用できます。

パフォーマンス監視イベントの「MACRO_INSTS.NON_CISC_DECODED」と「MACRO_INSTS.CISC_DECODED」を使用して、ワークロード中で MSROM を必要とした命令の割合を評価できます。

**アセンブリー/コンパイラー・コーディング規則 2 (影響 M、一般性 H)**: Intel Atom®プロセッサーでは、命令ワーキングセットの要素を小さくすると、2 つのデコードユニットによって提供される最適なデコード帯域幅をフロントエンドで活用できます。

**アセンブリー/コンパイラー・コーディング規則 3 (影響 MH、一般性 ML)**: Intel Atom®プロセッサーでは、連続する x87 命令の使用を避けると、2 つのデコードユニットによって提供される最適なデコード帯域幅をフロントエンドで活用できます。

パフォーマンス監視イベント「DECODE_RESTRICTION」を使用すると、ワークロード中でデコード・スループットの低下をもたらす遅延が発生した回数をカウントできます。

一般に、リタイアした「命令あたりのサイクル数」が 1 未満になるまで、フロントエンドの制約はパフォーマンスの制限要素にはなりません (理論上の最大リタイアメント・スループットは CPI 値 0.5 に相当)。CPI を 1 未満にするには、フロントエンドを通過する際に命令ペアが 2 つのデコーダーによって並列にデコードされる命令シーケンスを作成することが重要です。フロントエンドの通過後は、スケジューラーや実行ハードウェアがデコードペアを同じ順序でポート 0 とポート 1 にディスパッチする必要はありません。

デコーダーはジャンプ命令より先をデコードできないため、ジャンプはデコーダーに最適化されたペア内の 2 番目の命令として組み合わせる必要があります。フロントエンドでは x87 命令を 1 サイクルあたり 1 個のみ処理可能であり、デコーダーユニット 0 だけが MSROM への転送を要求できます。命令が 8 バイトを超えていたり、命令のプリフィクスが 3 つを超えていると、MSROM の転送が行われ、フロントエンドで 2 サイクルの遅延が発生します。

命令の長さとアライメントは、デコードのスループットに影響を与えることがあります。フロントエンドがバッファーをプリフェッチする場合、7 サイクル内に 48 バイトを超えるデコードを行うとバッファー待機によってフロントエンドで遅延が生じます。これはスループットを制限します。また、命令ペアが 16 バイト境界をまたぐたびに、フロントエンド・バッファーを少なくとも 1 サイクル余分に保持する必要が生じます。したがって、16 バイト境界をまたぐ命令アライメントは、大きな問題となります。無視できるプリフィクスと命令を組み合わせて使用することで命令アライメントを改善できます。

例 6-1 Intel Atom® マイクロアーキテクチャー上でデコードを最適化するための命令ペアとアライメント

| アドレス | 命令バイト | 逆アセンブル |
|---|---|---|
| 7FFFFDF0 | 0F594301 | mulps xmm0, [ebx+ 01h] |
| 7FFFFDF4 | 8341FFFF | add dword ptr [ecx-01h], -1 |
| 7FFFFDF8 | 83C2FF | add edx, , -1 |
| 7FFFFDFB | 64 | ; FS プリフィクスのオーバーライドは無視され、コードのアライメントを改善 |
| 7FFFFDFC | F20f58E4 | add xmm4, xmm4 |
| 7FFFFE00 | 0F594B11 | mulps xmm1, [ebx+ 11h] |
| 7FFFFE04 | 8369EFFF | sub dword ptr [ecx- 11h], -1 |
| 7FFFFE08 | 83EAFF | sub edx, -1 |
| 7FFFFE0B | 64 | ; FS プリフィクスのオーバーライドは無視され、コードのアライメントを改善 |
| 7FFFFE0C | F20F58ED | addsd xmm5, xmm5 |
| 7FFFFE10 | 0F595301 | mulps xmm2, [ebx +1] |
| 7FFFFE14 | 8341DFFF | add dword ptr [ecx-21H], -1 |
| 7FFFFE18 | 83C2FF | add edx, -1 |
| 7FFFFE1B | 64 | ; FS プリフィクスのオーバーライドは無視され、コードのアライメントを改善 |
| 7FFFFE1C | F20F58F6 | addssd xmm6, xmm6 |
| 7FFFFE20 | 0F595B11 | mulps xmm3, [ebx+ 11h] |
| 7FFFFE24 | 8369CFFF | sub dword ptr [ecx- 31h], -1 |
| 7FFFFE28 | 83EAFF | sub edx, -1 |

小さなループ中にレイテンシーの長い操作が含まれている場合、ループアンロールは、レイテンシーの長い命令とペアにできる隣接した命令を見つけて、その隣接した命令の順方向進行を可能にするための手法です。ただし、コードサイズの増加や分岐ターゲットバッファーへの負荷など、ループアンロールの影響も評価しなければなりません。

パフォーマンス監視イベント「BACLEARS」は、ループアンロールがフロントエンドのパフォーマンスにとってプラスであるかマイナスであるかを評価する手段となります。「ICACHE_MISSES」イベントも、ループアンロールによって命令要素が増加するかどうかを評価できます。

Intel Atom® プロセッサーの分岐予測機構は、異なる分岐タイプを区別しません。異なる分岐タイプが混在すると、分岐予測ハードウェアで混乱が生じることがあります。

パフォーマンス監視イベント「BR_MISSP_TYPE_RETIRED」は、分岐タイプに起因する分岐予測上の問題を評価する手段となります。

## 6.3.2 実行コアの最適化

ここでは、2 マイクロオペレーション (uop) の発行が可能な実行コアを利用して、ソフトウェアが 2 つの命令による順方向進行をより頻繁に行うためのいくつかの事項について説明します。

### 6.3.2.1 整数命令の選択

インオーダー・マシンでは、命令の選択と組み合わせが、データの実行準備が整った命令の命令レベルの並列性を検出するマシンの能力に影響を与えることがあります。以下にその例をいくつか示します。

- **EFLAG**: EFLAG フラグビットを参照する命令は、EFLAG レジスターを更新する命令と同じサイクル内では発行できません。例えば、ADD はキャリービットを変更する可能性がある、更新命令です。JC (または ADC) はキャリービットを読み出すため参照命令となります。
  - 条件ジャンプは、参照命令の後に続くサイクルで発行できます。
  - その他の EFLAG ビットの参照命令を更新命令の後に発行する場合、1 サイクル待機しなければなりません (2 サイクル遅延)。

**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 H)**: Intel Atom® プロセッサーでは、2 サイクルの遅延が生じるフラグ更新命令とフラグ参照命令の間に MOV 命令を配置します。これにより、部分的なフラグ依存関係を防止できます。

- **長いレイテンシーの整数命令**: これらの命令は、同じスレッド内のレイテンシーの短い命令の発行をブロックします (プログラムの順序により必要)。また、ライトバック・リソースを解決する際は、両方のスレッド内のレイテンシーの短い命令も 1 サイクルの間ブロックします。
- **共通のデスティネーション**: 結果を同じデスティネーションに格納する 2 つの命令は、同じサイクル内では発行できません。
- **コストがかかる命令**: 一部の命令には特別な要件があり、実行中長期間にわたりハードウェア・リソースを消費するため多くのコストがかかります。命令キュー内でその命令が最も古くなるまで、実行ユニットで遅延することがあります。また、ほかの新しい命令の発行を遅らせることもあります。このような例としては、FDIV や両方のポートから実行ユニットを必要とする命令などが挙げられます。

### 6.3.2.2 アドレス生成

ハードウェアは、実行準備が整った命令が利用するデータは、準備されている必要がある、という一般要件の最適化を行います。また、アドレス生成は、データの準備よりも先に行われます。ほかの命令が生成するデータを必要とする依存関係がアドレス生成で生じた場合、3 サイクルの遅延が発生します。

2 マイクロオペレーション (uop) の発行が可能なマシンの実行スループットに影響を与える 3 つの状況では、アドレス生成ユニット (AGU) を直接使用できます。この状況を以下に示します。

- **暗黙的な ESP 更新**: ESP レジスターが命令のデスティネーション (明示的な ESP 更新)として使用されない場合、PUSH、POP、CALL、RETURN などの命令では暗黙的の ESP 更新が行われます。明示的な ESP 更新と暗黙的な ESP 更新を混在させても、アドレス生成とデータ実行との依存関係が発生します。
- **LEA**: LEA 命令は、ALU ではなく AGU を使用します。LEA のソースレジスターの 1 つが実行ユニットから提供される場合、依存関係により 3 サイクルの遅延が発生します。したがって、2 つの値を加算して結果を第 3 のレジスターに格納する手法では、LEA を使用してはなりません。LEA は、アドレス計算に使用すべきです。
- **整数から FP/SIMD への転送**: 整数データをマシンの FP/SIMD 側に転送する命令でも、AGU が使用されます。このような命令の例として、MOVD や PINSRW が挙げられます。これらの命令のソースレジスターの 1 つが実行ユニットの結果に依存している場合、この依存関係でも 3 サイクルの遅延が発生します。

例 6-2 AGU と実行ユニットとの依存関係を回避する代替手法

```
a) 三項演算で LEA を使用した場合の 3 サイクルの遅延
     mov eax, 0x01
     lea eax, 0x8000[eax+ebp]; eax の値は直線の命令で生成される
     ; lea と実行の依存性により、3 サイクルの遅延
b) 実行中の依存関係を処理し、AGU と実行との依存関係を回避する
     mov eax, 0x01
     add eax, 0x8000
     add eax, ebp
```

**アセンブリー/コンパイラー・コーディング規則 5 (影響 MH、一般性 H)**: Intel Atom® プロセッサーでは、アドレス操作に LEA を使用すべきですが、ソフトウェアは、ALU から AGU に対する依存関係 (アドレス操作または ESP 更新に LEA の代わりに ALU 命令を使用する場合、または三項加算や、アドレス生成への供給を行わない非破壊的書き込みに LEA を使用する場合) を回避しなければなりません。または、AGU を参照する命令の前に更新命令を配置する際、3 サイクル以上の間隔を空けます。

### 6.3.2.3 整数乗算

整数乗算命令の実行には、数サイクルかかります。整数乗算命令はレイテンシーの長い別の命令とともに、実行フェーズで順方向に進行できるようにパイプライン化されます。ただし整数乗算命令は、プログラム順序の要件により、単一サイクルのほかの整数命令の発行をブロックします。

**アセンブリー/コンパイラー・コーディング規則 6 (影響 M、一般性 M)**: Intel Atom® プロセッサーでは、整数乗算命令の後に独立した FP 乗算または整数乗算を配置すると、IMUL をパイプライン実行できます。

例 6-3 整数計算における命令のパイプライン実行

```
a) 複数サイクル IMUL 命令が 1 サイクル整数命令をブロック
     imul eax, eax
     add ecx, ecx ; 1 サイクル整数命令は IMUL で 4 サイクルブロックされる
     imul ebx, ebx ; インオーダー発行による命令ブロック
独立して連続し発行される IMUL はパイプライン化される
     imul eax, eax
     imul ebx, ebx ; 2 番目の IMUL は 1 サイクル後に発行できる
     add ecx, ecx ; 1 サイクル整数命令は IMUL でブロックされる
```

### 6.3.2.4 整数シフト命令

シフトカウントを即値バイトでエンコードする整数シフト命令では、1 サイクルのレイテンシーが発生します。一方、ECX レジスターでシフトカウントを使用するシフト命令は、レジスターカウントが更新されるのを待機しなければならないことがあります。そのため、レジスターカウントを使用するシフト命令では、3 サイクルのレイテンシーが発生します。

**アセンブリー/コンパイラー・コーディング規則 7 (影響 M、一般性 M)**: Intel Atom® プロセッサーでは、整数シフト命令の暗黙的なレジスターカウントの更新命令は、シフト命令の 2 サイクル以上前に配置します。

### 6.3.2.5 パーシャル・レジスター・アクセス

パーシャル・レジスター・アクセスによって追加の遅延は生じませんが、インオーダー・ハードウェアはレジスター全体への依存関係を追跡します。したがって、AL や AH などの 8 ビット・レジスターは、独立したレジスターとして扱われません。

また、LEA、単純なロード、POP など一部の命令では、入力値が 4 バイト未満であると速度が低下します。

**アセンブリー/コンパイラー・コーディング規則 8 (影響 M、一般性 MH)**: Intel Atom® プロセッサーでは、LEA、単純なロード、POP の入力値が 4 バイト未満であると速度が低下します。

### 6.3.2.6 FP/SIMD 命令の選択

表 6-1 に、ソフトウェアで最も頻繁に使用される Intel Atom® マイクロアーキテクチャーの各種実行ユニットの特性を示します。

表 6-1 Intel Atom® マイクロアーキテクチャーの命令レイテンシー/スループットのまとめ

| 命令カテゴリー | レイテンシー (サイクル) | スループット | 実行ユニット数 |
|---|---|---|---|
| SIMD 整数 ALU | | | |
| 128 ビット ALU/論理/移動 | 1 | 1 | 2 |
| 64 ビット ALU/論理/移動 | 1 | 1 | 2 |
| SIMD 整数 シフト | | | |
| 128 ビット | 1 | 1 | 1 |
| 64 ビット | 1 | 1 | 1 |
| SIMD シャッフル | | | |
| 128 ビット | 1 | 1 | 1 |
| 64 ビット | 1 | 1 | 1 |
| SIMD 整数乗算 | | | |
| 128 ビット | 5 | 2 | 1 |
| 64 ビット | 4 | 1 | 1 |
| FP 加算 | | | |
| X87 op (FADD) | 5 | 1 | 1 |
| スカラー SIMD (addsd, addss) | 5 | 1 | 1 |
| パックド単精度 (addps) | 5 | 1 | 1 |
| パックド倍精度 (addpd) | 6 | | 1 |
| FP 乗算 | | | |
| X87 Ops (FMUL) | 5 | 5 | 1 |
| スカラー単精度 (mulss) | 4 | 2 | 1 |
| スカラー倍精度 (mulsd) | 5 | 1 | 1 |
| パックド単精度 (mulps) | 5 | 2 | 1 |
| パックド倍精度 (mulpd) | 9 | 2 | 1 |
| IMUL | | 9 | |
| IMUL r32, r/m32 | 5 | 1 | 1 |
| IMUL r12, r/m16 | 6 | 1 | 1 |

SIMD/FP 命令の選択では通常、最初にレイテンシーの短いものを選び、その次には可能な限りスループットの高いものを選ぶ必要があります。パックド倍精度命令はパイプライン化されない点に注意してください。代わりに 2 つのスカラー倍精度命令を使用すると、実行クラスターのパフォーマンスを高めることができます。

**アセンブリー/コンパイラー・コーディング規則 9 (影響 MH、一般性 H)**: Intel Atom® プロセッサーでは、FP スタックを使用する x87 命令よりも、XMM レジスターで動作する SIMD 命令を優先します。そして、可能な限り、パックド単精度命令を使用します。パックド倍精度命令はスカラー倍精度命令に置き換えます。

**アセンブリー/コンパイラー・コーディング規則 10 (影響 M、一般性 ML)**: Intel Atom® プロセッサーでは、超越関数のような高度な演算を実行するライブラリー・ソフトウェアは、ネイティブの x87 命令ではなく、XMM レジスターで動作する SIMD 命令を使用すべきです。

**アセンブリー/コンパイラー・コーディング規則 11(影響 M、一般性 M)**: Intel Atom® プロセッサーでは、可能な限り DAZ と FTZ を有効にします。

いくつかのパフォーマンス監視イベントは、SIMD/FP 命令選択のチューニングに有効です。例えば「SIMD_INST_RETIRED.{PACKED_SINGLE, SCALAR_SINGLE, PACKED_DOUBLE, SCALAR_DOUBLE}」は、プログラムでの命令選択の判断に使用できます。「FP_ASSIST」や「SIR」を使用すると、浮動小数点例外 (または不正なアラーム) がプログラムのパフォーマンスに影響を与えているかどうか確認できます。

除算命令のレイテンシーとスループットは、入力値やデータサイズによって異なります。Intel Atom® マイクロアーキテクチャーでは、基数 2 の除算器ユニットを実装しています。したがって、divide/sqrt のレイテンシーは、ほかの

FP 演算よりも大幅に長くなります。また、divide/sqrt の発行スループット速度も同様に低下します。除算器ユニットは 2 つの論理プロセッサー間で共有されるため、ソフトウェアは除算命令の代替手段を検討すべきです。

**アセンブリー/コンパイラー・コーディング規則 12 (影響 H、一般性 L)**: Intel Atom® プロセッサーでは、どうしても必要なときのみ除算命令を使用し、データサイズが最小のオペランドを使用するように考慮します。

パフォーマンス監視イベント「DIV」と「CYCLES_DIV_BUSY」を使用すると、除算がプログラム中のボトルネックになっているかどうかを確認できます。

一般に、FP 演算は整数命令よりもレイテンシーが長くなります。FP 演算からの結果のライトバックは通常、整数パイプラインよりも後のパイプライン・ステージで発生します。したがって、命令が FP 演算の結果に依存関係を持っている場合、2 サイクルの遅延が発生します。この種の命令の例として、FP から整数への変換である CVTxx2xx や、XMM から汎用レジスターへの MOVD があります。

連続する 4 つの単精度データ要素を用いた演算を必要とする場合、MOVUPS よりも PALIGNR + MOVAPS を優先します。MOVUPS xmm1, _pArray[k] (メモリーアドレス _pArray は 16 バイト境界にアライメント) のように、制約なしに配列インデックス k によって 4 つのデータ要素をロードすると、定期的にキャッシュライン分割が発生し 14 サイクルの遅延が生じます。

最適なアプローチでは、4 の倍数でない k ごとに、j = 4*(k/4) で k を 4 の倍数に丸め、MOVAPS と MOVAPS xmm1, _pArray[j] と MOVAPS xmm1, _pArray[j+4] を実行してから、PALIGNR を使用して演算に必要な 4 つのデータ要素を結合します。

**アセンブリー/コンパイラー・コーディング規則 13 (影響 MH、一般性 M)**: Intel Atom® プロセッサーでは、MOVUPS よりも MOVAPS + PALIGN のシーケンスを優先します。同様に、MOVDQU よりも MOVDQA + PALIGNR を優先します。

## 6.3.3 メモリーアクセスの最適化

ここでは、ソフトウェアによってメモリー・サブシステムのパフォーマンスを最適化できるいくつかの事項について説明します。

特定のハザードに遭遇するキャッシュアクセスからのシステムメモリーに対するメモリーアクセスを行うと、そのメモリーアクセスは、データの実行準備が整っているときでもレイテンシーの短い命令の発行をブロックするため、コストのかかる操作になります。

パフォーマンス監視イベント「REISSUE」を使用して、プログラムにおける再発行されたメモリー命令の影響を評価できます。

### 6.3.3.1 ストア・フォワーディング

一部の限られた状況では、Intel Atom® マイクロアーキテクチャーは、先行するストア操作から後続のロード命令にデータを転送できます。この状況を以下に示します。

- ストア・フォワーディングは、整数パイプラインでのみサポートされており、FP データや SIMD データには適用されません。また、以下の条件を満たす必要があります。

  a. ストア操作とロード操作は、同じサイズであり、かつ同じアドレスに対するものでなければなりません。
  b. データサイズが 8 バイトを超えると、ストア操作からの転送は行われません。

- データ転送が行われる際、データはアドレスの最下位 12 ビットに基づいて転送されます。そのためソフトウェアは、あるアドレスにストアした後に、その最下位 12 ビットをエイリアシングする別のアドレスからロードするような、アドレス・エイリアシングの状況を回避しなければなりません。

### 6.3.3.2 第 1 レベルキャッシュ

Intel Atom® マイクロアーキテクチャーは、16 個の 4 バイト・チャンクで構成される L1 データキャッシュの各 64 バイト・キャッシュラインを処理します。この実装特性は、データ・アライメントと一部のデータ・アクセス・パターンにパフォーマンス上の影響を与えます。

**アセンブリー/コンパイラー・コーディング規則 14 (影響 MH、一般性 H)** : Intel Atom® プロセッサーでは、データがメモリー内で自然サイズにアライメントされるように配置します。例えば、4 バイトのデータは、4 バイト境界にアライメントする必要があります。また、チャンク内の小規模なアクセス (4 バイト未満) では、異なるバイトにアクセスすると、遅延が生じることがあります。

### 6.3.3.3 セグメントベース

Intel Atom® マイクロアーキテクチャーでは、アドレス生成ユニットは、セグメントベースがデフォルトで 0 になると想定されます。そのためセグメントベースが 0 以外であると、ロード操作やストア操作で遅延が発生します。

- セグメントベースがキャッシュライン境界にアライメントされていない場合、メモリー操作の最大スループットは 9 サイクルあたり 1 操作に減少します。

セグメントベースが 0 以外でも、キャッシュラインにアライメントされている場合、ペナルティーはセグメントベースによって異なります。

- DS では、最大スループットが 2 サイクルあたり 1 操作になります。
- FS と GS では、最大スループットが 2 サイクルあたり 1 操作になります。ただし、FS と GS は 0 以外のベースでのみ使用されることが想定されているため、セグメントベースが 0 であっても、最大スループットは 2 サイクルあたり 1 操作のままです。
- ES:
  - 文字列操作のデスティネーションが暗黙的なセグメントベースとして使用される場合、0 以外であるがキャッシュラインにアライメントされたベースについては、最大スループットが 2 サイクルあたり 1 操作になります。
  - その他の場合は、9 サイクルあたり 1 操作しか実行されません。

- CS と SS では、セグメントベースが 0 以外であるがキャッシュラインにアライメントされている場合は常に、最大スループットが 9 サイクルあたり 1 操作となります。

**アセンブリー/コンパイラー・コーディング規則 15 (影響 H、一般性 ML)**: Intel Atom® プロセッサーでは、可能な限りベースが 0 にセットされたセグメントを使用し、キャッシュライン境界にアライメントされていない 0 以外のセグメント・ベース・アドレスはできるだけ回避します。

**アセンブリー/コンパイラー・コーディング規則 16 (影響 H、一般性 L)**: Intel Atom® プロセッサーでは、0 以外のセグメントベースを使うときは、DS、FS、GS を使用します。さらに文字列操作では、暗黙的な ES を使用すべきです。

**アセンブリー/コンパイラー・コーディング規則 17 (影響 M、一般性 ML)**: Intel Atom® プロセッサーでは、セグメントベースが 0 の FS、GS よりも ES、DS、SS の使用を優先します。

### 6.3.3.4 文字列移動

Intel Atom® プロセッサー上で MOVS/STOS 命令と REP プリフィクスを使用する場合は、以下の事項について認識する必要があります。

- カウント値が小さい場合、REP プリフィクスを使用すると、REP プリフィクスを使用しないときよりも効率が低下します。これは、ハードウェアによって小さな REP カウントの最適化が行われるためです。

- カウント値が大きい場合、REP プリフィクスを使用すると、16 バイト SIMD 命令を使用するときよりも効率が低下します。
- ループ反復でアドレスをインクリメントする場合は、明示的な ADD 命令よりも LEA 命令を優先すべきです。
- メモリー操作が L2 にアクセスするようなデータ要素の場合、ソフトウェア・プリフェッチを使用してデータを L1 に格納すると、メモリー操作が再発行されることを回避できます。
- 文字列/メモリー操作がシステムメモリーにアクセスする場合は、ストリーミング・ストア命令の非テンポラルなヒントを使用すると、キャッシュの汚染を回避できます。

例 6-4 64 バイトのメモリーコピー

```
T1: prefetcht0 [eax+edx+0x80] ; 先行する 2 つのループ分をプリフェッチ
    movdqa xmm0, [eax+ edx] ; ソースからデータをロード (プリフェッチにより L1 から)
    movdqa xmm1, [eax+ edx+0x10]
    movdqa xmm2, [eax+ edx+0x20]
    movdqa xmm3, [eax+ edx+0x30]
    movdqa [ebx+ edx], xmm0; デスティネーションへストア
    movdqa [ebx+ edx+0x10], xmm1
    movdqa [ebx+ edx+0x30], xmm2
    movdqa [ebx+ edx+0x30], xmm3
    lea edx, 0x40 ; 次の反復のオフセットを合わせるため LEA を利用
    dec ecx
    jnz T1
```

### 6.3.3.5 引数渡し

Intel Atom® マイクロアーキテクチャーにおけるロード-ストア・フォワーディングのサポートは限定的であるため、スタックを介したパラメーターの受け渡しは、呼び出し先関数による参照を制限します。例えば、「bool」データと「char」データは通常、32 ビット・データとしてスタック上にプッシュされますが、呼び出し先関数がスタックから「bool」データまたは「char」データを読み出すと、ストア・フォワーディングの遅延が発生し、メモリー操作が再発行されます。

コンパイラーはこの制約を認識してプロローグを生成し、呼び出し先関数がサイズの小さなデータではなく 32 ビット・データを読み出せるようにすべきです。

**アセンブリー/コンパイラー・コーディング規則 18 (影響 MH、一般性 M)**: Intel Atom® プロセッサーでは、「bool」値と「char」値は、32 ビット・データとしてスタックに渡したり、スタックから読み出すべきです。

### 6.3.3.6 関数呼び出し

Intel Atom®マイクロアーキテクチャーでは、PUSH/POP 命令を使用してスタック領域や、関数呼び出し/戻り間のアドレス調整を管理するほうが、ENTER/LEAVE 命令を使用するよりも適しています。

これは、PUSH/POP が MSROM を必要とせず、スタック・ポインター・アドレスの更新が AGU で行われるためです。

呼び出し先関数は、呼び出し元に戻る必要がある場合、POP 命令を発行して、データをリストアしたり、EBP からスタックポインターをリストアできます。

**アセンブリー/コンパイラー・コーディング規則 19 (影響 MH、一般性 M)**: Intel Atom® プロセッサーでは、PUSH/POP のレジスター形式を優先し、LEAVE の使用は避けます。また、ESP の調整には、ADD/SUB ではなく LEA を使用します。

### 6.3.3.7 乗算/加算の依存関係チェーンの最適化

依存関係がある乗算や加算は、Intel Atom® マイクロアーキテクチャーのフロントエンドとインオーダー実行パイプラインを最適化するための良いコーディング手法例です。

例 6-5a に、アウトオブオーダー・マイクロアーキテクチャーで使用可能なコードシーケンスを示します。このシーケンスは、Intel Atom®マイクロアーキテクチャーで最適なものとは大きく異なります。乗算や加算のレイテンシーが最大になり、2 マイクロオペレーション (uop) 発行が可能なパイプラインを活用するにはあまり適していません。

例 6-5b には、2 マイクロオペレーション (uop) 発行が可能な Intel Atom® マイクロアーキテクチャーのインオーダー・パイプラインを活用する改善されたコードシーケンスを示します。乗算と加算との間に依存関係が存在するため、レイテンシーの発生は部分的にのみ隠匿されます。

例 6-5 依存関係のある乗算と加算の例

```
a) ストールが発生する命令シーケンス
; アキュムレーター xmm2 は初期化されています
Top: movaps xmm0, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     movaps xmm1, [edi] ; 16 バイト境界でメモリーにストアされたベクトル
     mulps xmm0, xmm1
     addps xmm2, xmm0 ; 依存性と分岐は mul と add の遅延を招く
     add esi, 16 ;
     add edi, 16
     sub ecx, 1
     jnz top
```

```
b) 実行スループットを高めるように改善された命令シーケンス
; アキュムレーター xmm4 は初期化されています
Top: movaps xmm0, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm0, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm4, xmm0 ; 独立した命令により、部分的な遅延がカバーされる
     dec ecx ;
     jnz top
```

```
c) アンロールとインターリーブによってさらに改善された命令シーケンス
; アキュムレーター xmm0, xmm1, xmm2, xmm3 は初期化されています
Top: movaps xmm0, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm0, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm5, xmm1 ; アンロールとインターリーブによって依存する乗算は引き離される
     movaps xmm1, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm1, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm6, xmm2 ; アンロールとインターリーブによって依存する乗算は引き離される
     movaps xmm2, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm2, [edi] ;
     lea esi, [esi+16] ; mulps と同時にスケジュール可能
     addps xmm7, xmm3 ; アンロールとインターリーブによって依存する乗算は引き離される
     movaps xmm3, [esi] ; 16 バイト境界でメモリーにストアされたベクトル
     lea esi, [esi+16] ; movaps と同時にスケジュール可能
     mulps xmm3, [edi] ;
```

```
    lea esi, [esi+16] ; mulps と同時にスケジュール可能
    addps xmm4, xmm0 ; アンロールとインターリーブによって依存する乗算は引き離される
    sub ecx, 4;
    jnz top
    ; ループ内の依存性を減らすため、アキュムレーターは xmm0, xmm1, xmm2, xmm3 を集計する
```

例 6-5c は、命令レベルの並列性を高め、乗算と加算におけるレイテンシーの発生をさらに抑える手法を示しています。アンロールを 4 回行うことにより、各 ADDPS 命令を依存関係がある更新命令の MULPS から離して配置できます。また、インターリーブ手法を使用することにより、依存関係がない ADDPS と MULPS を近くに配置できます。MULPS と ADDPS を実行するハードウェアはパイプライン化されているため、この手法では、例 6-5b と比べてはるかに効果的にレイテンシーを隠匿できます。

### 6.3.3.8 位置に依存しないコード

位置に依存しないコードは多くの場合、命令ポインターの値を取得する必要があります。例 6-6a は、一致する RET がない CALL を発行して IP の値を ECX レジスターに格納する手法を示しています。例 6-6b は、一致する CALL/RET ペアを使用して IP の値を ECX レジスターに格納する代替手法を示します。

例 6-6 命令ポインターの照会手法

```
a) RET がない CALL を使用して IP を取得
   call _label; プッシュされた return アドレスは次の命令の IP
_label:
   pop ECX; この命令の IP は、ECX に代入されます
```

```
b) call/ret が一致したペアを使用
   call _lblcx;
   ...; ECX はこの命令の IP を含む
   ...
_lblcx
   mov ecx, [esp];
   ret
```

## 6.4 命令レイテンシー

ここでは、Intel Atom®マイクロアーキテクチャーのポート･バインディングとレイテンシーに関する情報を提供します。各命令のポート･バインディング情報は、以下の 3 つの状態のいずれかを示しています。

- 1 桁の数字 - 発行される特定のポート番号
- (0, 1) - ポート 0 またはポート 1
- ‘B’ - 両方のポートが必要

「命令」列では以下のような情報を示します。

- 同じ命令の異なるオペランド構文のポート･バインディングおよびレイテンシーが同じ場合は、オペランド構文を省略しています。
- 同じ命令の異なるオペランド構文のポート･バインディングやレイテンシーが異なる場合は、オペランド構文を記載しています。ただし、オペランドサイズが異なる命令構文については、脚注を使って簡潔に記載している場合があります。

MSROM によるデコーダーの支援が必要な命令には、「説明カラム」にその旨が記されています (よりデコード効率の高い代替手段があれば、そのような命令の使用は最小限に抑えるべきです)。

表 6-2 Intel Atom® マイクロアーキテクチャーの命令レイテンシー

| Instruction | Ports | Latency | Throughput |
|---|---|---|---|
| **DisplayFamily_DisplayModel** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** |
| ADD/AND/CMP/OR/SUB/XOR/TEST[1] (E)AX/AL, imm; | (0, 1) | 1 | 0.5 |
| ADD/AND/CMP/OR/SUB/XOR[2] mem, Imm8;<br>ADD/AND/CMP/OR/SUB/XOR/TEST[4] mem, imm; TEST m8, imm8 | 0 | 1 | 1 |
| ADD/AND/CMP/OR/SUB/XOR/TEST[2] mem, reg; ADD/AND/CMP/OR/SUB/XOR[2] reg, mem; | 0 | 1 | 1 |
| ADD/AND/CMP/OR/SUB/XOR[2] reg, Imm8; ADD/AND/CMP/OR/SUB/XOR[4] reg, imm | (0, 1) | 1 | 0.5 |
| ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, mem | B | 7 | 6 |
| ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, xmm | B | 6 | 5 |
| ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, mem | B | 5 | 1 |
| ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, xmm | 1 | 5 | 1 |
| ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, mem | 0 | 1 | 1 |
| ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, xmm | (0, 1) | 1 | 1 |
| BSF/BSR r16, m16 | B | 17 | 16 |
| BSF/BSR[3] reg, mem | B | 16 | 15 |
| BSF/BSR[4] reg, reg | B | 16 | 15 |
| BT m16, imm8; BT[3] mem, imm8 | (0, 1) | 2; 1 | 1 |
| BT m16, r16; BT[3] mem, reg | B | 10, 9 | 8 |
| BT[4] reg, imm8; BT[4] reg, reg | 1 | 1 | 1 |
| BTC m16, imm8; BTC[3] mem, imm8 | B | 3; 2 | 2 |
| BTC/BTR/BTS m16; r16 | B | 12 | 11 |
| BTC/BTR/BTS[3] mem, reg | B | 11 | 10 |
| BTC/BTR/BTS[4] reg, imm8; BTC/BTR/BTS[4] reg, reg | 1 | 1 | 1 |
| CALL mem | (0, 1) | 2 | 2 |
| CALL reg; CALL rel16; CALL rel32 | B | 1 | 1 |
| CMOV[4] reg, mem; MOV[1] (E)AX/AL, MOFFS; MOV[2] mem, imm | 0 | 1 | 1 |
| CMOV[4] reg, reg; MOV[2] reg, imm; MOV[2] reg, reg; ; SETcc r8 | (0, 1) | 1 | 0.5 |
| CMPPD/CMPPS xmm, mem, imm; CVTTPS2DQ xmm, mem | B | 7 | 6 |
| CMPPD/CMPPS xmm, xmm, imm; CVTTPS2DQ xmm, xmm | B | 6 | 5 |
| CMPSD/CMPSS xmm, mem, imm | B | 5 | 1 |
| CMPSD/CMPSS xmm, xmm, imm | 1 | 5 | 1 |
| (U)COMISD/(U)COMISS xmm, mem; | B | 10 | 9 |
| (U)COMISD/(U)COMISS xmm, xmm; | B | 9 | 8 |
| CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, mem | B | 8 | 7 |
| CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, xmm | B | 7 | 6 |
| CVTDQ2PS/CVTSD2SS/CVTSI2SS/CVTSS2SD xmm, mem | B | 7 | 6 |

| Instruction | Ports | Latency | Throughput |
|---|---|---|---|
| **DisplayFamily_DisplayModel** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** |
| CVTDQ2PS/CVTSD2SS/CVTSS2SD xmm, xmm | B | 6 | 5 |
| CVT(T)PD2PI mm, mem; CVTPI2PD xmm, mem | B | 8 | 7 |
| CVT(T)PD2PI mm, xmm; CVTPI2PD xmm, mm | B | 7 | 6 |
| CVTPI2PS/CVTSI2SD xmm, mem; | B | 5 | 4 |
| CVTPI2PS xmm, mm; | 1 | 5 | 1 |
| CVTPS2DQ xmm, mem; | B | 7 | 6 |
| CVTPS2DQ xmm, xmm; | B | 6 | 5 |
| CVT(T)PS2PI mm, mem; | B | 5 | 5 |
| CVT(T)PS2PI mm, xmm; | 1 | 5 | 1 |
| CVT(T)SD2SI[3] reg, mem; CVT(T)SS2SI r32, mem | B | 9 | 8 |
| CVT(T)SD2SI[3] reg, xmm; CVT(T)SS2SI r32, xmm | B | 8 | 7 |
| CVTSI2SD xmm, r32; CVTSI2SS xmm, r32 | B | 7; 6 | 5 |
| CVTSI2SD xmm, r64; CVTSI2SS xmm, r64 | B | 6; 7 | 5 |
| CVT(T)SS2SI r64, mem; RCPPS xmm, mem | B | 10 | 9 |
| CVT(T)SS2SI r64, xmm; RCPPS xmm, xmm | B | 9 | 8 |
| CVTTPD2DQ xmm, mem | B | 8 | 7 |
| CVTTPD2DQ xmm, xmm | B | 7 | 6 |
| DEC/INC[2] mem; MASKMOVQ; MOVAPD/MOVAPS mem, xmm | 0 | 1 | 1 |
| DEC/INC[2] reg; FLD ST; FST/FSTP ST; MOVDQ2Q mm, xmm | (0, 1) | 1 | 0.5 |
| DIVPD; DIVPS | B | 125; 70 | 124; 69 |
| DIVSD; DIVSS | B | 62; 34 | 61; 33 |
| EMMS; LDMXCSR | B | 5 | 4 |
| FABS/FCHS/FXCH; MOVQ2DQ xmm, mm; MOVSX/MOVZX r16, r16 | (0, 1) | 1 | 0.5 |
| FADD/FSUB/FSUBR[3] mem | B | 5 | 4 |
| FADD/FADDP/FSUB/FSUBP/FSUBR/FSUBRP ST; | 1 | 5 | 1 |
| FCMOV | B | 6 | 5 |
| FCOM/FCOMP[3] mem | B | 1 | 1 |
| FCOM/FCOMP/FCOMPP/FUCOM/FUCOMP ST; FTST | 1 | 1 | 1 |
| FCOMI/FCOMIP/FUCOMI/FUCOMIP ST | B | 9 | 8 |
| FDIV/FSQRT[3] mem; FDIV/FSQRT ST | 0 | 25-65 | 24-64 |
| FIADD/FIMUL[5] mem | B | 11 | 10 |
| FICOM/FICOMP mem | B | 7 | 6 |
| FILD[4] mem | B | 5 | 4 |
| FLD[3] mem; FXAM; MOVAPD/MOVAPS/MOVD xmm, mem | 0 | 1 | 1 |
| FLDCW | B | 5 | 4 |
| FMUL/FMULP ST; FMUL[3] mem | 0 | 5 | 1 |

| Instruction | Ports | Latency | Throughput |
|---|---|---|---|
| **DisplayFamily_DisplayModel** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** |
| FNSTSW AX; FNSTSW m16 | B | 10; 14 | 9; 13 |
| FST/FSTP[3] mem | B | 2 | 1 |
| HADDPD/HADDPS/HSUBPD/HSUBPS xmm, mem | B | 9 | 8 |
| HADDPD/HADDPS/HSUBPD/HSUBPS xmm, xmm | B | 8 | 7 |
| IDIV r/m8; IDIV r/m16; IDIV r/m32; IDIV r/m64; | B | 33;42;57;197 | 32;41;56;196 |
| IMUL/MUL[6] EAX/AL, mem; IMUL/MUL AX, m16 | B | 7; 8 | 6; 7 |
| IMUL/MUL[7] AX/AL, reg; IMUL/MUL EAX, r32 | B | 7; 6 | 6; 5 |
| IMUL m16, imm8/imm16; IMUL r16, m16 | B | 7; | 6 |
| IMUL r/m32, imm8/imm32; IMUL r32, r/m32 | 0 | 5 | 1 |
| IMUL r/m64, imm8/imm32; | B | 14 | 13 |
| IMUL r16, r16; IMUL r16, imm8/imm16 | B | 6 | 5 |
| IMUL r64, r/m64; IMUL/MUL RAX, r/m64 | B | 11; 12 | 10; 11 |
| JCC[1]; JMP[4] reg; JMP[1] | 1 | 1 | 1 |
| JCXZ; JECXZ; JRCXZ | B | 4 | 1 |
| JMP mem[4]; | B | 2 | 1 |
| LDDQU; MOVDQU/MOVUPD/MOVUPS xmm, mem; | B | 3 | 2 |
| LEA r16, mem; MASKMOVDQU; SETcc m8 | (0, 1) | 2 | 1 |
| LEA, reg, mem | 1 | 1 | 1 |
| LEAVE; | B | 2; | 2 |
| MAXSD/MAXSS/MINSD/MINSS xmm, mem | B | 5 | 1 |
| MAXSD/MAXSS/MINSD/MINSS xmm, xmm | 1 | 5 | 1 |
| MOV[2] MOFFS, (E)AX/AL; MOV[2] reg, mem; MOV[2] mem, reg | 0 | 1 | 1 |
| MOVD mem[3], mm; MOVD xmm, reg[3]; MOVD mm, mem[3] | 0 | 1 | 1 |
| MOVD reg[3], mm; MOVD reg[3], xmm; PMOVMSK reg[3], mm | 0 | 3 | 1 |
| MOVDQA/MOVQ xmm, mem; MOVDQA/MOVD mem, xmm; | 0 | 1 | 1 |
| MOVDQA/MOVDQU/MOVUPD xmm, xmm; MOVQ mm, mm | (0, 1) | 1 | 0.5 |
| MOVDQU/MOVUPD/MOVUPS mem, xmm; | B | 2 | 2 |
| MOVHLPS;MOVLHPS;MOVHPD/MOVHPS/MOVLPD/MOVLPS | 0 | 1 | 1 |
| MOVMSKPD/MOVSKPS/PMOVMSKB reg[3], xmm | 0 | 3 | 1 |
| MOVNTI[3] mem, reg; MOVNTPD/MOVNTPS; MOVNTQ | 0 | 1 | 1 |
| MOVQ mem, mm; MOVQ mm, mem; MOVDDUP | 0 | 1 | 1 |
| MOVSD/MOVSS xmm, xmm; MOVSXD[5] reg, reg | (0, 1) | 1 | 0.5 |
| MOVSD/MOVSS xmm, mem; PALIGNR | 0 | 1 | 1 |
| MOVSD/MOVSS mem, xmm; PINSRW | 0 | 1 | 1 |
| MOVSHDUP/MOVSLDUP xmm, mem | 0 | 1 | 1 |

| Instruction | Ports | Latency | Throughput |
|---|---|---|---|
| **DisplayFamily_DisplayModel** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** |
| MOVSHDUP/MOVSLDUP/MOVUPS xmm, xmm | (0, 1) | 1 | 0.5 |
| MOVSX/MOVZX r16, m8; MOVSX/MOVZX r16, r8 | 0 | 3; 2 | 1 |
| MOVSX/MOVZX reg[3], r/m8; MOVSX/MOVZX reg[3], r/m16 | 0 | 1 | 1 |
| MOVSXD[5] reg, mem; MOVSXD r64, r/m32 | 0 | 1 | 1 |
| MULPS/MULSD xmm, mem; MULSS xmm, mem; | 0 | 5; 4 | 2 |
| MULPS/MULSD xmm, xmm; MULSS xmm, xmm | 0 | 5; 4 | 2 |
| MULPD | B | 5; 4 | 2 |
| NEG/NOT[2] mem; PREFETCHNTA; PREFETCHTx | 0 | 10 | 9 |
| NEG/NOT[2] reg; NOP | (0, 1) | 1 | 0.5 |
| PABSB/D/W mm, mem; PABSB/D/W xmm, mem | 0 | 1 | 1 |
| PABSB/D/W mm, mm; PABSB/D/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PACKSSDW/WB mm, mem; PACKSSDW/WB xmm, mem | 0 | 1 | 1 |
| PACKSSDW/WB mm, mm; PACKSSDW/WB xmm, xmm | 0 | 1 | 1 |
| PACKUSWB mm, mem; PACKUSWB xmm, mem | 0 | 1 | 1 |
| PACKUSWB mm, mm; PACKUSWB xmm, xmm | 0 | 1 | 1 |
| PADDB/D/W/Q mm, mem; PADDB/D/W/Q xmm, mem | 0 | 1 | 1 |
| PADDB/D/W/Q mm, mm; PADDB/D/W/Q xmm, xmm | (0, 1) | 1 | 0.5 |
| PADDSB/W mm, mem; PADDSB/W xmm, mem | 0 | 1 | 1 |
| PADDSB/W mm, mm; PADDSB/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PADDUSB/W mm, mem; PADDUSB/W xmm, mem | 0 | 1 | 1 |
| PADDUSB/W mm, mm; PADDUSB/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PAND/PANDN/POR/PXOR mm, mem; PAND/PANDN/POR/PXOR xmm, mem | 0 | 1 | 1 |
| PAND/PANDN/POR/PXOR mm, mm; PAND/PANDN/POR/PXOR xmm, xmm | (0, 1) | 1 | 0.5 |
| PAVGB/W mm, mem; PAVGB/W xmm, mem | 0 | 1 | 1 |
| PAVGB/W mm, mm; PAVGB/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PCMPEQB/D/W mm, mem; PCMPEQB/D/W xmm, mem | 0 | 1 | 1 |
| PCMPEQB/D/W mm, mm; PCMPEQB/D/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PCMPGTB/D/W mm, mem; PCMPGTB/D/W xmm, mem | 0 | 1 | 1 |
| PCMPGTB/D/W mm, mm; PCMPGTB/D/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PEXTRW; | B | 4 | 1 |
| PHADDD/PHSUBD mm, mem; PHADDD/PHSUBD xmm, mem | B | 4 | 3 |
| PHADDD/PHSUBD mm, mm; PHADDD/PHSUBD xmm, xmm | B | 3 | 2 |
| PHADDW/PHADDSW mm, mem; PHADDW/PHADDSW xmm, mem | B | 6; 8 | 5;7 |
| PHADDW/PHADDSW mm, mm; PHADDW/PHADDSW xmm, xmm | B | 5; 7 | M |
| PHSUBW/PHSUBSW mm, mem; PHSUBW/PHSUBSW xmm, mem | B | 6; 8 | M |
| PHSUBW/PHSUBSW mm, mm; PHSUBW/PHSUBSW xmm, xmm | B | 5; 7 | M |

| Instruction | Ports | Latency | Throughput |
|---|---|---|---|
| **DisplayFamily_DisplayModel** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** |
| PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mem | 0 | 4 | 1 |
| PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, mem | 0 | 5 | 1 |
| PMAXSW/UB mm, mem; PMAXSW/UB xmm, mem | 0 | 1 | 1 |
| PMAXSW/UB mm, mm; PMAXSW/UB xmm, xmm | (0, 1) | 1 | 0.5 |
| PMINSW/UB mm, mem; PMINSW/UB xmm, mem | 0 | 1 | 1 |
| PMINSW/UB mm, mm; PMINSW/UB xmm, xmm | (0, 1) | 1 | 0.5 |
| PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mm; PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mem | 0 | 4 | 1 |
| PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, xmm; PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, mem | 0 | 5 | 1 |
| POP mem[5]; PSLLD/Q/W mm, mem; PSLLD/Q/W xmm, mem | B | 3 | 2 |
| POP r16; PUSH mem[4]; PSLLD/Q/W mm, mm; PSLLD/Q/W xmm, xmm | B | 2 | 1 |
| POP reg[3]; PUSH reg[4]; PUSH imm | B | 1 | 1 |
| POPA ; POPAD | B | 9 | 8 |
| PSHUFB mm, mem; PSHUFD; PSHUFHW; PSHUFLW; PSHUFW | 0 | 1 | 1 |
| PSHUFB mm, mm; PSLLD/Q/W mm, imm; PSLLD/Q/W xmm, imm | 0 | 1 | 1 |
| PSHUFB xmm, mem | B | 5 | 4 |
| PSHUFB xmm, xmm | B | 4 | 3 |
| PSIGNB/D/W mm, mem; PSIGNB/D/W xmm, mem | 0 | 1 | 1 |
| PSIGNB/D/W mm, mm; PSIGNB/D/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PSRAD/W mm, imm; PSRAD/W xmm, imm; | 0 | 1 | 1 |
| PSRLD/Q/W mm, mem; PSRLD/Q/W xmm, mem | B | 3 | 2 |
| PSRLD/Q/W mm, mm; PSRLD/Q/W xmm, xmm | B | 2 | 1 |
| PSRLD/Q/W mm, imm; PSRLD/Q/W xmm, imm; | 0 | 1 | 1 |
| PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS | 0 | 1 | 1 |
| PSUBB/D/W/Q mm, mem; PSUBB/D/W/Q xmm, mem | 0 | 1 | 1 |
| PSUBB/D/W/Q mm, mm; PSUBB/D/W/Q xmm, xmm | (0, 1) | 1 | 0.5 |
| PSUBSB/W mm, mem; PSUBSB/W xmm, mem | 0 | 1 | 1 |
| PSUBSB/W mm, mm; PSUBSB/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PSUBUSB/W mm, mem; PSUBUSB/W xmm, mem | 0 | 1 | 1 |
| PSUBUSB/W mm, mm; PSUBUSB/W xmm, xmm | (0, 1) | 1 | 0.5 |
| PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD | 0 | 1 | 1 |
| PUNPCKHQDQ; PUNPCKLQDQ | 0 | 1 | 1 |
| PUSHA ; PUSHAD | B | 8 | 7 |
| RCL mem[2], 1; RCL reg[2], 1 | 0 | 1 | 1 |

| Instruction | Ports | Latency | Throughput |
|---|---|---|---|
| **DisplayFamily_DisplayModel** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** | **06_1CH, 06_26H, 06_27H** |
| RCL m8, CL; RCL m16, CL; RCL mem[3], CL; | B | 18;16; 14 | 17;15;13 |
| RCL m8, imm; RCL m16, imm; RCL mem[3], imm; | B | 18; 17; 14 | 17;16;13 |
| RCL r8, CL; RCL r16, CL; RCL reg[3], CL; | B | 17; 16; 14 | 16;15;14 |
| RCL r8, imm; RCL r16, imm; RCL reg[3], imm; | B | 18;16; 14 | 17;15;13 |
| RCPSS | 0 | 4 | 1 |
| RCR mem[2], 1; RCR reg[2], 1 | B | 7; 5 | 6;4 |
| RCR m8, CL; RCR m16, CL; RCR mem[3], CL; | B | 15; 13; 12 | 14;12;11 |
| RCR m8, imm; RCR m16, imm; RCR mem[3], imm; | B | 16,;14; 12 | 15;13;11 |
| RCR r8, CL; RCR r16, CL; RCR reg[3], CL; | B | 14; 13; 12 | 13;12;11 |
| RCR r8, imm; RCR r16, imm; RCR reg[3], imm; | B | 15, 14, 12 | 14;13;11 |
| RET imm16 | B | 1 | 1 |
| RET (far) | B | 79 | |
| ROL; ROR; SAL; SAR; SHL; SHR | 0 | 1 | 1 |
| SETcc | | 1 | 1 |
| SHLD[8] mem, reg, imm; SHLD r64, r64, imm; SHLD m64, r64, CL | B | 11 | 10 |
| SHLD m32, r32; SHLD r32, r32 | B | 4; 2 | 3; 1 |
| SHLD m16, r16, CL; SHLD r16, r16, imm; SHLD r64, r64, CL | B | 10 | 9 |
| SHLD r16, r16, CL; SHRD m64, r64; SHRD r64, r64, imm | B | 9 | 8 |
| SHRD m32, r32; SHRD r32, r32 | B | 4; 2 | 3; 1 |
| SHRD m16, r16; SHRD r16, r16 | B | 6 | 5 |
| SHRD r64, r64, CL | B | 8 | 7 |
| STMXCSR | B | 15 | 14 |
| TEST[2] reg, reg; TEST[4] reg, imm | (0, 1) | 1 | 0.5 |
| UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS | 0 | 1 | 1 |

オペランドサイズ (osize) とアドレスサイズ (asize) に関する注意:

1. osize = 8, 16, 32 または asize = 8, 16, 32
2. osize = 8, 16, 32, 64
3. osize = 32, 64
4. osize = 16, 32, 64 または asize = 16, 32, 64
5. osize = 16, 32
6. osize = 8, 32
7. osize = 8, 16
8. osize = 16, 64

## 6.5 Silvermont† マイクロアーキテクチャー

Intel Atom® プロセッサー E3000 と C2000 シリーズは、Silvermont† マイクロアーキテクチャーをベースにしています。Silvermont† 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† 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。

- 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® マイクロアーキテクチャーでは、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。
- 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ーでは、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。
- 2 コアのモジュラーシステム設計。フロントサイド・バスの代わりにポイントツーポイント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。
- 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 6-2 に Silvermont† 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の設計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。

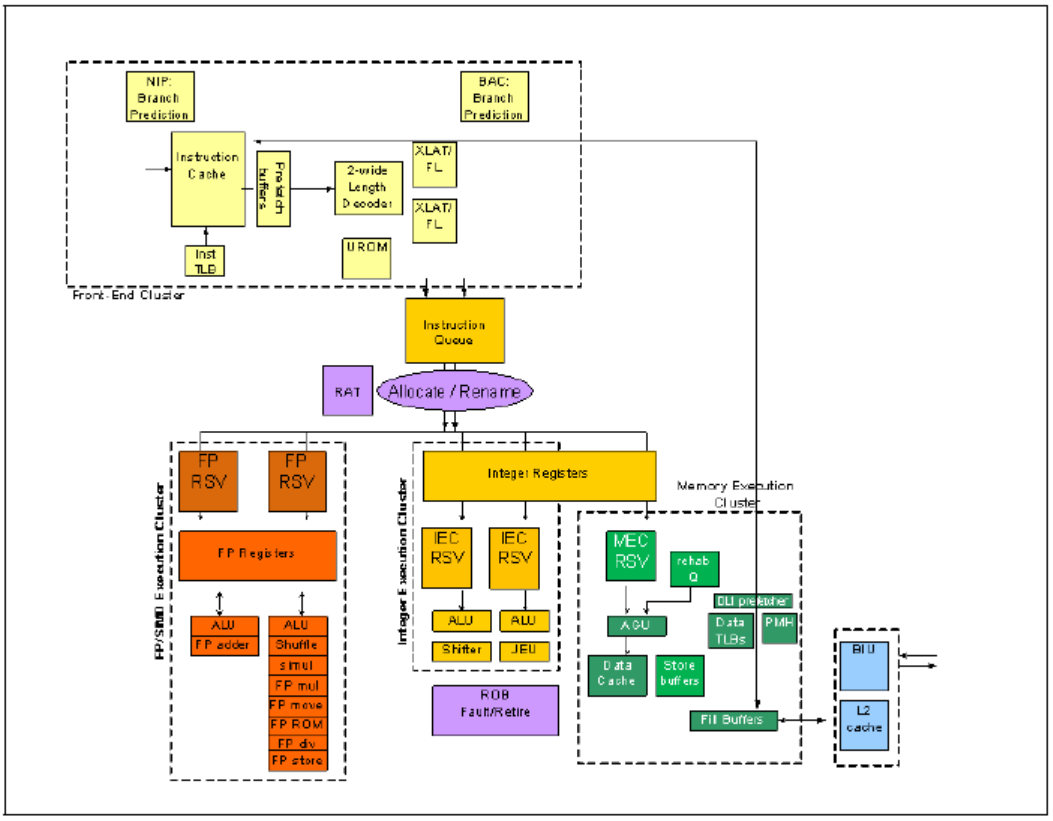


図 6-2 Silvermont† マイクロアーキテクチャーのパイプライン

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 6-2 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 6-2 の紫色)。ARR は、FEC からマイクロオペレーション (uop) 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 (uop) 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 6-2 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 クラスターからサイクルごとに最大 2 マイクロオペレーション (uop) 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 クラスターと FPC クラスターはインオーダー・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 6-3 の Silvermont† 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を示します。図 6-2 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® マイクロアーキテクチャーと比べると、Silvermont† 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 6-3 Silvermont† 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[1] |
| 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 (uop) 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ー (MEC) (図 6-2 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† 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† 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。

バスクラスター (BIU) の L2 キャッシュは、プロセッサー・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® 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® プロセッサーのフロントサイド・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・プリフェ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シング (CMP) システム構成では、2 つのプロセッサー・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ッ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。

## 6.5.1 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ーが前世代の Intel Atom® プロセッサーよりも 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® プロセッサーと同じで、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 (uop) 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® プロセッサーによく似ています。マイクロオペレーション (uop) 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。

## 6.5.2 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† 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。

## 6.6 Goldmont† マイクロアーキテクチャー

Goldmont† マイクロアーキテクチャーは、Silvermont† マイクロアーキテクチャー (6.5 節を参照) の成功を基に、次の拡張を提供します。

- 3 ワイド・スーパースカラー・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 uop をリザベーション・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。

- 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。
- 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。
- 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† 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† 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。
- Goldmont† 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。
- 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、PSHUFB 命令は 1 サイクルのスループット (Silvermont† 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 16-17 を参照してください。
- Goldmont† 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。
- Goldmont† 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。
- Goldmont† 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。
- 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。

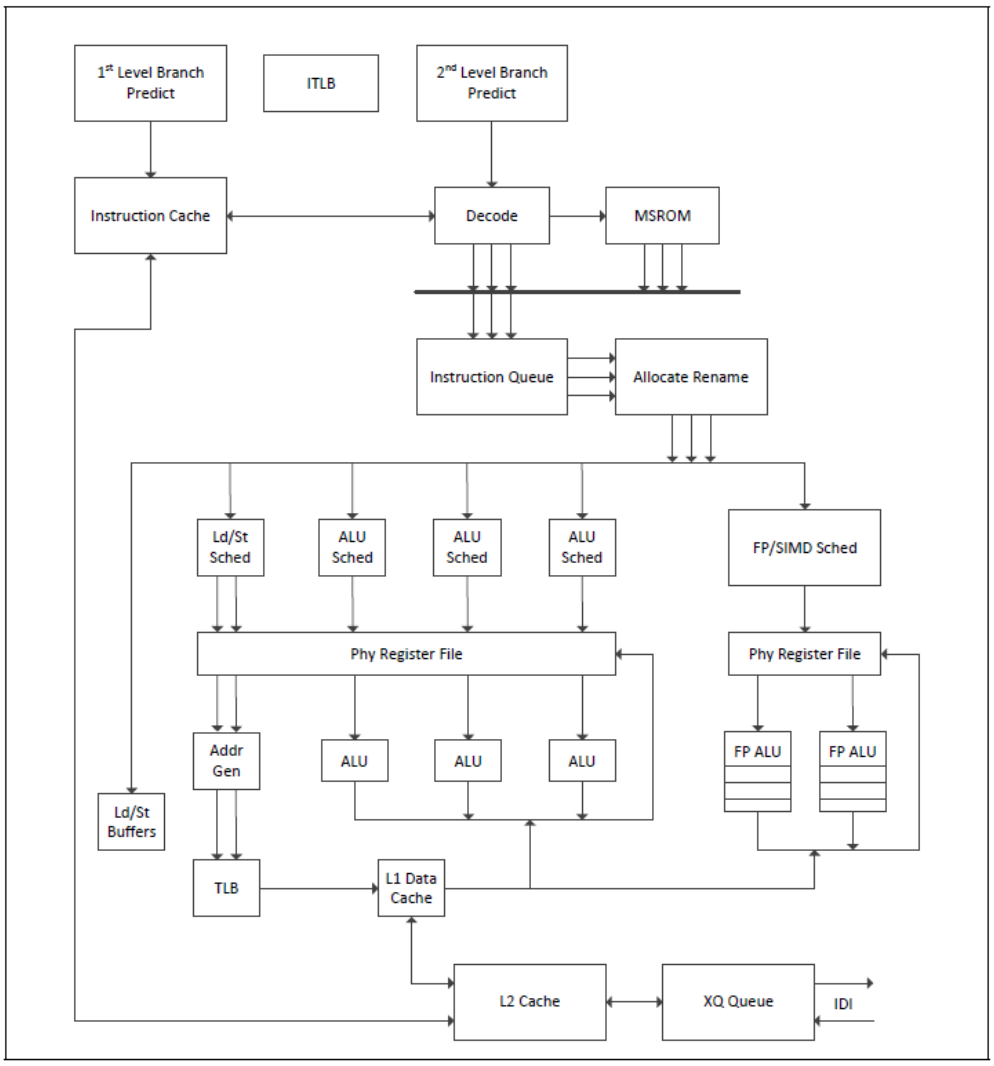


図 6-3 Goldmont† マイクロアーキテクチャーの CPU コア･パイプラインの機能

Goldmont† 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† 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 6-4 にこれらの拡張をまとめています。

表 6-4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。uop 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たリザベーション･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 uop 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 F-5 は、Goldmont† マイクロアーキテクチャーと Silvermont† 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 6-5 uop をスケジューリングする際の分散リザベーション･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 uop が MEC クラスターへ送られ、整数/FP uop が IEC/FPC クラスターへ送られます。そしてリソースが利用可能になると、表 6-5 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 6-6 は、それぞれのクラスター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 6-6 Goldmont† 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[1], 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 (uop) 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ーとストアバッファー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、未知のストアとロードに依存関係があると、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 Plus† 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 (µTLB 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ーに戻されます。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/ストアバッファーから再発行 (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･バッファー (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 6-7 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 |

## 6.7 Goldmont Plus† マイクロアーキテクチャー

Goldmont Plus† マイクロアーキテクチャーは、Goldmont† マイクロアーキテクチャー (6.6 節を参照) の成功を基に、次の拡張を提供します。

- 前世代の Intel Atom® プロセッサーに比べ、バックエンド・パイプラインが 4 ワイド割り当てと 4 ワイドリタイアに拡張されていますが、3 ワイドフェッチとデコード・パイプラインを維持します。
- 拡張分岐予測ユニット
- 64KB の共有第 2 レベル事前デコードキャッシュ (Goldmont† マイクロアーキテクチャーでは 16KB)
- 広いアウトオブオーダー・ウィンドウをサポートする、大きなリザベーション・ステーションと ROB エントリー
- さらに広い整数実行ユニット高速な分岐のリダイレクションをサポートする新しい専用 JEU ポート
- 高速なスカラー/パックド単精度、倍精度、および拡張精度の浮動小数点除算を実現する Radix-1024 浮動小数点除算器
- インテル® AES-NI 命令のレイテンシーとスループットを改善
- 大きなロードとストア・バッファー・レジスターからデータをストアするストア・ロード・フォワーディングのレイテンシーを改善
- 命令およびデータ向けの共有第 2 レベル TLB ページングキャッシュの拡張 (PxE/ePxE キャッシュ)
- 最大 4MB L2 キャッシュを 4 コアで共有するモジュラーシステム設計
- 新たにプロセッサー ID 読みとり (RDP) 命令をサポート

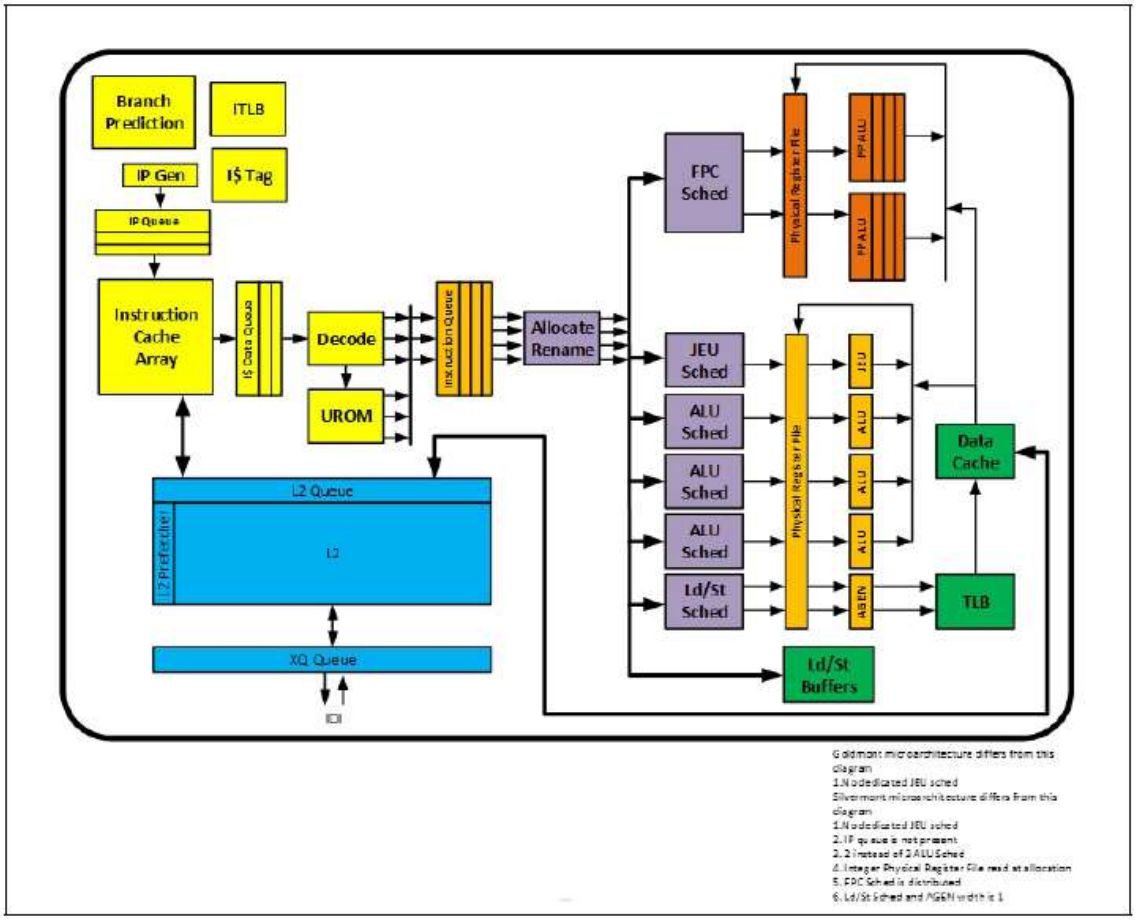


図 6-4 Goldmont Plus† マイクロアーキテクチャーの CPU コア･パイプラインの機能

Goldmont Plus† 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、Goldmont† 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 6-8 にそれらの拡張をまとめています。

表 6-8 フロントエンド･クラスター機能の比較

| **機能** | **Goldmont Plus† マイクロアーキテクチャー** | **Goldmont† マイクロアーキテクチャー** |
|---|---|---|
| デコーダー数 | 3 | 3 |
| デコーダーの最大スループット | サイクルごとに 20 バイト | サイクルごとに 20 バイト |
| フェッチと命令キャッシュ･パイプライン | 分離 | 分離 |
| 命令 TLB (ITLB) | 48 エントリー、ラージページをサポート | 48 エントリー、ラージページをサポート |
| 2 次 ITLB キャッシュ | DTLB と共有 | |
| 分岐予測ミスのペナルティー | 13 サイクル (特定の Jcc では 12 サイクル) | 12 サイクル |
| L2 事前デコードキャッシュ | 64K | 16K |

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。uop 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たリザベーション･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 uop 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 6-9 は、Goldmont Plus† マイクロアーキテクチャーと Goldmont† 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 6-9 uop をスケジューリングする際の分散リザベーション･ステーションの比較

| クラスター | Goldmont Plus† マイクロアーキテクチャー | Goldmont† マイクロアーキテクチャー |
|---|---|---|
| IEC リザベーション | 4x 各ポートに分散 | 3x 各ポートに分散 |
| | それぞれの IEC RSV 内と IEC と FPC、MEC をまたがるアウトオブオーダー | それぞれの IEC RSV 内と IEC と FPC、MEC をまたがるアウトオブオーダー |
| FPC リザベーション | 1x ポート 0 と 1 へ統合 | 1x ポート 0 と 1 へ統合 |
| | FPC RSV 内と IEC と MEC をまたがるアウトオブオーダー | FPC RSV 内と IEC と MEC をまたがるアウトオブオーダー |
| MEC リザベーション | 1x ポート 0 と 1 へ統合 | 1x ポート 0 と 1 へ統合 |
| | MEC RSV 内と IEC と FPC をまたがるアウトオブオーダー | MEC RSV 内と IEC と FPC をまたがるアウトオブオーダー |

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 uop が MEC クラスターへ送られ、整数/FP uop が IEC/FPC クラスターへ送られます。そしてリソースが利用可能になると、表 6-9 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 6-10 は、それぞれのクラスター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 6-10 Goldmont Plus† マイクロアーキテクチャーの機能ユニットの割り当て

| クラスター | ポート 0 | ポート 1 | ポート 2 | ポート 3 |
|---|---|---|---|---|
| IEC | ALU0, シフト/ローテート、LEA インデックスなし、F2I、変換/cmp、ストアデータ | ALU1、ビット処理、IMUL、IDIV、POPCNT、CRC32、LEA、I2F、ストアデータ | ALU2、LEA[1]、I2F、フラグマージ | JEU |
| FPC | SIMD ALU, SIMD シフト/シャッフル、SIMD 乗算、STTNI/AESNI/PCLMULQDQ/SHA ; FP_乗算、変換、F2I 変換 | SIMD ALU、SIMD シャッフル、FP_加算、F2I 比較 | | |
| MEC RS | ロードアドレス | ストアアドレス | | |

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 (uop) 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ーとストアバッファー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、未知のストアとロードに依存関係があると、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 Plus† 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 (µTLB 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ーに戻されます。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/ストアバッファーから再発行 (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･バッファー (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。

Goldmont Plus† マイクロアーキテクチャーは、データと命令の両方の変換をサポートする第 2 レベル TLB を搭載しています (Goldmont† マイクロアーキテクチャーでは、第 2 レベル TLB はデータのみをサポート)。

## 6.8 コーディングの推奨事項

『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 Volume 1』の第 3 章「一般的な最適化ガイドライン」で説明されている一般的なコーディングの推奨事項は、Intel Atom® 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 Intel Atom® 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。

### 6.8.1 フロントエンドの最適化

#### 6.8.1.1 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 (uop) 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、6.9 節のレイテンシー/スループットの表を参照してください。

Silvermont† 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 Plus† と Goldmont† 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† 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 6-11 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 6-11 MSROM 命令の代替

| Instruction from MSROM | Recommendation for Silvermont | Recommendation for Goldmont Plus and Goldmont |
|---|---|---|
| CALL m16/m32/m64 | Load + CALL reg | Load + CALL reg |
| PUSH m16/m32/m64 | Load + PUSH reg | Use as is (non MSROM) |
| LEAVE | No recommended replacement | Use as is (non MSROM) |
| FLD/FST/FSTP m80fp | No recommended replacement | Use as is (non MSROM) |
| 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 MSROM) |
| (I)MUL r/m64 (Result RDX:RAX) | Use (I)MUL r64, r/m64 if extended precision not required | Use as is (non MSROM) |
| PEXTRB/D/Q | No recommended replacement | Use as is (non MSROM) |
| PMULLD | No recommended replacement | Use as is (non MSROM) |

**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、MSROM 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ールトが含まれる)。

**アセンブリー/コンパイラー・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。

### 6.8.1.2 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。

- MSROM 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、レジスターへロードし、レジスターバージョンの PUSH と CALL を実行します。
- 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† 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† マイクロアーキテクチャー以降では、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。

- 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† 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† マイクロアーキテクチャー以降: 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† と Goldmont† 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。

- 各サイクルでデコード可能な分岐の最大数は、Silvermont† マイクロアーキテクチャーでは 1、Goldmont† 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† 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† 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† マイクロアーキテクチャー以降では、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® プロセッサーにおいて、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。

**アセンブリー/コンパイラー・コーディング規則 2 (影響 MH、一般性 H):** サイクルあたり 2 命令のスループ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、または (iv) 連続する分岐命令。

例えば、通常 Silvermont† と Goldmont† マイクロアーキテクチャー以降では、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† マイクロアーキテクチャー以降では通常どおりにデコードされますが、Silvermont† 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ーを記述するには、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† と Goldmont† マイクロアーキテクチャー以降に適用されます。

表 6-12 は、Silvermont†、Goldmont† および Goldmont Plus† 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 6-12 デコーダーの能力の比較

| | Goldmont Plus and 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 |

### 6.8.1.3 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。

#### 6.8.1.4 ループアンロールおよびループストリーム検出器

Silvermont† と Goldmont† マイクロアーキテクチャー以降は、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 (uop) 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荷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† マイクロアーキテクチャー以降の 3 ワイドのデコーダーでは 27 命令未満に、Silvermont† 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 BACLEARS.ANY 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ないかを確認します。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。

#### 6.8.1.5 コードとデータの混在

Intel Atom® プロセッサーは、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。

### 6.8.2 実行コアの最適化

#### 6.8.2.1 スケジューリング

Silvermont† 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† マイクロアーキテクチャー以降は、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 uop_NOT_DELIVERED.ANY (Silvermont† 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。

#### 6.8.2.2 アドレス生成

前世代の Intel Atom® マイクロアーキテクチャーのアドレス生成の制限は、Silvermont† 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、Silvermont† マイクロアーキテクチャー以降では、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 を使用するか、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。

### 6.8.2.3 FP 乗算-加算-ストアの実行

Goldmont† マイクロアーキテクチャー以降は、統合された FPC リザベーション・ステーションにより、Silvermont† マイクロアーキテクチャーで FPC uop 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 6-7 はこの問題を示します。

Silvermont† 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 6-7 について考えてみます。

例 6-7 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 | EX1 | EX2 | EX3 | EX4 | EX5 | | | | | | | | | | | | | |
| addps xmm1, xmm1 | | | | | | EX1 | EX2 | EX3 | | | | | | | | | | |
| movaps mem, xmm1 | | | | | | | | | EX1 | | | | | | | | | |
| mulps, xmm2, xmm2 | | | | | | | | | | EX1 | EX2 | EX3 | EX4 | EX5 | | | | |
| addps xmm2, xmm2 | | | | | | | | | | | | | | | EX1 | EX2 | EX3 | |
| movaps mem, xmm2 | | | | | | | | | | | | | | | | | | EX1 |

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 6-8 スト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 |

### 6.8.2.4 整数乗算の実行

Silvermont† マイクロアーキテクチャー以降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 6-13 に MSROM 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 (uop) 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 6-13 整数乗算命令のレイテンシー

| Integer Multiply Operations | Output | Goldmont Plus and Goldmont Latency | Silvermont Latency |
|---|---|---|---|
| imul/mul r/m8 | 16 | 4[u] | 5[u] |
| imul/mul r/m16 | 32 | 4[u] | 5[u] |
| imul/mul r/32 | 64 | 3 | 4[u] |
| imul/mul r/m64 | 128 | 5 | 7[u] |
| imul/mul r16, r/m16; r16, r/m16, imm | 16 | 4[u] | 4[u] |
| 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 | | | |

### 6.8.2.5 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。

Silvermont† マイクロアーキテクチャー以降には、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† 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† マイクロアーキテクチャー以降では、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。

### 6.8.2.6 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† マイクロアーキテクチャー以降は、NOP をリザベーション・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。

### 6.8.2.7 ムーブの排除 (Move Elimination) と ESP の折りたたみ (Folding)

ムーブの排除は、Goldmont† マイクロアーキテクチャー以降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/MOVAPD/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。MOVSX 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† マイクロアーキテクチャー以降では Silvermont† マイクロアーキテクチャーよりも効率的です。Goldmont† マイクロアーキテクチャー以降では、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。

### 6.8.2.8 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 uop 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† マイクロアーキテクチャー以降では、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† 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。

### 6.8.2.9 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 (uop) 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。

### 6.8.2.10 SIMD 浮動小数点と x87 命令

Silvermont† 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† マイクロアーキテクチャー以降では、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† マイクロアーキテクチャー以降を Silvermont† 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† 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 F-19 を参照してください。

Goldmont Plus† マイクロアーキテクチャーでは、浮動小数点除算器が Radix-1024 ベースを使用した設計に改良され、浮動小数点除算と平方根のレイテンシーと帯域幅が大幅に改善されました。

### 6.8.2.11 SIMD 整数命令

Silvermont† 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† マイクロアーキテクチャー以降の場合、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† マイクロアーキテクチャー以降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† 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† マイクロアーキテクチャー以降では、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† 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† 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† マイクロアーキテクチャー以降では、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† マイクロアーキテクチャー以降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† 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 19-17 を参照してください。

### 6.8.2.12 ベクトル化の注意事項

Silvermont† 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 uop 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† マイクロアーキテクチャー以降は、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。

### 6.8.2.13 その他の SIMD 命令

Silvermont† 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† マイクロアーキテクチャー以降では、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† 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† マイクロアーキテクチャー以降では、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† マイクロアーキテクチャー以降は、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参照してください。

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)

### 6.8.2.14 命令の選択

表 6-14 に、Silvermont† 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 6-14 浮動小数点と SIMD 整数のレイテンシー

| | Goldmont Plus | | Goldmont | | Silvermont | |
|---|---|---|---|---|---|---|
| | Latency | Through put | Latency | Through put | Latency | Throughp ut |
| SIMD integer ALU | | | | | | |
| 128-bit ALU/logical/move | 1 | 0.5 | 1 | 0.5 | 1 | 0.5 |
| 64-bit ALU/logical/move | 1 | 0.5 | 1 | 0.5 | 1 | 0.5 |
| SIMD integer shift | | | | | | |
| 128-bit | 1 | 0.5 | 1 | 0.5 | 1 | 1 |
| 64-bit | 1 | 0.5 | 1 | 0.5 | 1 | 1 |
| SIMD shuffle | | | | | | |
| 128-bit | 1 | 0.5 | 1 | 0.5 | 1 | 1 |
| 64-bit | 1 | 0.5 | 1 | 0.5 | 1 | 1 |
| SIMD integer multiplier | | | | | | |
| 128-bit | 4 | 1 | 4 | 1 | 5 | 2 |
| 64-bit | 4 | 1 | 4 | 1 | 4 | 1 |

| | Goldmont Plus | | Goldmont | | Silvermont | |
|---|---|---|---|---|---|---|
| | Latency | Through put | Latency | Through put | Latency | Throughp ut |
| FP Adder | | | | | | |
| x87 (fadd) | 3 | 1 | 3 | 1 | 3 | 1 |
| scalar (addsd, addss) | 3 | 1 | 3 | 1 | 3 | 1 |
| packed (addpd, addps) | 3 | 1 | 3 | 1 | 4 | 2 |
| FP Multiplier | | | | | | |
| x87 (fmul) | 5 | 2 | 5 | 2 | 5 | 2 |
| scalar single-precision (mulss) | 4 | 1 | 4 | 1 | 4 | 1 |
| scalar double-precision (mulsd) | 4 | 1 | 4 | 1 | 5 | 2 |
| packed single-precision (mulps) | 4 | 1 | 4 | 1 | 5 | 2 |
| packed double-precision (mulpd) | 4 | 1 | 4 | 1 | 7 | 4 |
| Converts | | | | | | |
| CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTTPD2DQ, CVTPD2PI, CVTPS2DQ | 4 | 1 | 4 | 1 | 5 | 2 |
| CVTPI2PS, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI | 4 | 1 | 4 | 1 | 4 | 1 |
| FP Divider | | | | | | |
| x87 fdiv (extended-precision) | 15 | 11 | 39 | 39 | 39 | 39 |
| x87 fdiv (double-precision) | 14 | 10 | 34 | 34 | 34 | 34 |
| x87 fdiv (single-precision) | 11 | 7 | 19 | 19 | 19 | 19 |
| scalar single-precision (divss) | 11 | 7 | 19 | 18 | 19 | 17 |
| scalar double-precision (divsd) | 14 | 10 | 34 | 33 | 34 | 32 |
| packed single-precision (divps) | 16 | 12 | 36 | 35 | 39 | 39 |
| packed double-precision (divpd) | 22 | 18 | 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。

**アセンブリー/コンパイラー・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。

**アセンブリー/コンパイラー・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。

### 6.8.2.15 整数除算

Silvermont† 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† マイクロアーキテクチャー以降では、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 6-15 と表 6-16 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 6-15 符号なし整数除算操作のレイテンシー

| | Dividend | Divisor | Quotient | Remainder | Silvermont[u] | Goldmont |
|---|---|---|---|---|---|---|
| DIV r8 | AX | r8 | AL | AH | 25 | 11-12 |
| DIV r16 | DX:AX | r16 | AX | DX | 26-30 | 12-17[u] |
| DIV r32 | EDX:EAX | r32 | EAX | EDX | 26-38 | 12-25[u] |
| DIV r64 | RDX:RAX | r64 | RAX | RDX | 38-123 | 12-41[u] |

表 6-16 符号付き整数除算操作のレイテンシー

| | Dividend | Divisor | Quotient | Remainder | Silvermont[u] | Goldmont |
|---|---|---|---|---|---|---|
| IDIV r8 | AX | r8 | AL | AH | 34 | 11-12 |
| IDIV r16 | DX:AX | r16 | AX | DX | 35-40 | 12-17[u] |
| IDIV r32 | EDX:EAX | r32 | EAX | EDX | 35-47 | 12-25[u] |
| IDIV r64 | RDX:RAX | r64 | RAX | RDX | 49-135 | 12-41[u] |

**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。

### 6.8.2.16 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† 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† マイクロアーキテクチャー以降では、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† と Goldmont† の両方で MSROM からのデコードを必要とします。

#### 6.8.2.17 ポーズ命令

Goldmont† マイクロアーキテクチャー以降では、PAUSE 命令のレイテンシーは Silvermont† 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。

### 6.8.3 メモリーアクセスの最適化

#### 6.8.3.1 PALIGNR でアライメントされていないメモリーアクセスを軽減

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ー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ー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。

**アセンブリー/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。

#### 6.8.3.2 メモリー実行の問題を最小化する

Goldmont† マイクロアーキテクチャー以降では、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。ロードが古いストアに依存している場合、ハードウェアがその状況を検出して、ロードとそれ以降の命令を再実行する必要があります。このような再実行が行われると、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† 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† マイクロアーキテクチャーの perfmon カウンター REHABQ は、Silvermont† 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† マイクロアーキテクチャーと前の世代において、古い未知のストアをロードがパスする場合のアプリケーション・パフォーマンスへの影響を評価できます。

#### 6.8.3.3 ストア・フォワーディング

Silvermont† マイクロアーキテクチャー以降では、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。

- ストア操作とロード操作の開始アドレスが同じである。
- ロード操作の幅がストア操作の幅以下である。
- 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 6-17 と表 6-18 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 6-17 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)

| | | Load Alignment | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Store Size | Load Size | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 1 | F | | | | | | | | | | | | | | | |
| 2 | 1 | F | N | | | | | | | | | | | | | | |
| | 2 | F | N | | | | | | | | | | | | | | |

表 6-18 ストア・フォワーディング条件 (4-16 バイト・ストア)

| | | Load Alignment | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Store Size | Load Size | 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)。

- メモリーの代わりにレジスターを使用する。
- 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† 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† 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。Goldmont Plus† マイクロアーキテクチャーでは、レジスター操作からのストアデータを最適化しており、ロードフォワードへのストアのレイテンシーを 3 サイクルに軽減しています。

## 6.8.3.4 PrefetchW 命令

Silvermont† マイクロアーキテクチャー以降は、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。

### 6.8.3.5 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† 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® プロセッサーとは対照的に、MOVUPS/MOVUPD/MOVDQU 命令はすべて単一のマイクロオペレーション (uop) 命令となっています。

### 6.8.3.6 セグメントベース

簡略化のため、Silvermont† 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† 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† マイクロアーキテクチャー以降では、サイクルごとに 1 つの NZB アドレス生成が可能です。

### 6.8.3.7 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 MOVS/STOS 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 MOVS/STOS 命令には固有のオーバーヘッドがあります。REP STOS 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 MOVS 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、バッファー長、バッファー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 MOV のパフォーマンスを大幅に向上させます。

Silvermont† マイクロアーキテクチャー・ベースのプロセッサーは、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ーは、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ー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、将来のプロセッサーでもある程度のパフォーマンスが期待できます。

## 6.9 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® プロセッサー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。

- Goldmont Plus† マイクロアーキテクチャー: 06_7AH。Goldmont Plus† マイクロアーキテクチャーの値が Goldmont† マイクロアーキテクチャーと異なる場合、表中に「GLP」を明記しています。
- Goldmont† マイクロアーキテクチャー: 06_5CH, 06_5FH.
- Silvermont† または Airmont† 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 6-19 Intel Atom® プロセッサー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, 7AH | 06_37H, 4AH,4CH ,4DH,5AH,5DH | 06_5CH, 5FH, 7AH | 06_37H, 4AH,4CH, 4DH,5AH, 5DH | 06_5CH ,5FH, 7AH | 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 |
| ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS | 0.5 | 0.5 | 1 | 1 | N | N |
| AESDEC/AESDECLAST/AESENC/AESENCLAST | 2<br>1 (GLP) | 5 | 6<br>4 (GLP) | 8 | N | Y |
| AESIMC/AESKEYGEN | 2<br>1 (GLP) | 5 | 5<br>4 (GLP) | 8 | N | Y |
| BLENDPD/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 |
| CMOVxx 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 |

表 6-19 Intel Atom® プロセッサー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, 7AH | 06_37H, 4AH,4CH ,4DH,5AH,5DH | 06_5CH, 5FH, 7AH | 06_37H, 4AH,4CH, 4DH,5AH, 5DH | 06_5CH ,5FH, 7AH | 06_37H, 4AH,4CH,4DH,5AH,5DH |
| 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 |
| CVTSI2SD/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[1] | 12, 65<br>18 (GLP) | 27-69 | 13, 66<br>22 (GLP) | 27-69 | N | Y |
| DIVPS[1] | 12,35<br>12 (GLP) | 27-39 | 13, 36<br>16 (GLP) | 27-39 | N | Y |
| DIVSD[1] | 12,33<br>10 (GLP) | 11-32 | 13,34<br>14 (GLP) | 13-34 | N | N |
| DIVSS[1] | 12,18<br>7 (GLP) | 11-17 | 13,19<br>11 (GLP) | 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<br>11 (EP GLP) | 39 | 39<br>15 (EP GLP) | 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 |

表 6-19 Intel Atom® プロセッサー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, 7AH | 06_37H, 4AH,4CH ,4DH,5A H,5DH | 06_5CH, 5FH, 7AH | 06_37H, 4AH,4CH, 4DH,5AH, 5DH | 06_5CH ,5FH, 7AH | 06_37H, 4AH,4C H,4DH,5 AH,5DH |
| 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 |
| 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[2]/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/MOVHLPS/MOVLHPS/MOVSHDUP/MOVSLDUP | 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 |

表 6-19 Intel Atom® プロセッサー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, 7AH | 06_37H, 4AH,4CH ,4DH,5AH,5DH | 06_5CH, 5FH, 7AH | 06_37H, 4AH,4CH, 4DH,5AH, 5DH | 06_5CH ,5FH, 7AH | 06_37H, 4AH,4CH,4DH,5AH,5DH |
| 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 |
| PCMPESTRI xmm, xmm, imm | 13 | 21 | 19(C)/ 26(F)[3] | 21(C)/ 28(F) | Y | Y |
| PCMPESTRM xmm, xmm, imm | 14 | 17 | 15(X)/ 25(F)[1] | 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)[1] | 17(C)/ 24(F) | Y | Y |
| PCMPISTRM xmm, xmm, imm | 7 | 13 | 10(X)/ 20(F)[1] | 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 |
| PHADDD/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 |
| PMAXSB/W/D xmm, xmm; PMAXUB/W/D xmm, xmm | 0.5 | 0.5 | 1 | 1 | N | N |
| PMINSB/W/D xmm, xmm; PMINUB/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 |
| PSHUFHW; PSHUFLW; PSHUFW | 0.5 | 1 | 1 | 1 | N | N |
| PSIGNB/D/W xmm, xmm | 0.5 | 1 | 1 | 1 | N | N |

表 6-19 Intel Atom® プロセッサー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, 7AH | 06_37H, 4AH,4CH ,4DH,5AH,5DH | 06_5CH, 5FH, 7AH | 06_37H, 4AH,4CH, 4DH,5AH, 5DH | 06_5CH ,5FH, 7AH | 06_37H, 4AH,4CH,4DH,5AH,5DH |
| PSLLDQ/PSRLDQ xmm, imm; SHUFPD/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 |
| 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<br>26 (GLP) | 70 | 68<br>30 (GLP) | 71 | N | Y |
| SQRTPS | 37<br>14 (GLP) | 40 | 38<br>18 (GLP) | 41 | N | Y |
| SQRTSD | 34<br>14 (GLP) | 35 | 35<br>18 (GLP) | 35 | N | Y |
| SQRTSS | 19<br>8 (GLP) | 20 | 20<br>12 (GLP) | 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 |

表 6-19 Intel Atom® プロセッサー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, 7AH | 06_37H, 4AH,4CH ,4DH,5AH,5DH | 06_5CH, 5FH, 7AH | 06_37H, 4AH,4CH, 4DH,5AH, 5DH | 06_5CH ,5FH, 7AH | 06_37H, 4AH,4CH,4DH,5AH,5DH |
| 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)

**注意**

最新のプロセッサーのレイテンシーとスループットの情報は、インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発マニュアルページ (英語)で公開されています。

この章には、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。[1] 命令タイミングデータは、プロセッサーのファミリー/モデルによって異なります。本章は、以下の節で構成されています。

- **7.1 節「概要」** — 命令の選択とスケジューリングに関係する問題の概要を説明しています。
- **7.2 節「用語説明」** — 本章で使用する用語の定義を示します。
- **7.3 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。

## 7.1 概要

この章には、アセンブリー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。

- キャッシュ・ミス・レイテンシー
- バス帯域幅
- I/O 帯域幅

第 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。

- ストア・フォワーディングの制約事項を守る。
- キャッシュラインの分割やメモリー・オーダー・バッファーの分割を避ける。
- 分岐予測を妨げない。
- 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、本章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。

- 実行コアへのマイクロオペレーション (uop) 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 (uop) 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 (uop) 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 (uop) 供給が停止します。

  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、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 (uop) 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 (uop) 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。

[1] 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。

- 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、スループ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 回というスループットになります。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プットが得られます。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、スループットの高い命令を選択するのも有効です。

- 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™ プロファイラーが含まれます。

## 7.2 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。

- **命令名** — 各命令のアセンブリー・ニーモニック。
- **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 (uop) 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。
- **スループット** — 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、命令のスループットの方がレイテンシーよりもかなり小さくなります。
- RDRAND 命令のレイテンシーとスループ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ー・コアと論理プロセッサーに依存します。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プットは、~100 サイクル程度に抑えることができます。Ivy Bridge マイクロアーキテクチャー・ベースの第 3 世代インテル® Core™ プロセッサーでは、アンコアの RDRAND による乱数送出の総帯域幅は、およそ 500MB/秒です。同一プロセッサー・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プットは、インテル® Core™ プロセッサーとインテル® Xeon® プロセッサーでは異なります。

## 7.3 レイテンシーとスループット

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プ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。

- 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 (uop) を並列実行する能力を制限します。命令のスループット・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。
- 数値はすべて概算値です。
  - マイクロアーキテクチャー (uop) 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ー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。

7.3.1 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。7.3.3 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。

## 7.3.1 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 7-4 から表 7-18 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。異なるプロセッサー・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。

プロセッサーの命令タイミングデータはプロセッサー固有であり、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® プロセッサーにも適用されます。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」は任意のステッピング・エンコーディングを示します。

インテル® Core™ Solo プロセッサーとインテル® Core™ Duo プロセッサーは、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・ベースのプロセッサーは、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ーは、06_17H と 06_1DH で示されます。Nehalem† マイクロアーキテクチャー・ベースのプロセッサーの CPUID ファミリー/モデル・シグネチャーは、06_1AH、06_1EH、6_1FH、および 06_2EH です。

Westmere† マイクロアーキテクチャー・ベースのプロセッサーは、06_25H、6_2CH、および 06_2FH です。

Sandy Bridge† マイクロアーキテクチャー・ベースのプロセッサーは、06_2AH および 06_2DH です。

Ivy Bridge† マイクロアーキテクチャー・ベースのプロセッサーは、06_3AH と 06_3EH です。

Haswell† マイクロアーキテクチャー・ベースのプロセッサーは、06_3CH、06_45H、および 06_46H です。

表 7-1 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† マイクロアーキテクチャー |
| 06_3DH, 06_47H, 06_56H | Broadwell† マイクロアーキテクチャー |
| 06_3CH, 06_45H, 06_46H, 06_3FH | Haswell† マイクロアーキテクチャー |
| 06_3AH, 06_3EH | Ivy Bridge† マイクロアーキテクチャー |
| 06_2AH, 06_2DH | Sandy Bridge† マイクロアーキテクチャー |
| 06_25H, 06_2CH, 06_2FH | Westmere† マイクロアーキテクチャー |
| 06_1AH, 06_1EH, 06_1FH, 06_2EH | Nehalem† 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 7-2 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ー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 7-2 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 |
| ADX, RDSEED | はい | はい | 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 |
| F16C, RDRAND, RWFSGSBASE | はい | はい | はい | はい | 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 |
| AESNI, PCLMULQDQ | はい | はい | はい | はい | はい | はい | いいえ | いいえ |
| SSE4.2, POPCNT | 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 |
| SSSE3 | 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 |
| SSE2 | 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 |
| MMX | はい | はい | はい | はい | はい | はい | はい | はい |

表 7-3 BMI1、BMI2 および汎用命令

| 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_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 7-4 256 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 |
| VPBLENDD | 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[b] | 5 | 5 | 0.5 | 1 | 1 |
| VPMADDUBSW | 5[b] | 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[b] | 5 | 5 | 0.5 | 1 | 1 |
| VPMULHRSW | 5[b] | 5 | 5 | 0.5 | 1 | 1 |
| VPMULHW/LW | 5[b] | 5 | 5 | 0.5 | 1 | 1 |
| VPMULLD | 10[b] | 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 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** |
| 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[d], ymm | | | | 1 | 2 | 2 |
| VPMASKMOVD/Q NUL, msk_0, ymm | | | | >200[e] | 2 | 2 |
| VPMASKMOVD/Q ymm, ymm[d], mem | 11 | 8 | 8 | 1 | 2 | 2 |
| VPMASKMOVD/Q ymm, msk_0, [base+index][f] | >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 7-5 L1D* からタイミングデータを収集

| 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** |
| 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 7-6 BMI1、BMI2 および汎用命令

| 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** |
| 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 7-7 F16C、RDRAND 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_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 |

**エラー! 参照元が見つかりません。**節を参照してください。

表 7-8 256 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_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[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_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** |
| 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, | 1 | 1 | 1 | 1 | 0.33 | 1 | 1 | 1 |

| 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_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** |
| ymm3 | | | | | | | | |
| 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 |
| VSHUFPD/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[a], ymm, ymm | | | | | 1 | 2 | 2 | 2 |
| VMASKMOVPD/PS NUL, msk_0, ymm | | | | | >200[b] | 2 | 2 | 2 |
| VMASKMOVPD/PS ymm, ymm[a], mem | 11 | 8 | 8 | 9 | 1 | 2 | 2 | 2 |
| VMASKMOVPD/PS ymm, msk_0, [base+index][c] | >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 のカラムに示されます。

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 7-9 インテル® AES-NI と PCLMULQDQ 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_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** |
| 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[b] | 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 7-10 インテル® SSE4.2 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_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** |
| CRC32 r32, r32 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 |
| PCMPESTRI xmm1, xmm2, imm | 15 | 10 | 10 | 11 | 5 | 4 | 4 | 4 |
| PCMPESTRM xmm1, xmm2, imm | 10 | 10 | 10 | 11 | 6 | 5 | 5 | 4 |
| PCMPISTRI xmm1, xmm2, imm | 15 | 10 | 10 | 11 | 3 | 3 | 3 | 3 |
| PCMPISTRM 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 7-11 インテル® SSE4.1 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_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** |
| BLENDPD/S xmm1, xmm2, imm | 1 | 1 | 1 | 1 | 0.33 | 0.33 | 0.33 | 0.5 |
| BLENDVPD/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[b] | 5 | 5 | 5 | 0.5 | 1 | 1 | 1 |
| PMULLD xmm1, xmm2 | 10[c] | 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 7-12 インテル® SSSE3 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_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[b] | 5 | 5 | 5 | 0.5 | 1 | 1 | 1 |
| PMULHRSW xmm1, xmm2 | 5[b] | 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 |
| PABSB/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 サイクルのバブルを含みます

表 7-13 インテル® SSE3 浮動小数点命令

| 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_3A 06_3E 06_2A 06_2D** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E 06_2A 06_2D** |
| 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 7-14 インテル® SSE2 の 128 ビット整数命令

| 命令 | レイテンシー[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E 06_2A 06_2D** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E 06_2A 06_2D** |
| 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/PADDUSB /PADDUSW xmm, xmm | 1 | 1 | 1 | 1 | 0.5 | 0.5 | 0.5 | 0.5 |
| PADDQ/ PSUBQ[3] 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[b] | 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 |
| PMOVMSKB[3] r32, xmm | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| PMULHUW/PMULHW/PMUL LW xmm, xmm | 5[b] | 5 | 5 | 5 | 0.5 | 1 | 1 | 1 |
| PMULUDQ xmm, xmm | 5[b] | 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[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E 06_2A 06_2D** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E 06_2A 06_2D** |
| PSRLDQ xm m, 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 7-15 インテル® SSE2 の浮動小数点命令

| 命令 | レイテンシー[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E (06_3A 06_3E)** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_2A 06_2D (06_3A 06_3E)** |
| 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[1] | 14 | <14 | 14-20 | 16-22 (15-20) | 4 | 8 | 13 | 22 (14) |

| 命令 | レイテンシー[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_3A 06_3E (06_3A 06_3E)** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_2A 06_2D (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[2] | 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[3] 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 7-16 インテル® SSE の浮動小数点命令

| 命令 | レイテンシー[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_2A 06_2D (06_3A 06_3E)** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_2A 06_2D (06_3A 06_3E)** |
| 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[1] | 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[2] | 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[3] | | | | | 6 | 5 | 5 | 4 |
| MFENCE[3] | | | | | ~40 | ~35 | ~35 | ~35 |

| 命令 | レイテンシー[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_2A 06_2D (06_3A 06_3E)** | **06_4E, 06_5E** | **06_3D, 06_47, 06_56** | **06_3C, 06_45, 06_46, 06_3F** | **06_2A 06_2D (06_3A 06_3E)** |
| SFENCE[3] | | | | | 7 | 6 | 6 | 5 |
| STMXCSR[3] | | | | | 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 7-17 汎用命令

| 命令 | レイテンシー[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **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[12] | | | | | ~2-50 | ~3-50 | ~3-50 | ~5-50 |
| CLFLUSHOPT[13] | | | | | ~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[1] | | | | スループット | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | **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** |
| 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[10] | 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)[8] | | | | | ~85-100 | ~85-100 | ~85-100 | ~85-100 |
| IDIV r32[9] | | | | | ~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)[8] | | | | | ~85-95 | ~85-95 | ~85-95 | ~85-95 |
| DIV r32[9] | | | | | ~20-26 | ~20-26 | ~20-26 | ~19-25 |
| MUL r64[10] | 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[11] | | | | | ~98 | ~100 | ~100 | ~100 |
| XSAVEOPT[11] | | | | | ~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 |

## 7.3.2 表の脚注について

この節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 (uop) 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. 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。
3. 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。
4. 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。
5. 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。
6. 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、**エラー! 参照元が見つかりません。**ボリューム 1 の 3.4 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。
7. 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 プロセッサーとインテル® Xeon® プロセッサーに適用されます。
8. 「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。
9. 「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。
10. 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。
11. 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。
12. CLFLUSH のスループットは、バッファー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シュラインに関連する追加のコスト。ボリューム 1 の 9.4.6 節を参照してください。
13. CLFLUSHOPT のスループットは、バッファー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。ボリューム 1 の 9.4.7 節を参照してください。

## 7.3.3 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 7-4 ~ 表 7-18 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、**エラー! 参照元が見つかりません。**に記載されています。

### 7.3.3.1 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。

表 7-18 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 7-18 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 |

## 8.1 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。

この章では以下について説明します。

- マルチスレッド アプリケーションを最適化および調整して、ロック除去にインテル® TSX 命令を使用する推奨アプローチ。
- インテル® TSX 命令を使用してロック除去を実装し、ロック保護されたクリティカル・セクションの同時実行を可能にするガイドラインを提供します。これには、次の方法があります。
  - ハードウェア・ロック除去 (HLE)[1] の場合と同様のプレフィクス・ヒント。
  - RTM の場合と同様に、新しい命令。

表 8-1. 参考資料

| リンク先 | 説明 |
|---|---|
| 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) の特定のプロセッサーに対する更新を無効にする (英語) | 更新されたマイクロコードによるインテル® TSX の動作の変更 (MSR の動作を含む) に関する詳細を提供します。 |
| 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組込み関数 (英語) | インテル® C++ コンパイラー・クラシック・デベロッパー・ガイドおよびリファレンスのセクション。 |
| 第 16 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・エクステンションを使用したプログラミング」 | 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、第 1 巻に記載されているインテル® TSX インターフェイスの詳細。 |
| インテル® プロセッサー向けインテル® 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) メモリーおよびパフォーマンス監視アップデート (英語) | 最新のサポート、製品情報、ドキュメント。 |
| Linux* perf を使用したインテル® 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) プロファイリング (英語) | 新しい Linux* システムに統合されている Linux perf (または「perf イベント」) プロファイラーを使用した TSX プロファイリングについて説明します。 |

### 8.1.1 インテル® TSX について

インテル® TSX により、プロセッサーは次のことが可能になります。

- スレッドがロック保護されたクリティカル・セクションを介してシリアル化する必要があるかどうかを動的に判断します。
- 必要な場合にのみシリアル化を実行します。

- これにより、ハードウェアは、ロックの省略と呼ばれる手法を通じて、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ーからは瞬時に起こったように見えるようにします。プロセッサーは、コミットに成功した場合のみ、ほかの論理プロセッサー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ー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、プロセッサー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、プロセッサー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、プロセッサーは実行をロールバックし、プロセスはトランザクション・アボートと見なされます。トランザクションがアボートすると、プロセッサー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。

- **Hardware Lock Elision** (HLE) は、従来のプロセッサー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。
- **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・アボ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、論理プロセッサー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ボートする XABORT 命令も用意されています。

プロセッサーは、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ーと別のプロセッサー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に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ー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 I/O 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「Programming with Intel® Transactional Synchronization Extensions」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。

### 8.1.2 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 (8.2 節を参照)。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 (8.3 節を参照)、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 (8.4 節を参照)、そして初期実装のパフォーマンス・ガイドライン (8.5 節を参照) 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。

## 8.2 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 (8.2.1 節を参照) 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、8.3 節で、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 (8.4 節を参照) 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ス・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ー機能が使用します。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「Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features」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。

### 8.2.1 既存のインテル® 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。

#### 8.2.1.1 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

- 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依存します。
- 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。
- 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。

#### 8.2.1.2 プログラムの修正を必要とするライブラリー

- 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) 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。
- 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。
- 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。
- 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)。

### 8.2.2 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。

- 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、8.3 節を参照してください。
- 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と、本来のデータ競合に注目しやすくなります。

### 8.2.3 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 (8.4 節を参照)。

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。
- 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 (8.4.2 節を参照)。
- 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対応させる必要があります (8.3.7 節を参照)。
- 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。

- ロックをすべて特定します。
- 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。
- 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。
- 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。

### 8.2.4 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ー・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「Programming with Intel® Transactional Synchronization Extensions」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 (8.4 節を参照)。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。

- データアクセス競合によるアボート。
- ロック変数の競合によるアボート。
- リソースバッファーの超過によるアボート。
- HLE インターフェイス固有の制限によるアボート。
- 『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。

### 8.2.4.1 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ー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ます。

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 (8.2.4.1 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 8-1 を参照)。

例 8-1 条件付き更新とデータ競合の軽減

| | |
|---|---|
| `state = true; // 毎回更新`<br>`var \|= flag;` | `if (state != true) state = true;`<br>`if (!(var & flag)) var \|= flag;` |

### 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。

### 8.2.4.2 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。

プロセッサーは、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 (evisction) 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† 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†、Broadwell† および Skylake† 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ー間で共有されるため、同じコアの一方の論理プロセッサー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 (8.4.4 節を参照)。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 (8.2.6 節を参照)。

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にのみ発生します。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。

### 8.2.4.3 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の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 (8.3.5 節を参照)。

### 8.2.4.4 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。

### 8.2.4.5 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 (8.4 節を参照)。

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、I/O 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) 命令を使用するべきです。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。

ページフォールト (#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、プロセッサーが参照するページ・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、プロセッサー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。

### 8.2.5 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) ページフォー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。

### 8.2.6 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。

#### 8.2.6.1 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 I/O 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 8-2 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 8-2 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 */
```

#### 8.2.6.2 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。

#### 8.2.6.3 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も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。

## 8.3 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。

### 8.3.1 HLE プリフィクスの追加

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、プロセッサー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。

### 8.3.2 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。

- ロックの所有権フィールドの競合
- 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。

### 8.3.3 ロックの省略における HLE または RTM の使用

プロセッサーが 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、プロセッサーが 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ー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、プロセッサーが 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ーで実行すると、#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 情報をある変数に格納しておくと良いでしょう。

HLE では、トランザクション実行中のプロセッサーがクリティカル・セクションでロックの値を読み取ると、プロセッサー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 (8.3.4 節を参照)。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 (8.3.9 節を参照)。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。

### 8.3.4 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にのみ実行されます。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 8-3 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、8.7 節を参照してください)。

例 8-3 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 8-3 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ことがあります。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。

### 8.3.5 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()[2] 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 ビットを無視する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ができないために生じます (8.3.8 節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。

## 8.3.6 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。

### 8.3.6.1 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。

2 _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 8-4 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 8-4 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 8-5 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 <imminitrin.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、8.7 節を参照してください。

### 8.3.6.2 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とき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。

### 8.3.6.3 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 (8.3.4 節を参照)。

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。

### 8.3.6.4 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に見えますが、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が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。

### 8.3.7 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、8.3.6.2 節の reader-writer (読み取り/書き込み) ロックの実装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、(8.3.4 節の例に似た) 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 8-6 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 8-7 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。

### 8.3.8 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、処理が続行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 8-8 のように記述できます。

例 8-8 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 8-9 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。

### 8.3.9 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。

## 8.3.10 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 を使用する場合の動作は実装固有です。第 4 世代インテル® Core™ プロセッサーの初期実装では、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。以降のプロセッサーでは動作が変わる可能性がありますが、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、呼び出し関数があるかどうか、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 8-10 にこの概念を示します。

例 8-10 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
}
```

## 8.4 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 19 章「Architectural Last Branch Records」に、現在の実装でサポートされる各種パフォーマンス監視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™ プロファイラーを含むプロファイル・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) を参照してください。

### 8.4.1 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 (8.3 節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 (8.2.4 節を参照してください)。

### 8.4.2 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_UOP_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_uop_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のには効果的ではありません (8.3.7 節を参照してください)。また、MEM_uop_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。

### 8.4.3 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 プリサイス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。

### 8.4.4 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。

RelativeCostOfAbortForIP = SUM(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。ページフォールト (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ー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で、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。

### 8.4.5 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 8-2 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<br>(「15.3.4 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<br>(「15.3.9 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 (15.3.9 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。

### 8.4.6 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うことです。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、8.4.7 節を参照) または SDE ソフトウェア・エミュレーション (8.4.8 節を参照) 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。

### 8.4.7 LBR とトランザクション・アボート

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ボ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・ハンドラーはアボート時に 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。

### 8.4.8 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 (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・アボ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。

8.4.4 節で述べたとおり、アボ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、ハードウェアはアボ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 (競合でアボートされた) 向けに、競合するメモリー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 レジスターからアボ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・アボ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ボートします。これは、未解決のページフォールトや同様の操作が原因でトランザクション実行がアボートするケース (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ー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。

### 8.4.9 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、8.2.4.4 節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。

## 8.4.10 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_RETIRED.COMMIT + HLE_RETIRED.ABORTED
#RTM Regions Started: RTM_RETIRED.COMMIT + RTM_RETIRED.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_RETIRED.ABORTED/HLE_RETIRED.START)
%AbortedRTM = 100.0 * (RTM_RETIRED.ABORTED/RTM_RETIRED.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 8.4.1 節を参照)。

AvgCyclesInHLE = CyclesInTX/HLE_RETIRED_START
AvgCyclesInRTM = CyclesInTX/RTM_RETIRED.START
AvgCyclesInTX=CyclesInTX/(HLE_RETIRED.START + RTM_RETIRED.START)

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。

%AbortedHLEDataConflict = TX_MEM.ABORT_CONFLICT/HLE_RETIRED.START;
%AbortedRTMDataConflict = TX_MEM.ABORT_CONFLICT / RTM_RETIRED.START;
%AbortedTXDataConlict= TX_MEM.ABORT_CONFLICT / (HLE_RETIRED.START+RTM_RETIRED.START);

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。

%AbortedTXStoreResource = TX_MEM.ABORT_CAPACITY_WRITE

Broadwell† と Skylake† マイクロアーキテクチャーをベースとするプロセッサーでは、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ます。

%AbortedHLEResource = HLE_RETIRED.ABORTED_MISC1 - TX_MEM.ABORT_CONFLICT
%AbortedRTMResource = RTM_RETIRED.ABORTED_MISC1- TX_MEM.ABORT_CONFLICT
%AbortedTXResource = (HLE_RETIRED.ABORTED_MISC1+RTM_RETIRED.ABORTED_MISC5) - TX_EM.ABORT_CONFLICT

HLE_RETIRED.ABORTED_MISC1 は、8.4.9 節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。

## 8.5 パフォーマンスのガイドライン

第 4 世代インテル® Core™ プロセッサーはインテル® TSX をサポートする最初のプロセッサーです。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 プリフィクスのレイテンシーは、LOCK プリフィ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™ プロセッサー実装では、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† マイクロアーキテクチャー・ベースのプロセッサーでは軽減されています。

## 8.6 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 (8.2.5 節を参照)。

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 固有のものではなく、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ーを利用して、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。バッファー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® プロセッサー・トレースをサポートするプロセッサーでは、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); }
```

## 8.7 インテル® TSX 用の一般的な組込み関数

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。

### 8.7.1 RTM C 組込み関数

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べきです (『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 2A』の第 3 章「Basic Execution Environment」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。

- _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。
- _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。設定されていなければ、再試行しても成功する確率が低いままです。
- _XABORT_CAPACITY: 容量のオーバーフローによるアボートです。
- _XABORT_DEBUG: デバッグトラップによるアボートです。
- _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 8.4.5 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込み関数を有効にする必要があります。

### 8.7.1.1 古い gcc* 互換コンパイラーによる RTM 組込み関数のエミュレート

immintrin.h で RTM 組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 8-11 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 8-11 古い gcc 互換コンパイラーによる RTM 組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;
}
```

## 8.7.2 gcc およびその他の Linux* 互換コンパイラーの HLE 組込み関数

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。

- 32 ビットのターゲット上の 8 バイト値
- 16 バイト値
- 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
- __atomic_store と __atomic_clear は、__ATOMIC_HLE_RELEASE のみサポート

### 8.7.2.1 gcc 4.8 による HLE 組込み関数の生成

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。

### 8.7.2.2 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 アトミック組込み関数の制限が適用されます。

例 8-12 に C++ を使用した HLE 組込み関数の例を示します。

例 8-12 HLE 組込み関数の 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
   }
}
```

### 8.7.2.3 古い gcc 互換コンパイラーによる HLE 組込み関数のエミュレート

これらの組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 8-13 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。

例 8-13 古い GCC コンパイラーでの HLE 組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");
}
```

## 8.7.3 Windows* C/C++ コンパイラーの HLE 組込み関数

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 組込み関数を提供しています。例 8-14 を参照してください。

例 8-14 インテルと Microsoft コンパイラーによる HLE 組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。